Benchmarking Big Data Cloud-Based Infrastructures

A Major Qualifying Project report to be submitted to the faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the Degree of Bachelor of Science

Submitted By:

Kurt Bugbee
Jeffrey Chaves
Daniel Seaman

Advisor(s):

Mohamed Eltabakh

03/03/17

Abstract **2**

Introduction **3**

Hadoop 3

CouchDB 8

MongoDB 13

Apache Spark 20

Methodology **23**

YCSB 23

Dataset 25

Queries 27

CouchDB Implementation 29

MongoDB Implementation 31

Spark Implementation 33

Testing Environment 34

Metrics 35

Results & Analysis **36**

Installation 36

Overall Procedure 39

Syntax 40

Lines of Code 42

Latency 44

Conclusions **48**

Acknowledgements **50**

References **51**

Appendix **54**

Abstract

As software applications demand more complicated schemas from the storage platforms they utilize, many developers have migrated from the use of rigid relational columnar data sets to more flexible document-oriented datasets. Non-relational databases and processing platforms have evolved to tackle the task of operating on these new data structures. Three commonly used platforms were chosen for this project to be benchmarked against each other: CouchDB, MongoDB, and Apache Spark. Each platform was used to execute a series of queries involving both nested and unnested aggregation, projection, and filtering of a specific JSON dataset. The benchmarking was performed on Amazon Web Services EC2 instances, ensuring hardware resource consistency. Query latency, the total duration needed to query the data subset, was the quantitative performance metric used to analyze relative benchmarking. Each platform was also evaluated using a number of ease-of-use metrics. This report introduces the reader to each of the platforms mentioned and provides appropriate background information to help explain the purpose of this evaluation. The motives behind the queries and performance metrics are then explained to help provide a foundation for the project's methodology. Finally, the metrics are used to analyze testing results and draw conclusions from the performance of each platform.

Introduction

Hadoop

Named after one of the developer's toy elephants which now serves as its logo, Apache Hadoop is a massive open source project for distributed storage and processing. It's purpose is to allow for the processing of very large data sets of any type using low-end hardware.

The project began when the Google File System paper was published in October of 2003[1], leading to another significant research paper from Google in 2004 titled MapReduce: Simplified Data Processing on Large Clusters. What started as the Apache Nutch project was moved to the first official instance of the Hadoop project in January of 2006. The first official commit was made by Owen O'Malley in March of the same year [2]. Hadoop's first release, 0.1.0 was made available in April of 2006, and has since been used for a myriad of data processing functions, and is also the underlying data platform for a large number of other Apache projects. The official Apache website lists the official branches of the project as Hadoop Common, Hadoop Distributed File System (HDFS), Hadoop Yarn, and Hadoop MapReduce. The following are members of the Hadoop project community[3]:

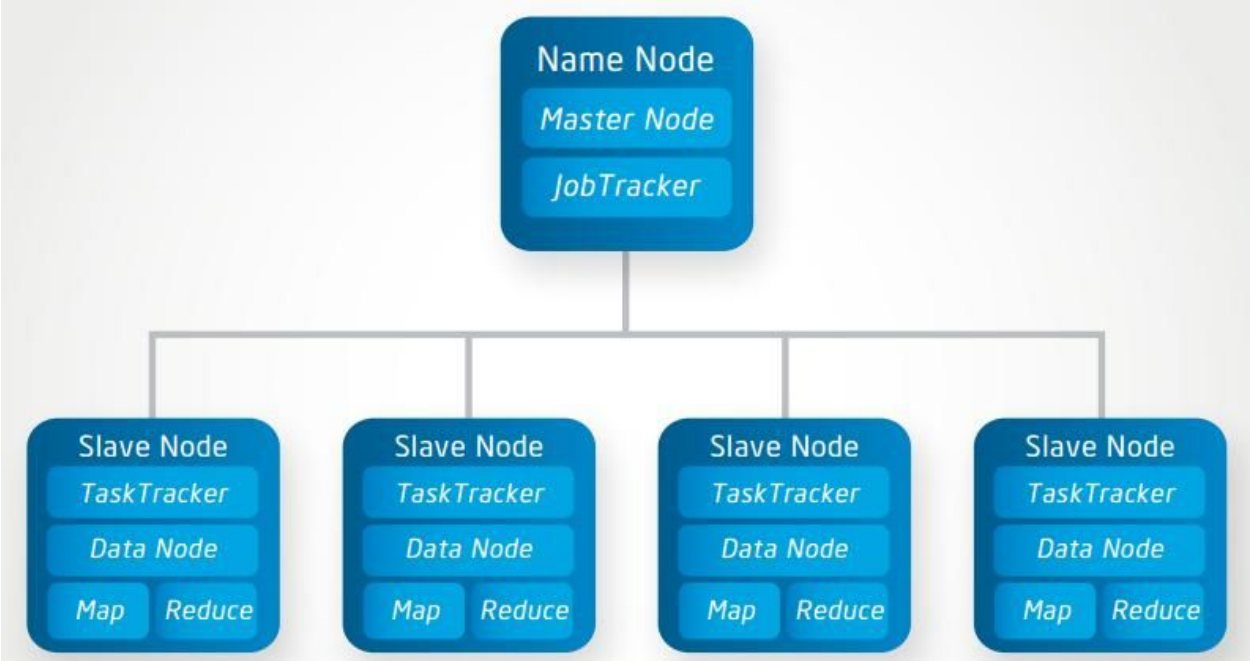
- Ambari™
- Avro™
- Cassandra™
- Chukwa™
- HBase™
- Hive™
- Mahout™
- Pig™
- Spark™
- Tez™
- Zookeeper™

Hadoop is currently used by hundreds of companies such as Facebook, Hulu, Google, Ebay, IBM, LinkedIn, the New York Times, and Spotify[4]. In 2011, developer Rob Bearden formed a partnership with Yahoo! and 24 engineers from the original Hadoop team to establish Hortonworks, the current leading resource on Apache Hadoop[2].

By design Hadoop is prepared to handle data of any size or type, having the motto that "Any Data Will Fit". Users need to define Input Formats in Java code for Hadoop to be able to manage data, but many common data types have pre-existing interfaces available for free online or natively through Hadoop. It is scalable from a single server up to thousands of machines and handles failures at the application layer, a unique feature that allows it to be run on low-end machines that are prone to failures. Hadoop has four primary pieces:

1. Hadoop Common: Common data utilities.
2. Hadoop Distributed File System: Commonly referred to as HDFS, the storage component of Hadoop.
3. Hadoop Yarn: The scheduling component of Hadoop, a framework for job scheduling and cluster management.
4. Hadoop MapReduce: The original Hadoop execution engine, the processing component of Hadoop.

