w4112 Project 1

Cody De La Vara, Evan Drewry

June 21, 2013

Chapter 1

Introduction

The relational model, initially proposed by Edgar F. Codd in 1969, has dominated the world of databases.[16] The relational model was a revolution in data management, providing developers with a high-level way to describe and query their data. [16] As time progressed, hardware characteristics changed dramatically while new types of applications focusing on big data and analytics began to emerge. The change in hardware caused new bottlenecks in database performance to appear, driving researchers to re-evaluate the traditional relational-model paradigm as a general solution to all database needs. In particular, the great divide between modern CPU and disk bandwidth has inspired new types of database technologies to emerge that minimize disk latency while taking full advantage of the CPU. Scalability has also become a major concern for databases, pushing researchers to build new types of databases that can keep up with expanding businesses.

This report aims to accomplish three things: 1) to highlight the primary components of each emerging technology, 2) to compare different vendors and their key features, and 3) to provide a critical analysis of each technology and their importance. By examining each trend, we may get a better understanding of their applicability and improve our ability to make intelligent conclusions about future technologies.

Chapter 2

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.[25]*

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. [25] The traditional recovery algorithms must also be replaced since they assume persistent data storage.

2.1 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.

2.1.1 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.[25]

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.[43] 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.[21]

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.[43]

2.2 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

2.3 Vendor Analysis

The wide difference between certain vendors' target market is testament to the applicability of main-memory databases. While each vendor has different approaches to implementing main-memory databases, they all take advantage of the superior performance of modern CPUs to improve database system throughput. The following is an overview of each vendor.

2.3.1 IBM's Blink

Blink is an main-memory database system built for ad-hoc querying over massive fact-tables.[9] The key features Blink provides include row compression that uses a specialized dictionary encoding method, and a complex scan-only query executor that operates on compressed data. In comparison to a majority of traditional RDBMS, Blink does not have a performance layer that uses indexes or cached materializations to improve query response time since it can quickly scan all data in RAM. Blink's method of compression, titled *frequency partitioning*, involves partitioning the columns of a table based on data frequency; afterwards, each partition is compressed using a dictionary-encoding scheme with dictionaries built for each partition. [9] Blink claims that this compression scheme is better than column-store compression because column-stores have fixed-sized columns and must therefore pad data on the page, whereas Blink's dictionaries minimize the padding in each partition.[9] Blink has a unique process of executing queries since Blink has no indexes and therefore has no need to perform access-plan optimizations. [9] In practice, Blink has found success as a database *accelerator* rather than a stand-alone database system. [9] As an *accelerator*, Blink allows traditional disk-based database systems to bulk-load tables into a Blink cluster in order to route ad-hoc queries to Blink for real-time analytic performance. [9] As an accelerator, Blink claims to achieve 60x to 1400x decrease in query response time on a retail fact-table containing over 1 billion rows, and shows a near-uniform response time between 2-4 seconds in comparison to disk-based data warehouse [9] While these speedups are impressive, one must keep in mind that Blink has to be loaded with the data first, which may take considerable time.

2.3.2 H-Store

H-Store is a heavily-distributed OLTP database that keeps all data permanently in main-memory across multiple physical computers. [31] H-Store's focus on transaction processing is significantly different than Blink's focus on ad-hoc queries, though both architectures are centered around storing data in main-memory for performance reasons. Unlike Blink's lack of indexes and query-optimization, H-Store utilizes indexes for access-cost optimization and implements many layers of query planning and optimizing both in deployment and runtime. [31] The driving aspect behind H-Store is the use of distributed clusters to divide the storage of large OLTP databases across multiple servers which keep that data available in main memory. [31, 26] Since OLTPs are likely to have few distinct transaction procedures, H-Store clusters are deployed with administrator-defined transaction procedures to improve query optimization and evaluation time. [31] Once H-Stores are running, it may receive an SQL query via its API and sends the request to a transaction manager which serializes the query over multiple nodes and returns the result.[31] The developers behind H-Store have shown that significant OLTP bottlenecks such as locking, logging and buffer management can account for up to 25% of the overall instructions needed to execute a query. [26] In order to overcome the logging bottleneck while still maintaining full availability, H-Store avoids implementing REDO logs in favor for state-restoration using duplicates databases that can take-over in case of failures.[26] As for locking, H-Store avoids multi-threading in favor for a single-threaded engines that treat each processor in a multi-processor server as its own node.[31]

