1 Answer 1 - Relational vs NoSQL Data Model

The key differentiator between the two models of relational and NoSQL databases is in the principles of the CAP theorem which the solution prioritises. Relational databases (in which data is structured rigidly in hierarchal n-tuples and relations) conform to the ACID management model; this model prioritises the consistency and accuracy of the database by strictly enforcing that data is consistent and adheres to the predefined structure and schema, demands that transactions which occur bring the database from one valid state to another or halt and are reverted, that transactions are isolated from other transactions and behave essentially sequentially, and that data for completed transactions should persist in the event of an outage. This model is a vertically-scalable model (as more storage, processing power, and memory can be added to the single database) which has been the leading paradigm of data management since its inception in the 1970s. In contrast, NoSQL databases (in which data may be stored as files, key-value pairs, graphs, or flexible-wide columns) largely implement the BASE model (with some exceptions such as Neo4J). This model trades the consistency of traditional databases for rapid implementation, horizontal scalability, and significant flexibility as it drops many of the strict requirements of ACID. In this system it instead demands that data should be basically available by distributing it amongst other nodes (instead of being immediately consistent) which protects data horizontally in the event of an outage of one node (as some data may be unavailable but most data should be accessible), that the data may change over time as the state of data is soft and nodes do not need to be immediately consistent, and the database's state only needs to eventually converge [3].

The decision to implement a NoSQL vs relational solution is domain dependant and there is not a universal situation where one is necessarily better than the other but instead it depends on the key project priorities as to which is more suitable; broadly, in financial contexts and other contexts where consistency and persistence is a necessity, or where data has a natural and set hierarchy are better suited towards RDBMS solutions, while scenarios where these are less important to scalability and flexibility are better suited towards NoSQL solutions.

In the financial services industry, the soft state and eventual consistency of NoSQL solutions make them largely non-viable for solutions. Drawing upon experience working in Data Science in EY, the consistency, accuracy, and need for data persistence over time to facilitate proper auditing renders the compromises of NoSQL unviable as the key transactional data store. Similarly in the banking industry the negative impact of non-consistency (where a customer of a bank may withdraw money or deposit money and not have the balance immediately update due to eventual consistency) renders these untenable as the backbone of data processing. While these industries may benefit from some of the scalability BASE allows, techniques such as sharding for relational databases exist to help offset the predisposition towards the vertical scaling of RDBMS model into a 'more' (although imperfect) NoSQL-like structure. A non-industry specific and more wide-reaching (but less technical) aspect of the RDMBS model which I believe is sometimes overlooked is the long-standing implementation of this model and thus the level of data professionals with long-standing experience in SQL; due to the newer nature of NoSQL, development skills in NoSQL are largely less developed (professionals with a maximum of 12 years experience vs RDBMS's max of 40) and the ease of hiring in data roles with a lower technical barrier may also be an improtant factor in the choosing of an RDMBS solution.

Outside of the financial services industry however, there can be significant benefits towards a NoSQL solution over a relational solution. The non-strict requirement for a data structure suits the flexibility and rapid-implementation of AGILE development methods without the need for a long discovery phase modelling the data requirements. The horizontal scaling by distributing data across nodes fits towards the modern movement towards cloud-based and distributed systems and provides a largely fault-tolerant model. The speed of queries not requiring joins fits the consumer expectations for performance of modern applications. For analytics and data applications, in contexts where strict (100%) accuracy is not required but large data volumes may be present such as customer analysis in retail (as speaking from my personal experience a percentage of error is normally expected due to data capture methods anyway), the flexibility of the data structure and ability to capture non-hierarchal data (such as photos and videos, customer survey and NPS data, store-topology for cross-selling and cannibalisation analysis, and web session and interaction data) NoSQL databases may be an opportune scenario (and indeed this type of big data capture played a key role in driving the development of these systems). For web applications where data sensitivity and immediate consistency is non-essential, I believe NoSQL solutions can be optimal and align more closely with modern practices. Ultimately, I think both solutions can coexist and complement each other, however it is undeniable that the dominance RDMBS's have had over the last 40 years will, at a minimum, be diluted as NoSQL solutions are increasingly adopted in a world of more complex and larger data.