This overview will focus on the two defining aspects of Hadoop, HDFS and MapReduce. Hadoop is designed with a master-slave architecture, with every node having a layer of HDFS storage and a processing layer of MapReduce. A master node defines these two layers as NameNode (HDFS) and JobTracker (MapReduce) while all slave nodes define their layers as DataNode and TaskTracker. See the following figure for an illustration[5]:



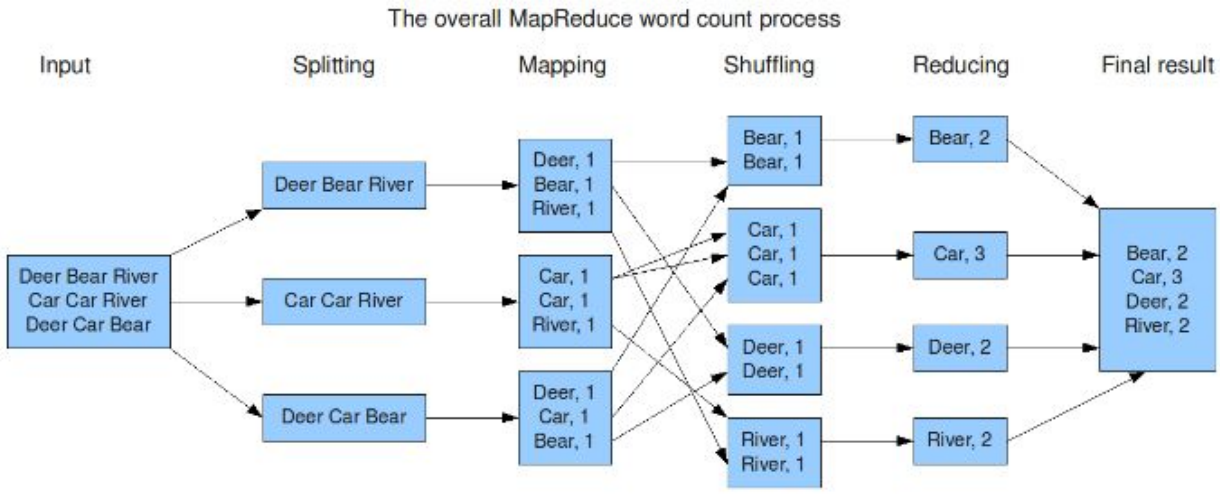
The central NameNode holds metadata on the files in the system, while the potentially infinite DataNodes store files that have been divided into blocks and then replicated across the clusters. Hadoop differs from traditional relational databases in that it is a black-box model. Rather than containing data tables and SQL queries that run quickly, Hadoop manages large amounts of data with data structures and longer-term jobs that are unseen by the user and provided in Java code.

HDFS, the storage backbone of Hadoop, can be distributed across hundreds or thousands of server machines. The current version of Hadoop still uses HDFS, though other Apache alternatives still exist such as Cassandra, HBase, or Accumulo. In order to handle server failures, files on HDFS systems are split into blocks and the blocks are replicated onto numerous instances. Hadoop defaults to having 3 replications of data blocks, but users can increase that number if they wish. This replication is crucial to Hadoop's fault tolerance, which will be covered shortly.

MapReduce, like HDFS, now has other Apache alternatives such as Spark, but continues to be a powerful execution engine that many other data processing platforms have implemented. MapReduce works in two parts: Mapper jobs are given a key and return a value. Mappers tend to hold the majority of the logic in a query of a Hadoop database. Reducer jobs then reduce the

values on the given keys, consolidating the values. Reducer jobs commonly happen across the cluster, but users are also able to specify local reducer jobs that occur on each individual machine to partially aggregate the mapper's output. These jobs are called Combiners, and in certain cases can improve processing time.

A popular MapReduce example is a word count: given a large passage of text, say 10-15 years of World Book Encyclopedia, HDFS would distribute the data into manageable sized blocks. Mapper jobs would split it into words, assigning them as keys and giving each a count value of "1" while stripping things like punctuation and ignoring case, assuming the user coded those functions. Hadoop then makes use of a shuffling/sorting algorithm that won't be covered here, but that would sort the key-value pairs according to the keys. Reducer jobs would then consolidate on the given keys: instances of the word "what" would be combined, and the end result would be key-value pairs in which the keys were unique words from the encyclopedias and the values were the total number of times each word occurred. The following graphic displays this nicely with a smaller example[6]:



MapReduce is highly scalable, and meant for massive data sets. More manageable data sizes with simple structures will almost always be faster to use with relational databases, but for the truly massive data sets, typical MapReduce computations can utilize parallel processing to

query many terabytes of data across thousands of nodes[7]. The current version of Hadoop is able to scale to petabytes of data.

Hadoop is optimized in a number of ways. The Combiners mentioned above are one way to optimize processing, as is concise code in the Mapper Class. Hadoop also makes use of speculative execution, meaning tasks are automatically run multiple times in parallel on different nodes. Since the NameNode is receiving constant feedback from the DataNodes, when one of the jobs finishes the others are immediately killed. Hadoop also focuses on locality, always trying to run mapper jobs on the MapReduce layer of TaskTrackers on the same machines as DataNodes that have relevant data in their HDFS layer. Both the localization and speculative execution efforts serve to reduce overall job time.

Finally, Hadoop has an incredible fault tolerance system. Failures are normal and in fact expected since Hadoop is optimized for running on mid-tier hardware. Since tasks are run in duplicate or triplicate and in parallel, if a task fails the responsible TaskTracker detects the failure and notifies the JobTracker. The JobTracker then reschedules the job to a new slave machine. Throughout jobs, the NameNode is constantly checking DataNodes. Therefore, if a DataNode fails, the NameNode and JobTracker recognize the failure almost immediately. Thanks to data block replication, the NameNode triggers a recovery from replicas as the JobTracker reschedules all tasks from the failed node, allowing processing jobs to continue virtually unhindered. The NameNode replicates the data to a new node to maintain the proper number of backups while the system continues running. Issues arise if the Master goes down, causing the NameNode and JobTracker to fail. This can only happen if the entire cluster has crashed, in which case the system has suffered a larger systemic issue than Hadoop can be responsible for.

Hadoop was originally one of the three platforms we were going to benchmark as part of this project, alongside MongoDB and Apache Spark. Our work gravitated towards using what could be defined as day-to-day realistically sized data. This decision is covered further in our data section, but amounts to the fact that while traditional MapReduce is incredibly powerful for massive data sets as an execution engine, it is outclassed when it comes to smaller sets by newer engines.

Newer platforms have made use of HDFS while finding more intuitive ways to write queries and handle input. Our eventual dataset consists of thousands of nested JSON records, and writing the Input Format with vanilla Hadoop was an arduous process, as was writing MapReduce queries in Java from scratch. The newest version of Hadoop includes Hadoop Yarn, which allows for newer data processing tools like Spark to be used; in other words, basic MapReduce is often phased out of Hadoop systems. After research of our problems repeatedly resulted in recommendations to use different wrappers for traditional Hadoop such as Spark (which we were already implementing), it was decided that continuing with Hadoop would not be a valuable contribution to the project. We chose instead to add CouchDB, which is covered in the next section.