2.3.3 Exasol's EXASolution

Exasol's EXASolution is an main-memory database designed for analytics, incorporating the storage paradigm of a column-oriented database with the scalability of distributed clusters. [22] Like Blink, EXASolution was designed with business intelligence applications in mind, yet un-

like Blink, EXASolution has found successes as a standalone database management system.[22] EXASolution's general query execution strategy is to have multiple, shared-nothing nodes process a small part of the query locally, returning the results on a user-request basis. [22] Since EXASolution is column-oriented, it heavily compresses its data due to the nature of column-store blocks having highly similar data - a subject discussed in more detail in the next section. Another key feature EXASolution provides is a collection of "intelligent" main-memory algorithms that Exasol claims are capable of accessing data in nanoseconds, though they leave the details of these algorithms out of their technical literature. [22] Unlike H-Store, which stores data entirely on RAM, each EXASolution node writes data out to a local hard-diss, providing each node with the ability to recover from failure using a local raid technology. [22] Exasol claims that EXASolution is the "fastest and most scalable database in the world"[8], which is clearly an exaggerated statement considering how it is not intended to be a general-purpose RDBMS, yet EXASolution's TPC-H benchmark performances are indeed impressive.[8] In every size category, EXASolution outperformed the competition by considerable margins. [8]

2.3.4 RAMcloud

RAMcloud, built at Stanford, is a high-performance main-memory storage system that distributes data across thousands of servers, storing all data in DRAM for fast access.[32, 39] Unlike EXASolution, Blink and H-Store, RAMcloud is designed as a replacement for traditional disk-based relational databases that do not scale well for growing web applications, seeking to reduce random-access data latency that comes with retrieving data from a traditional disk-bound storage server. [39] RAMcloud boasts a uniform 5-10 μs data latency, however this is only within a specialized, highly tuned network, with actual data latency ranging between 300-500μs over standard ethernet/TCP networks. [39] In order to achieve these results, RAMcloud distributes data across 1000 to 10000 servers that have between 32-256 GB of DRAM to store data. [32] In terms of durability, RAMcloud provides a solution that combines DRAM replicas and writing to a persistent disk called *buffer logging*.[39] This approach is not as expensive as keeping multiple copies of each DRAM, but must sacrifice disk bandwidth in order to maintain DRAM-like read and writes, reducing overall system throughput. [39, 32] As well, RAMcloud's data-model is unique in that is represents data using indexable key-value pairs that stores unstructured data, allowing servers to be data-structure agnostic, increasing scalability. [39, 32]

2.4 Analysis

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.

2.4.1 Is it important?

Building main-memory database systems has been a beneficial challenge in the database research community, as the impressive data access times of RAM have inspired innovative ways to make RAM as ACID-complaint as possible. Moreover, it has spurred researchers to identify and quantify new bottlenecks in main-memory query execution in order to amortize query response time. [45] Main-memory databases have also expanded the use of column-store technology by incorporating many of the techniques found in column-stores in order to further improve system throughput.[22] On a larger scale, globalization has been pushing the demand for services availability, making distributed in-memory databases more appealing to companies that wish to provide robust service to customers around the globe.[41] In relation to cloud-computing, in-memory databases can drive cloud-computing services by providing a very scalable storage solution. [39] It seems that nearly any industry can find ways to utilize the speed of main-memory databases to improve productivity.