2 Answer 2 - Spark and Hadoop

Hadoop is an open-source framework from Apache for the processing and analysis of massive dataset through the implementation of map-reduce which allows for a cluster of computers or servers to process large datasets in parallel. While Hadoop has many sub-modules to handle various components of a big data workflow, such as its scripting module Pig, Hive for data warehousing, and Zookeeper for the coordination of distributed systems, two of the key components of the Hadoop project are the Hadoop Distributed File System (HDFS), a distributed file system for application data, and MapReduce for the parallel processing of data in batches.

The HDFS is the system for distributed storage for Hadoop and consists of a NameNode and DataNodes. Data is partitioned into multiple blocks, and each block is then replicated a selected number of times (three by default; this redundancy is required to ensure basic availability in the event of a node failure) and distributed across the DataNodes. The NameNode then manages the overall operations of the DataNodes (i.e. block operations) and stores metadata related to the data. Due to the replication of data blocks, there is an inherent 'wastage' of storage however as storage is (currently) relatively cheap, creates to fault tolerance, and allows for the usage of multiple cheap devices instead of powerful machines, for large organisations with massive data volumes this additional storage is comparatively insignificant. The MapReduce framework is an algorithmic process, consisting of the Map step and Reduce step, to allow for the distributed computation and processing of large data volumes. An input (a subset of the data volume broken into key value pairs by the record reader) is passed to a Map function and breaks this data into key value pairs (this process will be done in parallel across nodes for the dataset). Once the map process is complete across all nodes, intermediate key value pairs are produced which pass into a partitioner that exchanges values between nodes and assigns a reducer to process that 'chunk' of data and data assigned to a reducer is then sorted. The second phase of this process is the Reduce function, which takes this data and performs some operation (e.g. summing, counting, etc.) and produces and consolidates final key value pairs.

Spark is also an open-source framework, developed in UC Berkely, which originally started as an extension of the Hadoop framework but can also operate independently of Hadoop, which also allows for the distributed processing of large datasets and supports iterative processing by extending beyond mapreduce tasks and batch processing (e.g. via the support of streaming data), with extensions to aid in ease of use such as a query editor Spark SQL and MLLib for machine learning tasks, while also leveraging in-memory processing rather than just using file processing. Spark differs from Hadoop in that it does not come with its own file storage system allowing it to plug into multiple management systems (e.g. Hadoop's HDFS). The key data structure, and a driver of some of the main differences between Spark and Hadoop, is the Resilient Distributed Datasets (RDDs).

RDDs are an immutable, distributed collection of objects which can be worked on by different nodes in the cluster. These RDDs can be persisted in distributed memory (or on-disk if the memory is insufficient to store an RDD), and Spark distributes partitions of the RDDs to nodes to process them in parallel, which allows fast access to data, and states are shared between jobs which allows for iterative processes (without intermediate storage as would be required for Hadoop). Using RDDs, multiple iterations of a map-reduce task can be run.

Through the usage of RDDs vs. HDFS's, while Hadoop and Spark can be considered complimentary (as Spark can sit on Hadoop), the key benefits of Spark are that Spark allows for in-memory usage and state persistence. By storing data in memory, iterative processes are greatly sped up (jobs can be between 10 to 100 times faster in Spark vs Hadoop) by reducing the number of read-write processes and allowing for more efficient processing. This plays a key advantage in the implementation of more advanced algoritms such as K-Means clustering and and working with more complex data structures where iterative processing is required (e.g. graphs and graph processing), while also allowing for the efficient processing of real-time streaming data (whereas the HDFS structure of Hadoop is designed for batch processing).