CouchDB

An acronym for "Cluster of Unreliable Commodity Hardware"[8], the CouchDB project was created in April of 2005 by Damien Katz, a former IBM developer. Katz funded the project himself for almost two years before releasing it as open source, originally under the GNU General Public License. In February of 2008 it became an Apache Incubator project and was moved to the Apache License. It soon graduated to a top-level project and the first stable version was released in July of 2010. This release had phenomenal read and write times according to Couchio, CouchDB's corporate sponsor at the time[9]. It was also lauded for running not only on Linux and MacOS machines but Microsoft Windows as well.

Katz left the project in 2012 to work on CouchDB's successor, CouchBase Server, but the CouchDB effort continued, releasing v1.2 in 2012 and v1.3 a year later. In July of 2013, the CouchDB open source community merged the codebase for Cloudant's clustered version of the the database named BigCouch, giving CouchDB a native clustering framework[10]. The current version, 2.0.0, was released in September of 2016 with a number of GUI upgrades, a streamlined installation process, and the new Mango Query Server that allows for queries to be written without JavaScript or MapReduce, should a user wish to do so.

CouchDB's features made it a logical choice for us when we were looking to replace Apache Hadoop in our project. It holds many of the same attractions, but the similarities have received a facelift and the platform is optimized for the types of technology and data that are

now more common. It is a NoSQL database made for the modern developer who needs to work with multiple platforms such as PCs, mobile devices, and unreliable networks. It's also designed with simplicity in mind as shown by its one-word motto, appearing in the logs after installation: "Relax".

CouchDB is written in Erlang, an open source functional programming language that was released in 1998 after about 12 years of proprietary use. It is well-suited for systems with the following characteristics[11]:

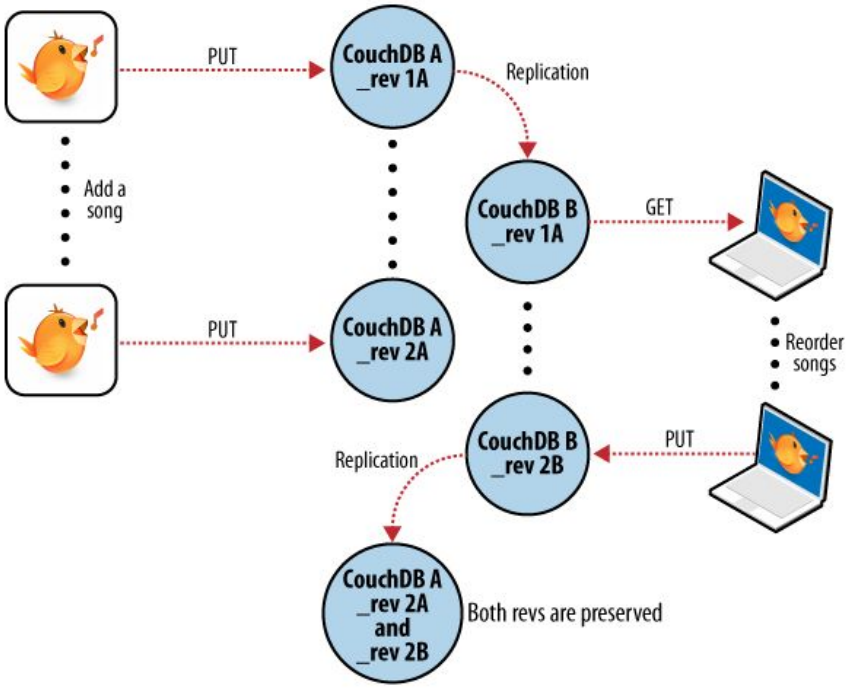
- Distributed
- Fault-tolerant
- Soft real-time
- Highly available, non-stop applications
- Hot swapping (changing code without stopping the system)

Common configurations for most users include launching CouchDB as a single-node database, but any single-node project can be upgraded to a cluster for more computing power. An important feature of its development is that although a cluster provides high capacity, there are no API changes whatsoever.

An incredibly impressive feature, and one that has some roots in Hadoop's data block structure, is CouchDB's bidirectional replication. It replicates incrementally, meaning that if something interrupts the replication, it will simply stop and wait to start up again where it left off. It also only replicates data that isn't already backed-up, decreasing replication time noticeably. Replication is either done continuously or ad-hoc at the behest of a user, and has full conflict detection at both end points, making master-master replication easy[12]. In other words, there is no authoritative node, and every node in a cluster can act independently of all the others if desired. This makes CouchDB very reliable simply due to redundancies, since any multi-node cluster replicates data automatically to all nodes. Used as the backbone of websites it can load balance by distributing jobs into subsets across the cluster, replicate data to distant locations to allow for lower latency access, and even work offline on mobile devices or devices with similarly poor network connectivity.

CouchDB holds its data in JavaScript Object Notation (JSON) documents. This is different from Hadoop, which was made for any type of data and allowed for unique input formats, but the reasoning behind it will become clear later in this section. In the meantime, it was convenient for us that CouchDB's integral data structure matched the format that our testing data was already in. Dealing with nested JSON documents was an issue that required some additional work with the other two infrastructures used in this project, but required no extra add-ons to the CouchDB database or edits to the data. It stores these documents in an append-only Binary-Tree structure. Leaf nodes of the tree are written via appends when revisions are made to documents, and the parent node is rewritten, also via append, to reference the new leaf. This process continues until the root node is updated in the same way, which is essentially when the new revision is fully committed.

This data structure can get confusing, and even more so when factoring in CouchDB's Multi-Version Concurrency Control (MVCC). A relational database, under heavy load from users, can run into locking issues and spend lots of processing power deciding who has permission to edit what, and when. With CouchDB, MVCC means no locking[13]. Writes don't block reads, so CouchDB can run at full speed at all times, even under heavy loads. To cope with this, the appends mentioned above allow the database to version its documents. A read request made for a document will be sent the document as it is. If a write is made while that document is being viewed, CouchDB will simply append the revision to the database without waiting for the read to finish. Any future read requests will get the new version of the document that was appended. This goes hand-in-hand with CouchDB's "Eventual-Consistency" feature. Because revised documents may not exist on other replications of the database that are actively used, they're flagged for replication at the next time it occurs. The whole process resembles a slow-moving but automatic variant of a version-control system like GitHub. This insures that during the next replication the outdated documents existing anywhere else will be replicated and all instances of the database with eventually be consistent. The following graphic shows how this process could work with a fictional music application when a playlist is edited[13]:



The way that users interact with CouchDB is unique among the three infrastructures used in this project because it fully embraces the environment a modern server is likely to find itself in. To go along with data stored in JSON documents, MapReduce queries are written in JavaScript. In addition, HTTP serves as the RESTful API interacting with the database. This made uploading our data to CouchDB possible through a simple cURL command:

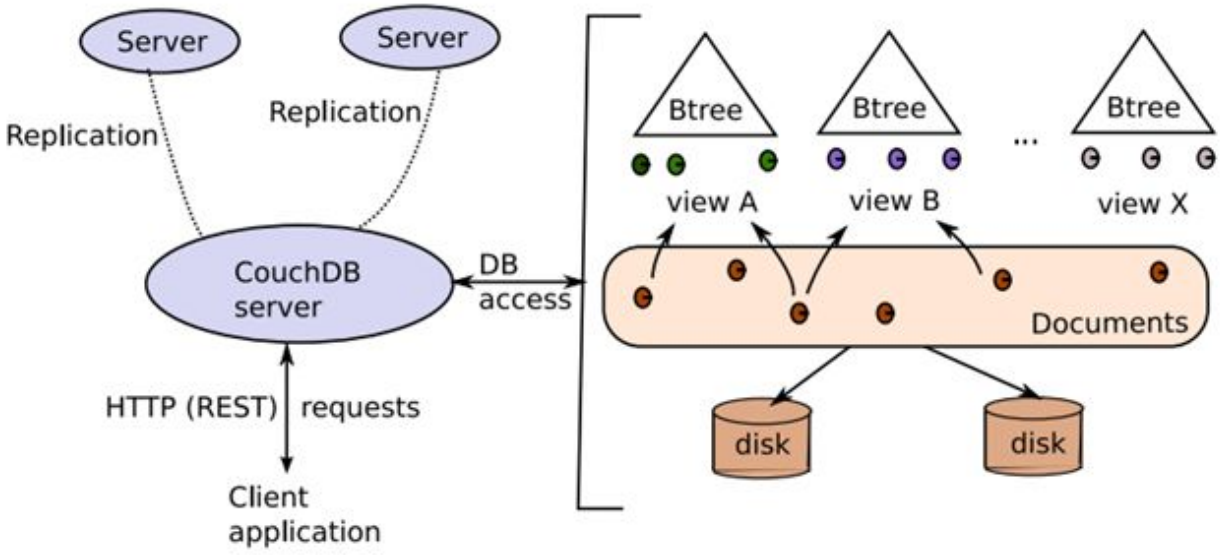
```
curl -H "Content-Type:application/json" -d @myFile.json -v POST "http://SERVER_PATH/DB_NAME/_bulk_docs"
```

The MapReduce logic is significantly simplified when compared to Hadoop's Java code. Users only need to write the Mapper logic and can either choose from a small selection of built-in Reducers or specify a custom one. The MapReduce code is used to create new Views of the document that can be accessed by HTTP requests to the server. We implemented these in the form of additional cURL commands, which is further mentioned in the Queries section of the report.

Through cURL commands (or similar utilities) CouchDB can be fully accessed using the REST API. This makes it incredibly easy to add data, create views, query the views, and check

on the overall health of the server. Replication is also controlled through the same REST API, and as mentioned above, the API doesn't change when moving from a single-instance setup to a cluster of computing nodes. CouchDB also shipped with a Web GUI called Futon in all primary versions, updated to Fauxton with version 2.0.0. Fauxton allows for visual traversal of documents in the overall database and in any Views. Users can also create new views and write new query logic in Fauxton, and live-track CouchDB's processing as it gathers the appropriate data. Fauxton also provides some ease-of-life administrative tools like database and user creation and management. Perhaps the best thing about Fauxton, for those who prefer to avoid command line interfaces, is that it is entirely a web application. This means it is exactly the same whether you're working on a Windows, MacOS, or Linux machine. Also, when writing HTTP requests that must be executed outside of the application, the relevant RESTful API call can be seen right in the browser's address field.

With all of CouchDB's significant features explained at a high level it is useful to see a simple graphic of CouchDB's architecture for visual explanation[14]:



CouchDB has definitive weaknesses in that every single query that a user wants to write requires a hard-coded view to go with it. It's speed in data retrieval can be partially attributed to the fact that it has these views saved, so pulling data from them is faster than querying the entire database of original documents. Instead, a new set of documents is created with only the desired

information that can be returned on request. This makes CouchDB a very good option for an application that accumulates data which isn't altered too often, where most of the queries that will be run can be pre-defined, and where versioning is important[14]. It's much, much slower out of the box (read: impromptu queries) than infrastructures like MongoDB might be, and limited by design to single-server capacity models due to its append and replicate structure, but for the same reasons it's also safer in terms of data security[15].

Unlike Hadoop MapReduce, CouchDB was on the same modern playing field as MongoDB and Apache Spark. It's optimized for use-cases that aren't applicable to every implementation of a database, but no data platform is or can be. Given CouchDB's promises of speed and ease of use, and it's convenient compatibility with the data type our project uses, it seemed an appropriate choice for our benchmarking comparison.

MongoDB

MongoDB is a flexible NoSQL database tool which stores data in binary JSON (BSON) format, allowing documents to contain many fields composed of different data types such as arrays, sub-documents, and binary data. MongoDB has drivers which make it easy to use in the popular language of your choice and is designed to be easy to deploy, provision, and scale. It is easily scaled horizontally through MongoDB auto-sharding and automatic leader election, supporting high availability across racks and data centers. MongoDB's speed comes from its extensive use of RAM and, unlike most NoSQL databases, it provides comprehensive secondary indexes. MongoDB also has multimodal capabilities including in-database analytics, graph, cross-document relations, search, faceted navigation, and more. MongoDB's design is focused around combining the critical capabilities of relational databases with the innovations of NoSQL technology to address the requirements of modern applications[16].

The relational database features offered by MongoDB include expressive query language and secondary indexes, strong consistency, and Enterprise management and integrations. The expressive query language and secondary indexes allow users to easily and efficiently access and manipulate their data, supported natively by the database rather than being maintained in application code. Available secondary indexes include geospatial and text search as well as extensive security and aggregation capabilities. MongoDB's strong consistency allows

applications to immediately read what has been written to the database. This makes it much easier on the developer as it is very challenging to build applications around an eventually consistent model. MongoDB also lends itself to Enterprise integration as it can be secured, monitored, automated, and integrated with existing technology infrastructure, processes, and staff.

Relational databases, however, do not address many of the requirements that are imposed by modern applications. NoSQL databases address many of these requirements through their flexible data model, scalability, performance, and always-on global deployments. NoSQL databases emerged to adapt to the evolving requirements of modern applications. By offering a flexible data model, MongoDB makes it easy to store and combine data of any structure including documents, graphs, key-value pairs, and more. This flexibility also makes it easy to dynamically modify the schema without downtime or performance impact.