2.4.2 Can the benefits be quantified?

Main-memory databases have found their way onto TPC benchmarks, most notably TPC-H, where EXASolution leads most of the competition in queries per hour. [1] While we can easily point to fast random access times as the major benefit of main-memory databases, it seems that many of the other benefits are derived from other emerging technologies such as column-stores and MapReduce databases. Recall that IBM Blink was originally developed to be a standalone database, but has so far only been used as an analytics accelerator for more robust databases, and has future plans to become a more column-oriented database that does not enforce that all data be kept in main memory.[9]

2.4.3 Mature products?

Main-memory databases are among the most highly developed and researched alternative database solutions, and also among the oldest. So, there are a number of mature products available, both open-source and proprietary. Once again we can refer to EXASolution's impressive TPC-H benchmarks to argue that there are mature main-memory databases that successfully demonstrate the performance benefits.[1]

2.4.4 Should I invest?

While we believe main-memory databases will be replaced by more cache-conscious, robust in-disk databases, the hype surrounding main-memory databases is enough to justify an investment. It seems that as long as CPU performance continues to improve, there will be more main-memory databases thrown into the market boasting novel main-memory algorithms that take advantage of the latest CPU performance features. Moreover, the wide applicability of main-memory database systems makes it a safer investment.

Chapter 3

Column-Store Databases

The paradigm of row-oriented database systems has faced several performance barriers in recent years as businesses and research facilities have become interested in performing more ad-hoc queries in less time. As CPU processing speeds have continued to rise, disk IO latency still remains a critical bottleneck for many database systems.[47] One proposed solution for improving read performance is a new architectural design referred to as column-oriented database systems, or "column-store" for short. Column-store, in a nutshell, stores the attribute values for a column into disk pages as opposed to row-stores which store whole tuples into disk pages. Moreover, column-stores are particularly good at compressing data and working with compressed data to execute queries, allowing businesses to store more data on disk, therefore saving money.[5] The ability to read only the columns of interest from a database has piqued the interest of data warehouse, decision support, scientific data-mining and business intelligence application developers due to their ability to avoid reading in data unassociated with the current query.[27, 6] Researchers have been picking apart the components of classic column-store databases such as MonetDB and C-store in order to determine whether the benefits of column-store can be implemented in a row-store architecture with minimal consequences.[27, 13, 28]

3.1 History and Motivation

Before column-store, the NSM storage model, or nary storage model, was the storage model for row-oriented databases. NSM stored all n attributes of a record together on disk, and were generally considered the best approach for database systems.[17] Motivation towards a column-oriented database began in the 1970s when medical researchers were trying to use statistical analysis on patient records to improve care decisions, but were unsatisfied with query performance.[51] Searching for improved read performance, the medical researchers worked with computer scientists to develop a database bank for patient records that included "transposed" auxiliary data files for efficient retrieval. [51] 10 years later a new model titled the Decomposition Storage Model, or DSM for short, was developed as an alternative to NSM by storing attribute values in clusters on disk and using "surrogates", or primary keys, to explicitly represent a database entity.[17] While this model required twice as many writes for updates compared to NSM, the ability to compress column attributes as well as improved cache performance distinguished it from NSM.[17]

DSM introduced researchers to the advantages of vertically partitioned databases, sparking new research towards ways of improving tuple reconstruction and update performance in DSM databases, even at the expense of adding CPU overhead.[44, 11] The clear motivation for column-stores is to build a database that can avoid reading unnecessary data from disk without complex indexes or fine-tuning. In addition, breakthrough performance features of modern CPUs has

compelled database developers to design more CPU-conscious databases that harness modern CPU optimizations. [11]

3.2 Overview of Column-Store Architecture

Column-stores are *read-optimized* databases, marked beyond their vertical disk layout for their highly compressed data on disk and their use of vector processing for pipelining data. [47, 11] Column-stores seek to get the best out of lackluster disk bandwidth performance by storing compressed column data, rather than whole tuples, in disk pages. Moreover, it is not uncommon for column-stores to substitute slotted pages — a practice that is useful when updating tuples — for larger, denser pages since all column data will be fixed size and compressed. [27] All of these measures allow column-stores to read in more data at once and only those attributes of each record that the query needs.