While Spark is a more modern technology, aimed at addressing some of the bottlenecks of Hadoop, there are instances where Hadoop may be more suitable for use than Spark. As Spark primarily speeds up processing by saving data in-memory rather than using file storage, Spark benefits from having nodes in its cluster with more RAM and this can result in a more costly cluster setup than what Hadoop requires (due to the cost of RAM vs disk memory). If cost is a prohibitive requirement, businesses may choose to implement a Hadoop cluster over Spark. Similarly, if a business is sufficient with batch processing (and does not require streaming data) over 'simple' data structures, and is only performing basic processing, Hadoop may be suitable for that business. Finally, as Hadoop is a framework which has been

established for a longer time period, it is feasible that a business may choose to maintain an existing Hadoop cluster if, in addition to some of the above conditions (low cost is a requirement, batch processing is sufficient, only basic analytics requirements aimed at reporting rather than prediction or personalisation), there is more expertise or familiarity with Hadoop within their organisation. For these reasons - the cost of implementation, low analytics requirements for data, speed of processing not being a priority, familiarity, and minimal work with advanced data structures - one may choose to implement Hadoop in place of Spark.

Yet while there are some reasons a business may choose to focus on Hadoop over Spark, there are many and significant benefits for businesses which would drive the business to choose Spark over Hadoop. A business which is seeking a more fully-fledged analytics engine, with fewer cost constraints, would find many advantages to choosing a Spark-based workflow over one which solely leverages Hadoop (obiously hybrid workflows are also feasible). The most obvious and arguably significant reason as to why an organisation would choose to use Spark in place of Hadoop is instances where processing speed is critical. As RDDs are highly efficient and leverage a nodes memory to process data, processes implemented in Spark can be many times faster than Hadoop. When working with large datasets, where companies may be dealing with petrabyes of data, this rapid processing of data can be essential and is a clear reason to choose Spark.

Even beyond the processing speed improvements which Spark brings, Spark has extensive API support which naturally fits into many modern languages and workflows including Scala, Python, and Java. This allows businesses to interact with Spark through some of the most common and widespread backend and analytics languages which could fit neatly into the businesses existing codebase and infrastructure. The MapReduce functionality in Spark is extended to capture and optimise many desirable features such as an interactive query engine and stream processing while allowing for easier implementation/quicker access of algorithms in comparison to Hadoop (the amount of code required to implement WordCount in Hadoop and Spark an easily) makes it enticing to developers and analysts who wish to rapidly develop features and the ability to interactively work with data in the Spark Shell (in comparison to Hadoop's MapReduce) allows for easier and more instaneous feedback on queries and processes. The widespread module support capturing areas such as graph processing, machine learning, and the overall flexibility of Spark (being easily able to slot into multiple existing distibuted file management systems) not only lessens the risk of becoming vendor dependant but also allows for the potential of having a more consistent codebase lessening the requirement for resources with experience across platforms and frameworks which may be desirable (and more efficiently processes algorithms for which iterativation is required). Data is increasingly complex, unstructured, and volume-heavy, and businesses are engaged in an 'arms race' to enhance their data processing capabilities to optimise revenue and enhance user personalisation; as Spark supports streaming data, is better equipped to handle complex data types such as graphs (through GraphX), and implementation of Spark-based workflows are growing rapidly, the final reason why a business may choose Spark over Hadoop is to better prepare for where the market is moving in recent years. While new technology may be developed that will eventually cap the popularity of Spark, arguably a business is positioning themselves better for future requirements the business may have by focusing on implementing a Spark-focused workflow.

While there are certainly reasons to choose Hadoop in place of Spark, and vice versa, as a final note it is worth once again emphasising that these are not mutually exclusive as Hadoop's HDFS' can be used as the distributed file system for Spark, while Spark (and the extension of MapReduce provided by the Spark Core API) can be used for the benefits which it brings. For scalable distirbuted analytics systems, this co-usage can be essential. As Spark does not have its own file system, as the volume of data grows considerably Spark needs to be integrated with another file system (such as HDFS). For this reason it makes sense to view aspects of Hadoop and Spark as (potentially) complimentary, rather than inherently competing data frameworks as the two can co-exist within a businesses data workflow.