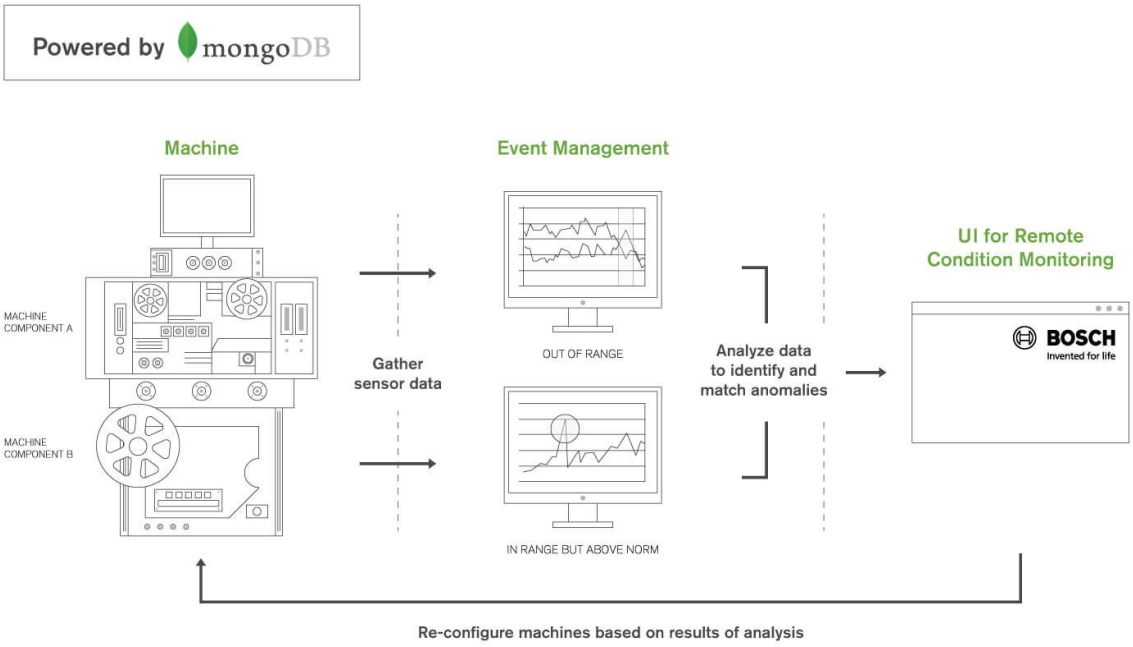
Another focus of NoSQL databases is scalability in the form of sharding or or partitioning. This allows the database to scale out on commodity hardware, or in the cloud, enabling almost unlimited growth with higher throughput and lower latency than relational databases. The always-on global deployment of NoSQL databases allows for highly available data, distributed across many nodes with automatic replication across servers, racks, and data centers.

This scalability is not just about speed, it is measured using three different metrics: cluster scale, performance scale, and data scale. MongoDB's cluster scalability is used by EA Sports FIFA and Russia's largest search engine, Yandex, which both utilize the high-performance auto-sharding to support gameplay servers and the billions of objects and terabytes of data which grow at 10 million file uploads a day. The performance scale of MongoDB lends itself well to applications like Foursquare, which boasts a user base of over 50 million people, and Comcast, which supports 100,000 operations per second with 99.999% availability. MongoDB has also provided high data availability to companies such as McAfee Global Threat Intelligence (GTI), a cloud based intelligence service which correlates data from millions of sensors around the globe, and Craigslist, which hosts 80 million classified ads per month[17].

Most NoSQL databases sacrifice the key properties of relational database transactions, described by the ACID acronym (atomicity, consistency, isolation and durability). However, with its Nexus architecture, MongoDB is the only NoSQL database that harnesses the capabilities of NoSQL without sacrificing the foundation of relational databases[18].

MongoDB can be used to easily create single-view applications, making it possible to aggregate data from multiple sources into a central repository. This is extremely useful to companies dealing with financial services, government, high tech ventures, and retail providers which benefit from having a single view for their respective asset classes, military assets, cross-product use, and customer behavior. This type of application lends itself better to MongoDB than a relational database because of its ability to consolidate data in different formats from different systems. MongoDB's dynamic schemas and expressive query language also saves time and money by making it easy to adjust the schema and access the data in whatever format is needed[19].

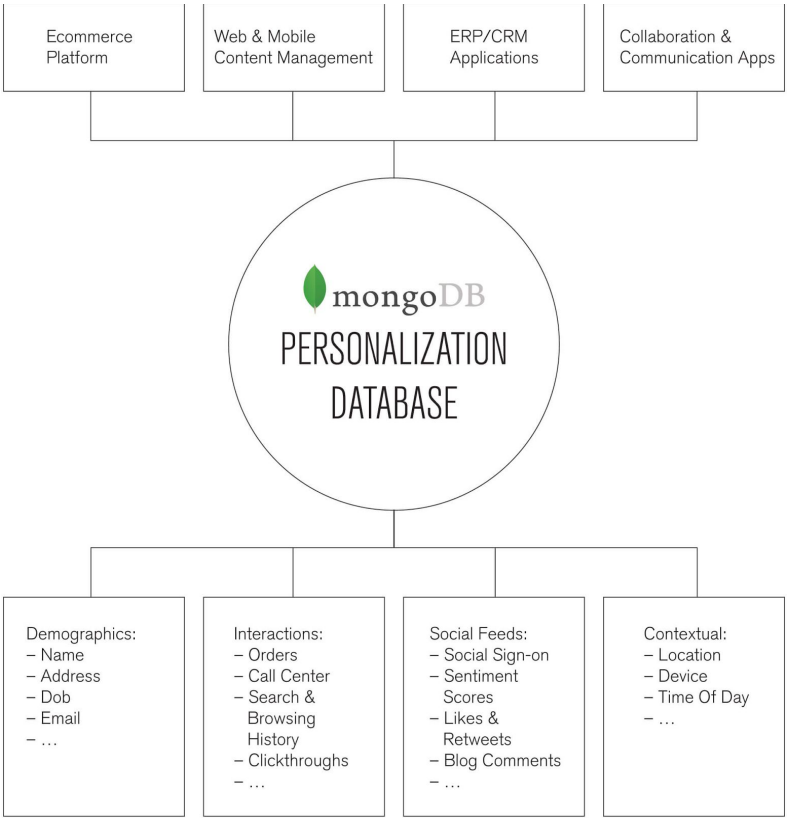
MongoDB is also very useful for Internet of Things (IoT) applications. The internet of things describes the interconnectivity of devices, mainly allowing them to share data and make life easier in some way. Financial services use it to monitor vehicle performance and driver behavior, government organizations use biometric sensor data from patients to alert doctors early, high-tech companies provide wearable tech to analyze diet, exercise and sleep, and retailers can present enticing offers to shoppers using in-store beacons and purchase history data as they walk through the store. As new types of sensors emerge, they provide new forms of data, which can easily be handled by the flexible schema of MongoDB. These billions of sensors provide volumes of data that relational databases cannot handle, yet MongoDB is designed to scale out and handle these extreme amounts.