Row-stores are not new to compression: they have benefitted from compression techniques such as Dictionary Encoding in reducing the size of tuples on disk.[5] So what makes column-store compression distinct? One of the major benefits of column-store databases is that all the data on a page belong to the same column, and are therefore highly likely to have strong similarities to physically adjacent data, allowing for a wider variety of compression algorithms to choose from which take advantage of homogenous data.[5] This reduces the distance between data on disk and reduces the amount of data passed to memory, improving overall system throughput.[5] Compressing data provides more space for the database system to store copies in different sort orders, providing the query optimizer with more access-plan options for each query.[47] While compression and decompression require additional CPU overhead, the trade-off for reduced IO time has been found to outweigh the cost of decompression and improve performance.[11, 5]Another key aspect of column-stores are their unique query executors that are designed to operate directly on compressed data and take advantage of modern CPU SIMD instructions allowing for heavy loop-pipelining optimizations. [11, 5]

The astute reader may have already realized that updates on a column-store must have to access each column separately to update a single record, which pales in comparison to row-stores. The major disadvantage of column-stores is their slow update performance and complex tuple reconstruction process, making them poor options for OLTP databases which have update-heavy workloads.[11] Popular open-source column-store "C-store"[47] introduced a novel solution to the write performance problem by including an additional bulk tuple loader named a write-store into the database system that is optimized for such procedures. In TPC-H benchmarks, column-stores have been shown to outperform row-stores[47, 7], yet there are few caveats to column-stores that deserve mentioning. One caveat is that column-store performance steadily decreases with each additional column that needs to be accessed, even to the point where column-stores are outperformed by row-stores if a majority of a tables' columns must be accessed. [27] Moreover, even if a query only has to access a few columns, it may still wish to project entire tuples. Just as column-stores are poor at updating, they are also poor at reconstructing tuples from multiple columns.[27]

3.3 Applications

If column-stores can't promise to outperform row-stores given any read-intensive query, in what situations would a column-store be beneficial? Imagine a retail company has a 10 million row, 100 column fact table that stores all the transactions from different stores, and they have business analysts who want to get information such as how many of product X were purchased

in comparison to Product Y, or the total number of purchases above $100 that happened over the weekend. Now, the queries that can answer these questions will likely only need data from 10 of the 100 columns to execute the query. In a row-store, we would have to read in all attribute values for a tuple, wasting disk bandwidth on irrelevant data. In a column-store, the amount of irrelevant data read in is minimized by only reading the column data relevant to the query. The main idea behind this example is to show how column-stores are a strong match for ad-hoc queries that only need to touch a few columns of a massive database. The column-store architecture has clear commercial value as more and more enterprises entered the data-mining and business analysis markets.[29] Data centers and corporations are interested in improving the efficiency of read scans because they run data-warehousing and business intelligence applications that read large fact-tables to gather analytics. [27] As previously noted, medical researchers often depend on data analysis of medical records for making better diagnosis, and can easily benefit from a technology that allows them faster access to information that will help them treat patients better. [51]

3.4 Vendor Analysis

Vendors such as Sybase IQ, MonetDB, VectorWise, Vertica, and Kx Systems have developed different approaches towards implementing column-store databases. Many of these vendors claim to be built "from the ground up", with some boasting benchmark results showing an impressive 270x query response time speedup.[11, 30] They primarily target business intelligence and decision support applications, focused on providing low latency analytics over massive datasets.[11, 30, 49, 18] The following is an overview of each vendor.

3.4.1 SybaseIQ

Sybase IQ was one of the first commercial column-store database systems on the market, designed primarily for queries over massive fact-tables that touch few columns but many rows.[36] For improved disk bandwidth, Sybase IQ increases page size and compresses data in order to fit as much data onto a page as possible.[36] SybaseIQ columns can have multiple indexes because data is amortized through compression, allowing the query executor a larger search space for calculating access plans. Like other column-stores, SybaseIQ tries to choose the best compression scheme for the database, but always to compress the index records into segmented bitmaps, which are used by the query engine to quickly resolve where clauses using boolean operators.[36]

3.4.2 MonetDB

MonetDB is a vertically fragmented database that focuses on improving CPU efficiency by taking advantage of optimizations such as loop pipelining.[11] MonetDB's key characteristics are its use of lightweight data compression, late materialization, and vector processing.[11, 46] MonetDB claims that the lack of IPC efficiency in row-stores is due to extended query expression calculation and lack of loop pipeline.[11] Columns are represented on disk using Binary Association Tables (BAT), which are two-column tables with the first column representing the *oid*, or object id, and the second column is the attribute value. The X100 Vectorized Query Processor is MonetDB query engine that is cache-conscious, storing data vertically in the cache in order to take advantage of the loop-pipelining compiler optimizations.[11]