3 Answer 3 - Cassandra: A Decentralised Structured Storage System

The research paper I have chosen from the Brightspace list is "Cassandra - A Decentralised Structured Storage System" by Lashman and Malik. In this paper, the authors detail a distributed storage system named Cassandra, which is awide-column based implementation NoSQL system, which is designed to handle high write volume and read efficiency while simultaneously providing continuously failing components. The key driver of the development of Cassandra, as outlined by the authors in the paper, is that Facebook maintains an incredibly vast array of data centers located across the world, with hundreds of millions of users simultaneously accessing the platform. Due to the massive volume of both active users and network infrastructure required to support this volume, it is typical for there to always be at least some servers or infrastructural component which are failing. Because of the expectation that some elements of the infrastructure will fail, there is a need for software solutions and backend infrastructure to be able to cope and expect systems to be unreliable (or have failures) and be highly fault tolerant. For this reason, combined with the user volume, Facebook developed Cassandra to serve as the backend of its Inbox Search feature and subsequently deployed it on other projects within the Facebook ecosystem - as a lengthy aside, it is worth noting, although not mentiond in the paper, that the rollout of Facebook's new messenger post-2010 resulted in the decomissioning of Cassandra as the Inbox Search Backend as the paper describe, and it was replaced with HBase, however Cassandra is used as a backend in Instagram and a number of other large companies such as Netflix have adopted it since this paper.

In deciding upon their implementation and in designing their system, the authors look to other distributed storage systems such as Ficus, and Coda which replicate files trading availability for consisteny and manage discrepencies across nodes with specialised conflict management processes. They note that Farsite does not use a centralised server and manages to achieve availability and scalability with replication of data. The Google File System is detailed however the authors note that the original implementation is weakened by the GFS master server creating a single point of failure but has been reinforced using the Chubby abstraction to add fault tolerance. Finally, they highlight Bayou which is a distributed RDBMS which provides eventual consistency and allows disconnected operations. They note that out of these systems, only Bayou, Coda, and Ficus meet the need to be resilient to infrastructure challenges while simultaneously allowing disconnected operations, and each of these force eventual data consistency using custom conflict resolution processes. The authors detail that while relational databases do exist, their enforcement of strong consistency limits their scalability and availability making them inadequate for Facebook's requirements. The authors finally compares the three aforementioned file systems to the storage systems which serve as arguably the greatest influence on the development of Cassandra - Amazon's Dynamo and Google's BigTable. The key issue which Dynamo provides is that lthough it allows operations during infrastruture failures and uses conflict resolution techniques to resolve discrepencies across nodes, it is limited by requiring a read operation whenever a write operation is performed as it manages resolutions using a clock system which renders it unsuitable for high write throughput. The key issue which the authors find with Google BigTable is that although it does have the sought structure and data distribution, the reliance upon the file system for durability renders it as unsuitable. While the authors find many implementations for distributed storage, these all ultimately share key challenges where they are reliant upon distributed file systems, they implement custom (and sometimes unsuitable) conflict resolution protocols, or fail to be sufficiently decentralised to cope with Facebook's volume and fault-tolerance requirements.