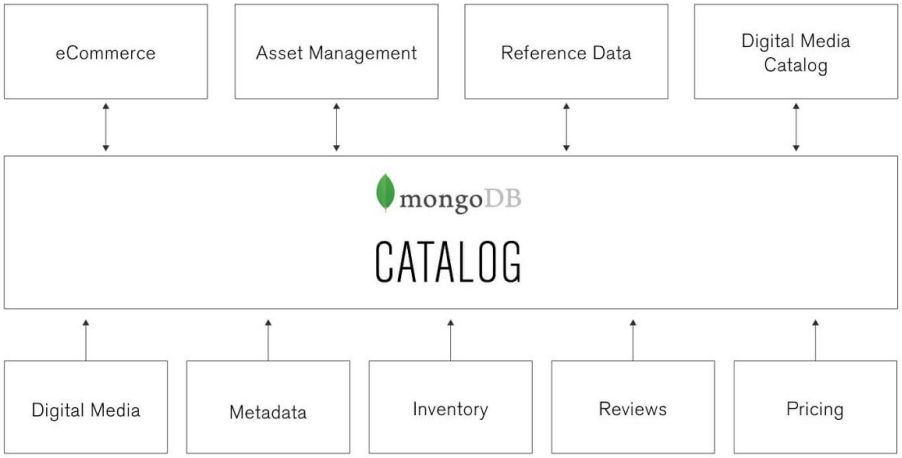
MongoDB makes it easy to make significant progress on each iteration of your application development, making it easier to roll out consistent mobile application updates. Examples include payroll service ADP, which hosts over 1 million end-users, and the Weather Channel, which uses MongoDB to handle 2 million requests per minute and provide real-time weather alerts for 40 million users. Financial services use smartphone apps for users to submit insurance claims with geo tags and pictures taken on their phones. Government organizations use mobile apps for healthcare appointment scheduling, check-ins and prescription refills. MongoDB also allows retailers to produce apps with a product catalog, barcode scanner, advertisements for local deals, loyalty programs, and a store locator. The datasets evolving in mobile applications benefit from MongoDB's flexible schema and their large user bases benefit from the scalability of the database.

MongoDB is also used for real-time analytics which require low latency (sub-second) and high availability (e.g., 99.99%). Examples include financial services' analysis of satellite imagery and weather trends, government identification of social program fraud, identifying unique individuals across different devices, and retailers' generation of in-store incentives to shoppers in real time.

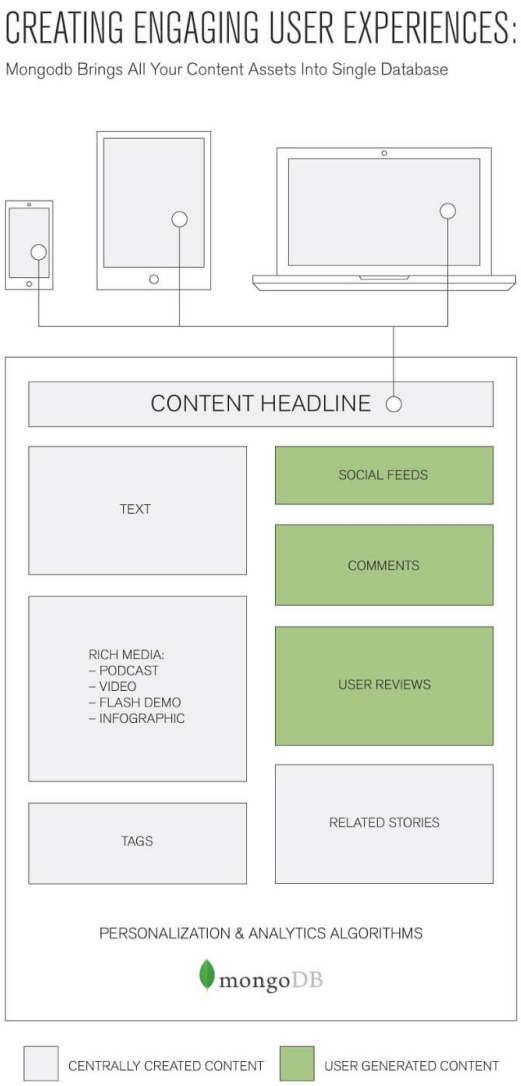
MongoDB makes it very easy to personalize the experiences of your customers based on previous online interactions they have made with your business. Knowing a customer's likes, dislikes, and previous history allows you to predict their wants and needs. MongoDB can be used on top of legacy systems, or replace them altogether. Financial services can relevant lending offers based on transaction history and credit score, government organizations can create portals for users to interact with based on location and status, high tech companies can stay compliant by identifying customer location and keeping data within geographic borders, and retailers can recognize returning digital customers and match their preferences and history to products that can be provided. The flexible schema of MongoDB allows for personal tracking in any data form needed while providing real-time personalization capability because of its real-time analytics capabilities.



While using a relational database it is very hard to handle catalog changes since you cannot easily add new items, attributes, or features without impacting database performance or taking the database offline. This is why Otto, Europe's second-largest e-commerce company, uses MongoDB to continually update its catalog of over two million products which attracts its thirty million shoppers and drives in its yearly 2.3 billion Euro revenue. Financial services use MongoDB as a central repository of trades across multiple asset classes for aggregated analysis, government organizations are able to keep a single data store for thousands of assets under an organization's purview, high tech companies are able to match customers with the right products at the right time by keeping metadata on user activity, and retailers' omni-channel product catalog and inventory management allows the ability to make informed recommendations.



MongoDB also makes it easier to manage, store, and serve any type of content, new or old, from a single database. This ability to build any feature and incorporate any data faster and for less money has allowed companies like Forbes, who created a custom content management system (CMS) in two months and a new mobile site in one month, to greatly benefit from MongoDB's content management ability and flexibility. Financial services are able to replace expensive software like Sharepoint to aggregate, store, and serve equity research, government organizations are able to publish government archives online, high tech companies are able to consolidate services and app backends into a single database for clean integration and simple operations, and retailers are able to get visitors to click, interact, and shop online by pairing product listings with YouTube videos, live demos, and Twitter feeds to get customers closer to the product.



The decision to use MongoDB for this project stemmed from the widespread use of the infrastructure in the software community. As a well established platform that continues to be implemented in many different types of applications we felt it was important to compare it to some of the newer infrastructures available today.