3.4.3 Vertica

Vertica is the commercial version of the open-source C-Store database.[3] Compared to MonetDB and Sybase IQ, Vertica supports a hybrid-store architecture that contains a main-memory write-store optimized for updates and inserts and a read-store optimized for ad-hoc queries. [47, 30] Both the write and read store are column-oriented, allowing for the stores to share one column-oriented query executor. The write-store moves tuples asynchronously to the read-store, providing for decent update and insert performance.[46] In their technical overview, Vertica claims to have an "aggressive" column compression scheme that on average can achieve a 90% reduction, but clearly this depends on the type of data stored, its sort order and the number of distinct values it has, ie. its cardinality.[30] Vertica samples column-data in order to determine the best compression scheme in order to provide enough room for multiple "projections", or copies of the column data in different sort orders.[30] Like Sybase, Vertica also uses multiple projections to optimize query execution. [36]

3.4.4 VectorWise

VectorWise claims to be "unique" because it takes full advantage of CPU performance features overlooked by competitors such as using SIMD (single instruction, multiple data) instructions to process vectorized data, however other column-stores like MonetDB provide similar SIMD support. [18] [46] In order to increase system throughput, VectorWise isolates all CPUs and caches to avoid memory contention. [18] VectorWise also provides support for row-oriented tables as a disk-conservative alternative for wide-tables with few rows, which can be beneficial for database administrators who want to further tune a VectorWise database.[18] Similar to Vertica, VectorWise includes an in-memory data structure for supporting efficient updates and inserts, called a Positional Delta Tree.[18] The amount of memory allocated for the PDT can also be tuned for performance. Unlike other column-oriented databases, VectorWise supports the use of storage indexes, which provide the minimum and maximum values of each page that helps avoid reading unnecessary pages from disk.[18]

3.4.5 Kx Systems

Kx Systems have designed their column-store database kdb+ to gather and mine both historical (disk) and real-time (RAM) data efficiently, ideal for investment traders and risk management.[49] Handling real-time data in a column-store is a challenge due to its suboptimal insert and updates architecture. In order to provide real-time performance, kdb+ maintains an in-memory database of new data.[49] As well, kdb+ is built to support high parallelism across distributed databases, providing core support for UUIDs as well as a publish and subscription mechanism for load-balancing.[49] Unlike other competitors, kdb+ provides their own programming language "q", a vector processing language intended to compliment a column-oriented database.[49]

3.5 Analysis

Over time, we should expect to see more mature advances in hybrid row-column stores that fuse the best of both technologies. Key aspects of column-stores such as vectorized processing, columnar layout, and heavy compression have already found their way into in-memory databases such as Exasol's EXASolution, and will likely continue to be adopted into other database management systems that want to improve ad-hoc query support. In fact, database companies such as Teradata have already released an analytic platform named Aster that allows for both

row and column oriented tables, signaling a shift towards a commercial hybrid-store technologies. [2] Research has failed to show that row-stores and column-stores have exclusive features, suggesting that an innovative hybrid-store is not impossible. [7] The TPC-H benchmarks[1] show that the in-memory column-store EXASolution outperforms non-clustered VectorWise databases with twice the queries-per-hour, suggesting that column-stores have a strong future within in-memory database systems. The two database types work well since both have been used to power real-time analytic applications and both focus on achieving the best performance out of modern CPUs.

3.5.1 Is it Important?

Column-stores have been beneficial in demonstrating the performance improvements one can get achieve using a domain-specific database solution instead of a general-purpose DBMS. The lack of disk IO speed improvements over the last couple decades has become a massive bottleneck for businesses and corporations that have petabytes of records that they want to gather statistics from. Big Data is arguably one of the most popular trends in computer science today, yet in order for analysts to get the most out of massive datasets, they require technologies like column-store to scan over billions of rows in a reasonable amount of time. When we look back at the TPC-H benchmarks for the 3,000GB and 10,000GB range, we find analysts-driven databases like EXASolution outperforming traditional databases by considerable scales. [1] For analytic companies, this means they can get more out of their data faster by adopting these technologies.