In order to develop a system which meets Facebook's requirements while alleviating some of their concerns with existing solutions, the authors first of all develop a wide-columed NoSQL data model similar in structure to BigTable (row with a structured tuple) grouped into Simple Columns and Super Columns (a family of columns within a family of columns) and all operations against a row key are atomic, and an application can specify the ordering of columns to be time-based or name-based allowing for rapid retrieval of message data which is inherently time ordered. There are three operations which are supported, an insert, get, and delete, and although multiple tables are feasible the authors' deployment only uses a single table. The system is then designed such that Cassandra implements five key modules; a parititoning system, a replication system, a membership and failure system, and a scaling system. The paritioning system partitions the data across nodes using an order-preserving hasing where each node is assigned a consistent hash in a ring of hashes, and when data is hashed it is assigned to the node with a position larger than the item's position. by consistently hashing and slotting in data in this manner, a node entering or existing only affects the nodes on either side of it within the 'ring'. This system introduces challenges by potentially resulting in imbalanced data and load distributions, and Cassandra attempts to resolve this by analysing load information and moving lightly populated nodes

into the ring. This results in a distributed system of storage hosts. The data is replicated using the replication system to achieve the authors' availability and durability requirements. The coordinator node (the node to which data has been assigned based upon its key's position in the partitioning ring) is responsible for replicating all data items within its range by storing it both locally but also at the replication factor - 1'th nodes in the ring ('wrapping' appropriately) and giving additional options to the user based on their data center setup. Apache Zookeeper is used to elect a leader node within the ring which tells new nodes what ranges they are replicas for and ensuring no nodes manage more than replication factor - 1 ranges and this info is then cached both at each node and then also inside Zookeeper to keep this information immune to crashes or outages. As each node is aware of the other nodes in the system, and therefore what they are responsible for, the system is highly durable against failures and allows for replication across data centers. The membership and failure system ensures a node transmits data on whether they are up or down to prevent communication with failed nodes, and membership in the cluster is managed using the Scuttlebutt algorithm. When nodes start, they choose a random token which is stored locally and in Zookeper, and then transmitted to the cluster. The adding or reoval of a node is initiated via a manual process overseen by an administrator (since nodes rarely should depart from clusters if they're down for a lengthy time period). For the persistence and storage of data locally the local file system of the node is used and data is saved onto the disk in a manner which is easily retrievable. Data is logged and then added to an in-memory data structure and separate disks on each machines are present for the commit log. Data is dumped to disk from memory once a threshold is reaches and persistently indexed for efficient lookup, and as time passes a merge process runs on the backup to combine multiple files into a singular file as in BigTable. Data is retrieved by querying in memory data and then file data in order from newest to oldest. A filter is present to prevent data from files being searched which don't contain the key to optimise the efficiency of the read processes.

The authors don't particularly follow a 'scientific method' in their design and implementation of this system, but do so more from the perspective of a commercial enterprise with an end-goal (in this case a highly distributed fault tolerant storage system to serve as the backend of Facebook's Inbox Search which would allow for fast retrieval and fast write in heavy volumes) in mind. The paper itself does not particularly delve into an evaluation of the system but instead presents challenges the author faced during implementation, particularly focusing on the need to index 7TB of inbox data for 100M users in a MySQL infrastructure and load it into Cassandra which involved running MapReduce jobs against the data to index the MySQL data and then store the reverse index in Cassandra. They highlight that they had received requiests in applications for transactional operations for maintaining additional indices. They had experimented with Failure Detectors but found that the time grew unacceptably as the cluster size increased and number of nodes grew. They integretated their Cassandra system into a monitoring system to identify the systems behaviour in production. finally, they note that integrating the system with something such as Zookeeper (or some coordination system) is essential for the proper coordination and fault tolerance of tasks at scale. The only 'proper' level of evaluation which te authors provide is the performance of read operations on 50TB of data across a 150 node cluster as highlighting term search times between 7ms and 44ms, and although these seem impressive there is little comparison as to how other systems would perform.

The authors, ultimately, achieve their result by building a storage system which is highly scalable, performant, and applicable to a number of applications with high write throughput which is based on BASE properties. Cassandra is now available from Apache as an open-source NoSQL database and is used by companies including Apple, Netflix, and Instagram, and is one of the most popular NoSQL database implementations for distributed backend systems. It is, therefore, tough to deny that the authors achieved their result.