Apache Spark

Apache Spark is an open-source cluster-computing framework. The Spark codebase was donated to the Apache Software Foundation after being developed by the University of California, Berkeley's AMPLab. Apache Spark provides programmers with an application programming interface build around a resilient distributed dataset, a data structure consisting of a read-only multiset of data items distributed over a cluster of machines, that is maintained in a

fault-tolerant way. [20] This platform was developed in response to limitations in the MapReduce cluster computing paradigm, which forces a particular linear dataflow structure on distributed programs: MapReduce programs read input data from disk, map a function across the data, reduce the results of the map, and store reduction results on disk. Spark's RDDs function as a working set for distributed programs that offers a (deliberately) restricted form of distributed shared memory. [21]

Apache Spark contains a library for to compute SQL queries on datasets. Spark SQL was released in May 2014, and is now one of the most actively developed components in Spark. As of this writing, Apache Spark is the most active open source project for big data processing, with over 400 contributors in the past year. Spark SQL provides a DataFrame API that can perform relational operations on both external data sources and Spark's built-in distributed collections. This API is similar to the widely used data frame concept in R, but evaluates operations lazily so that it can perform relational optimizations. In addition, to support the wide range of data sources and algorithms in big data, Spark SQL introduces a novel extensible optimizer called Catalyst. Catalyst makes it easy to add data sources, optimization rules, and data types for domains such as machine learning. Spark SQL has already been deployed in very large scale environments. For example, a large Internet company uses Spark SQL to build data pipelines and run queries on an 8000-node cluster with over 100 PB of data. Each individual query regularly operates on tens of terabytes. In addition, many users adopt Spark SQL not just for SQL queries, but in programs that combine it with procedural processing. For example, 2/3 of customers of Databricks Cloud, a hosted service running Spark, use Spark SQL within other programming languages. Performance-wise, we find that Spark SQL is competitive with SQL-only systems on Hadoop for relational queries. It is also up to 10x faster and more memory-efficient than naive Spark code in computations expressible in SQL. [22]

The DataFrame API mentioned above offers rich relational/procedural integration within Spark programs. DataFrames are collections of structured records that can be manipulated using Spark's procedural API, or using new relational APIs that allow richer optimizations. They can be created directly from Spark's built-in distributed collections of Java/Python objects, enabling relational processing in existing Spark programs. Other Spark components, such as the machine

learning library, take and produce DataFrames as well. DataFrames are more convenient and more efficient than Spark's procedural API in many common situations. For example, they make it easy to compute multiple aggregates in one pass using an SQL statement, something that is difficult to express in traditional functional APIs. They also automatically store data in a columnar format that is significantly more compact than Java/Python objects. Finally, unlike existing data frame APIs in R and Python, DataFrame operations in Spark SQL go through a relational optimizer, Catalyst.

Regarding the development for Apache Spark, Spark is powerful and expressive in terms of how a user orders operations to be executed. The ability to use additional functions like Filter, Join, and Group-By enables an intuitive development process. Apache Spark promotes focus towards higher abstraction instructions, and away from lower level MapReduce implementation. For example, a preconfigured program that is included with the Apache Spark install is a sample wordcount example. This program is roughly 100 lines of code when done with MapReduce, but the equivalent in Apache Spark is executed in 4 lines of code. This intuitive design allows for more efficient execution and query design.

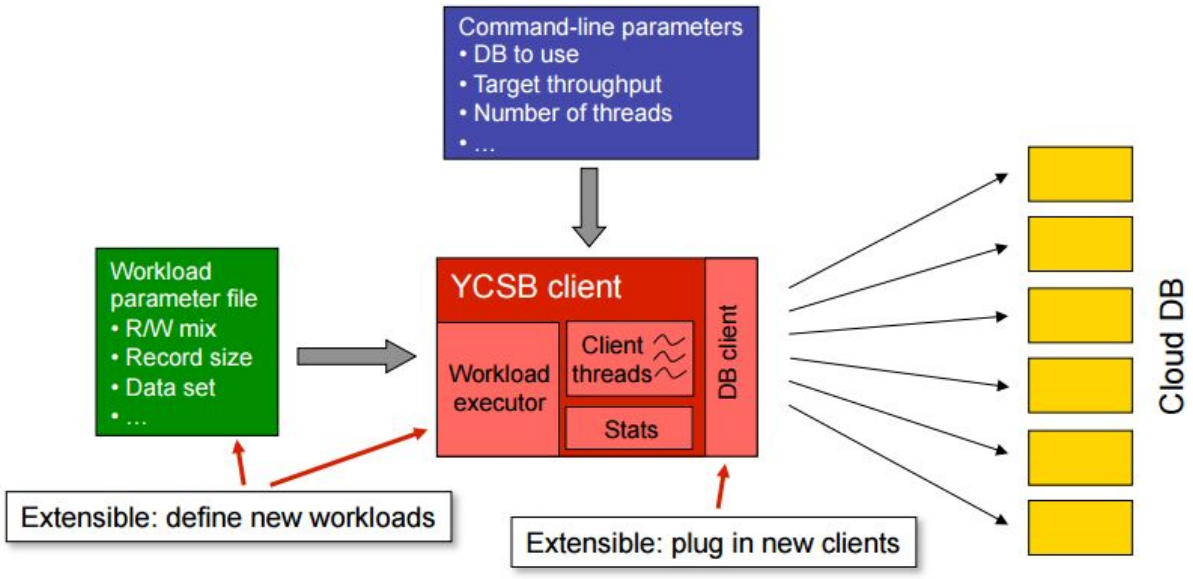
Apache Spark swears by their speed, ease of use, generality, and ability to run everywhere. Statistics on spark.apache.org claim that Apache Spark runs programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk. Mentioned in the previous section, Apache Spark offers efficient program creation and execution, requiring fewer lines of code to execute similar programs. It's apparent Apache Spark is a modern staple in cluster computing, and continues to dominate data processing.

Methodology

YCSB

The Yahoo! Cloud Serving Benchmark (YCSB) was released in 2010 by the research division of Yahoo!. It was developed with the goal of "facilitating performance comparisons of the new generation of cloud data serving systems"[23]. YCSB competes with the TPC-H benchmark created by the Transaction Processing Performance Council, though YCSB is more focused on benchmarking big data platforms. YCSB is now a standing open-source project.

YCSB provides users with two parts: the YCSB client itself, which is an extensible workload generator, and a set of core workloads that can be executed immediately[24]. The core workloads are designed to provide a well-rounded summary of a platform's performance metrics and users are able to define their own workloads to cover additional system aspects. By default YCSB workloads will provide latency and throughput statistics as they finish running. The following graphic displays the YCSB system architecture.