3.5.2 Can the benefits be quantified?

The typical way to calculate the benefits of a column-store is to compare the performance of ad-hod queries on large star-schema datasets (databases with large fact-tables and small dimension tables) between column-stores and on traditional DBMSs. Since column-store technology is a domain-specific solution for read-intensive applications, it makes sense to compare column-stores performance to row-store performance using decision support benchmarks such as TPC-H. That said, you shouldn't abandon your DBMS in favor for a column-store to speed up read performance in your application. There are a few things to keep in mind when choosing a column-store: is your fact table *large* and *wide*? Do your queries only touch a small portion of your tables columns? Is your data added in real-time, or do you load records in bulk? Research has shown that tuple-width and query selectivity have significant effects on column-store performance depending on their values, marking queries that access over 50% of a table's columns as having too large of a CPU overhead to compensate for their IO performance. [27]

3.5.3 Are there mature products based on this technology?

There are certainly mature column-store databases such as Sybase IQ and MonetDB which have been around for nearly two decades. VectorWise is another mature column-store which has lead most column-stores in TPC-H benchmark performance. [1] Gartner's 2012 "hype cycle" shows column-store databases reaching a plateau of expectation within the next two years, which means that column-stores are a "demonstrated and effected"[10] technology.

3.5.4 Should I invest?

Since disk IO performance is unlikely to dramatically improve in the coming years, column-stores will remain a valuable technology for business intelligence companies or data warehouses that wish to examine all their data quickly. There is considerable room for improvement within the

column-store architecture that we can expect to see in the future, such as improved write and update performance time and tuple reconstruction performance. If disk-based column-stores can reach write and update performances that can start to compare to traditional OLTP DBMSs, then a new paradigm in OLTP column-stores will inevitably begin to emerge, opening up a new set of potential column-store customers. Researchers have already began investigating the possibility of using in-memory column-store databases for OLTP applications, predicting that large enterprises will use in-memory column-stores for everything from business transactions to ad-hoc queries. [42]

Chapter 4

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.[35] 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.

4.1 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."[14] 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.[48] 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.[48] Most of the other characteristics that have come to define NoSQL data stores are simply consequences of this primary goal.

4.1.1 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.[48] 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).

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."[15] HBase, HyperTable and Apache's Facebook-developed Cassandra are other notable extensible record stores.

4.1.2 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.[12] 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.[12, 48] 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.

4.2 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.[19]

MapReduce is also the name Google gave to their widely mimicked implementation of the MapReduce model.[19] 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.[52]

4.2.1 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.[19]

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."[19]

4.2.2 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.[19]

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

4.3 Vendors

4.3.1 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.[20] 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.[48] 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.[20] 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.[38] 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."[20]

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.[4, 40] 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.[40]

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.[40]

4.3.2 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.[37, 4] The name comes from the word "hu**mongo**us", as the system was designed with scalability and speed in mind.[38] 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.[48] 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.[38]

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.[48] 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.[38]

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.[38]

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.[24] 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."[48]

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.[48]

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.[38]

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. [38]

4.3.3 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.[33] 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.[15] 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.[34, 50, 33]

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."[15] 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.[15]

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.[15]

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. [15]

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.[34, 23] 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.[34]

Cassandra's design combines the BigTable data model with the built-in infrastructure and eventual consistency properties of Dynamo.[23] 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.

4.4 Analysis

As we have seen, there is huge variation across the different NoSQL data models and implementations that exist today. Given that they were each designed around the motivation to fit a specific set 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 every day.

4.4.1 Is it important?

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.

4.4.2 Can the benefits be quantified?

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.*[16]

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.

4.5 Conclusion

Database technology has adapted to new trends in business and research that demand scalability, availability and performance over massive databases. As these technologies continue to mature, we can expect to find more hybrid technologies on the frontier that combine the best of in-memory, column-oriented, NoSQL, and MapReduce technologies. We discussed In-Memory Databases which take advantage of the RAM access speeds and CPU processing capabilities to provide uniform data access times. As well, we described the column-store paradigm that organizes data by column on disk, rather than by tuple, and uses heavy compression to maximize disk bandwidth for ad-hoc queries. Finally, we examined NoSQL and MapReduce technologies that provide big data with horizontal scalability and availability using distributed clusters. Among these technologies, NoSQL databases are the most interesting for their emphasis on handling unstructured data at a time when the internet provides a plethora of unstructured data ripe for analysis.