While I enjoyed that the paper was concise and focused on the key implementation details, I ultimately felt that the paper was too light on some key aspects to truly understand both the benefits and limitations of the system; in particular, I believe the paper is quiet light on some of the key limitations of Cassandra and outside of the overall description of the architecture and implementation does not make a particularly convincing case in the evaluation of Cassandra.

One of the key limitations which are present in Cassandra, which was actually the driver of why Facebook ultimately abandoned Cassandra in favour of HBase for its Inbox Search application as outlined in its Engienering Block (and subsequently from HBase to my Rocks as outlined in Facebook's Engineering Blog), is that the eventual consistency model ultimately proved a challenging design pattern to work with in contrast to HBase's ability to maintain consistency (this, ultimately, was mitigated to a degree in Cassandra through the usage of Merkel Trees in its conflict resolution process). While the article does successfully describe the specific use-case which was implemented for Facebook, it fails to necessarily showcase areas outside of this context where Cassandra may or may not be optimal, or how the performance compares to other distributed storage systems (such as BigTable and Dynamo). Cassandra has heavily penalties in the instance of a primary key not being known as querying data without this info requires a complete scanning of all nodes in the cluster. Similarly, a description of how the application fared in non-messaging based applications would have helped serve as a good direction for future research and providing more explicit information around precisely the use

cases which the authors believe the system would be well or poorly suited for implementation. Finally, by the time this paper was written HBase was released. It would have been sensible for the authors to provide at least some initial comparison between HBase and Cassandra. Given that we know now Cassandra was ultimately abandoned at Facebook in favour of HBase, and similar to the Cassandra implementation HBase also took many positive features from Dynamo and BigTable, I believe it is possible HBase was not highlighted as it was potentially known at this time that some aspects of Cassandra may have compared unfavourably.

The authors in the paper mention that they implemented Ganglia for the monitoring of the systems behaviour, and it may have been useful or beneficial to include some of the insight and metrics obtained from this system into the paper in order to provide insight in the bootsrap algorithm and understand whether there should be a systematic need to remove this from an administrative operation to an automated and algorithmic one. Similarly, the authors mention that Cassandra slots into a workflow featuring Ganglia (as mentioned) along with Zookeepr for coordination, but I beleive it would have been useful to also detail how well other aspects of the workflow can fit into a Cassandra-based backend architecture and any insights which were obtained.

The paper describes some of the core aspects of the system architecture (partitioning, scaling, failure management, etc) which are specific to Cassandra, however by the authors' own admission there are in fact many more characteristics. The authors claim that it is outside of the paper's scope to detail solutions to each component, but I would argue that it would actually have potentially been insightful to gain a more complete overview into how Cassandra helped support the implementation of the Facebook Inbox Search feature and any considerations which must be made in the implementation of Cassandra.

Ultimately my own conclusions on the paper are that while Cassandra is an interesting system and has become a widely used NoSQL solution among many Fortune 500 companies for its fault-tolerance and scalability, which is detailed relatively concisely (but hitting key components) in the authors' paper, the paper is inadequate in its evaluation and is not particularly self-critical - even though we know from context now that much of its performance metrics would compare positively to other NoSQL solutions, and that the described replication process and conflict management system combined with the focus on eventual consistency turned out to be a bigger issue than even hinted at in the paper, there really should have been at least some comparison with more substance contained within the paper itself outside of the single table on read operation performance on the Inbox dataset.

Bibliography

[1] Batra R. (2018) A History of SQL and Relational Databases. In: SQL Primer. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-3576-8_19

[2] Lake P., Crowther P. (2013) NoSQL Databases. In: Concise Guide to Databases. Undergraduate Topics in Computer Science. Springer, London. https://doi.org/10.1007/978-1-4471-5601-7_5

[3] Lourenço, J.R., Cabral, B., Carreiro, P. et al. Choosing the right NoSQL database for the job: a quality attribute evaluation. Journal of Big Data 2, 18 (2015). https://doi.org/10.1186/s40537-015-0025-0