Bibliography

[1] Tpc-h benchmark, April 2012.

[2] Aster database: Big data analytics for the enterprise, April 2013.

[3] C-store: A column-oriented dbms, April 2013.

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

[5] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682. ACM, 2006.

[6] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, 2009.

[7] Daniel J Abadi, Samuel R Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 967–980. ACM, 2008.

[8] Exasol AG. Tpc-h benchmark, June 2009.

[9] Ronald Barber, Peter Bendel, Marco Czech, Oliver Draese, Frederick Ho, Namik Hrle, Stratos Idreos, Min-Soo Kim, Oliver Koeth, Jae-Gil Lee, Tianchao Tim Li, Guy M. Lohman, Konstantinos Morfonios, René Müller, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Richard Sidle, Knut Stolze, and Sandor Szabo. Business analytics in (a) blink. *IEEE Data Eng. Bull.*, 35(1):9–14, 2012.

[10] Andreas Bitterer. Hype cycle for business intelligence, 2012. Gartner, 2012.

[11] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR), Asilomar, CA, USA*, pages 225–237, 2005.

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

[13] Nicolas Bruno. Teaching an old elephant new tricks. *arXiv preprint arXiv:0909.1758*, 2009.

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

[15] 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.

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

[17] George P Copeland and Setrag N Khoshafian. A decomposition storage model. In *ACM SIGMOD Record*, volume 14, pages 268–279. ACM, 1985.

[18] Ingres Corporation. Vectorwise - simply fast: A technical whitepaper. Technical report, 2011.

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

[20] 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.

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

[22] Exasol. Exasolution technical white paper. Technical report, 2010.

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

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

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

[26] Stavros Harizopoulos, Daniel J Abadi, Samuel Madden, and Michael Stonebraker. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 981–992. ACM, 2008.

[27] Stavros Harizopoulos, Velen Liang, Daniel J Abadi, and Samuel Madden. Performance tradeoffs in read-optimized databases. In *Proceedings of the 32nd international conference on Very large data bases*, pages 487–498. VLDB Endowment, 2006.

[28] Allison L Holloway and David J DeWitt. Read-optimized databases, in depth. *Proceedings of the VLDB Endowment*, 1(1):502–513, 2008.

[29] Marcel Holsheimer and Martin L Kersten. *Architectural support for data mining*. Stichting Mathematisch Centrum, 1994.

[30] Vertica Systems Inc. The vertica® analytic database technical overview white paper. Technical report, 2010.

[31] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[32] Ankita Kejriwal. Ramcloud: A low-latency datacenter storage system. Stanford, 2013.

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

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

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

[36] Roger MacNicol and Blaine French. Sybase iq multiplex-designed for analytics. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 1227–1230. VLDB Endowment, 2004.

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

[38] 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.

[39] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

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

[41] Massimo Pezzini, Donald Feinberg, Andrew Norwood, Roxane Edjlali, Joseph Unsworth, and James Richardson. Predicts 2013: In-memory computing: Growing gains, but also growing pains. 2012.

[42] Hasso Plattner. A common database approach for oltp and olap using an in-memory column database. In *Proceedings of the 35th SIGMOD international conference on Management of data*, pages 1–2. ACM, 2009.

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

[44] Ravishankar Ramamurthy, David J DeWitt, and Qi Su. A case for fractured mirrors. *The VLDB journal*, 12(2):89–101, 2003.

[45] Kenneth A Ross. Selection conditions in main memory. *ACM Transactions on Database Systems (TODS)*, 29(1):132–161, 2004.

[46] Peter Boncz Stavros Harizopoulos, Daniel Abadi. Vldb 2009 tutorial: Column-oriented database systems. vldb, 2009.

[47] Mike Stonebraker, Daniel J Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, et al. C-store: a column-oriented dbms. In *Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

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

[49] Kx Systems. The 21st century time-series database. Technical report, 2013.

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

[51] Stephen Weyl, James Fries, Gio Wiederhold, and Frank Germano. A modular self-describing clinical databank system. *Computers and Biomedical Research*, 8(3):279–293, 1975.

[52] 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.