Since its deployment, YCSB has been used to benchmark a number of products for both marketing and scholarly comparisons. To date, YCSB has built-in interfaces for the following data platforms[25]:

- Apache Accumulo
- Apache Cassandra
- Apache Geode
- Apache HBase
- Apache Kudu
- Apache Solr
- Aerospike
- Amazon S3
- Async HBase
- ArangoDB
- Couchbase
- DynamoDB
- ElasticSearch
- Google Bigtable
- Google Data Storage
- Hypertable
- Infinispan
- JDBC
- Mapkeeper
- Memcached
- MongoDB
- NoSQL DB
- OrientDB
- Redis
- Riak
- Tarantool
- Voldemort

Because this project originally required benchmarking Apache Hadoop (later replaced by CouchDB), Apache Spark, and MongoDB, we were able to utilize the built-in interface for

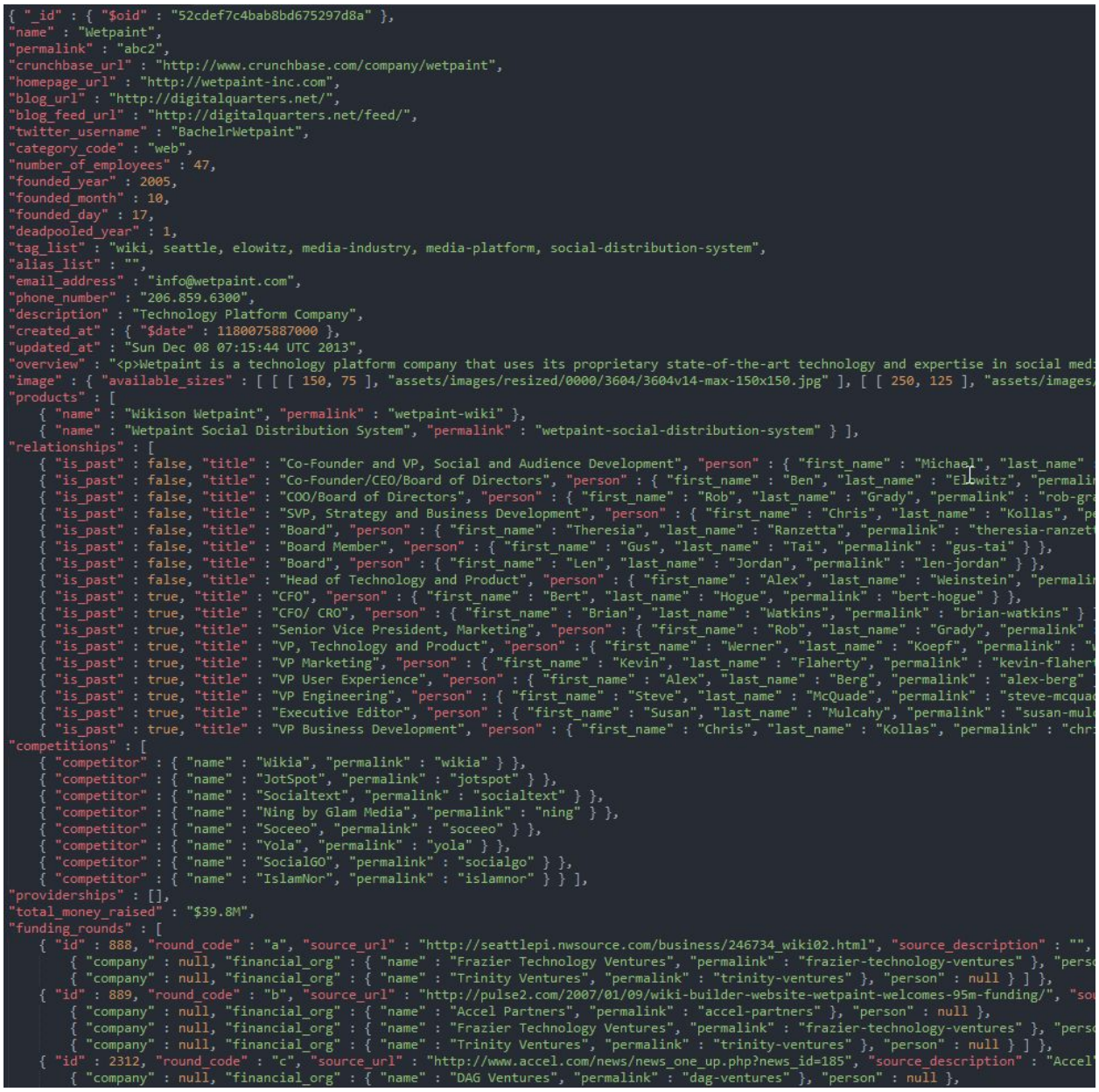
MongoDB only. Our original goal with the project was to implement our own interfaces for Hadoop and Spark. This would have allowed us to complete the comparison and possibly contribute to the open source project as well.

A shortcoming of YCSB was discovered while we were using the MongoDB interface: the format of the data produced for and queried from MongoDB. One of the conveniences of the JSON document format, which is what we eventually decided to use, is the ability to store documents within documents as data is stored in the format of key-value pairs, with the title of the sub-document as the key and the embedded document as the value. One of the requirements of the project was to evaluate the performance of queries that handle data within these embedded documents, yet YCSB failed to do so. Within the dataset produced by YCSB the documents are composed of a single field containing a string of randomly generated characters; there are no embedded documents. This was the first factor in our eventual decision to abandon using YCSB as our benchmarking tool and develop our own custom benchmarking process.

After a period of time trying to implement YCSB with Hadoop and Spark, we concretely concluded that it was not going to be an effective way to complete our project, despite YCSB being the current standard for measuring NoSQL database metrics. Our first issue with this task arose when trying to create an interface for Hadoop. It's a well-documented shortcoming of YCSB that it does not support MapReduce functions [26]. Although it's able to effectively benchmark other processing engines built on top of Hadoop HDFS like HBase, our project originally specified using vanilla Hadoop, which made using YCSB impossible within our time constraints. The eventual move to CouchDB, another MapReduce database, and the fact that there was also no pre-existing interface for Spark solidified our decision to benchmark the platforms in a different way.

Dataset

Moving away from YCSB meant benchmarking the databases with our own metrics, and in order to do that we needed data to query. We decided on a 76MB data set that contained 18001 records on companies in JSON format. Each company record contains 38 top-level attributes, 17 of which can contain additional nested documents. Here is a screenshot of a single record, split into new lines for readability. The entire record is not visible:



We chose this data set for a number of reasons. First, JSON is a very popular data format that any modern database, as of the writing of this report, should be able to support. Second, we thought that a size of around 100MB seemed appropriate for initial testing, and as we implemented AWS (see the subsection on our Testing Environment) it turned out to be a good size for final testing as well. Finally, a large part of our assessment is whether or not the data

platforms could handle embedded documents, and how well. JSON is well-equipped to create embedded documents, and this particular data set had many to choose from, some nested to 3 or 4 levels.

Queries

In order to obtain a more complete evaluation of our big data tools we benchmarked them using different query types. These queries included projection, filtering, and single purpose aggregations which used sum, group, and distinct operations. The dataset we used was in JSON format and was composed of a collection of data from worldwide startup companies. Since we were using a JSON dataset we also felt it was necessary to implement queries that retrieve embedded documents. To do this we implemented all of the same query types previously described, but to deeper levels of the document hierarchy.

Aggregation is, by dictionary definition, a collection or gathering of things together. In the case of our big data tools, aggregation is implemented by executing a collection of query operations, combined to retrieve and format data into the form that is needed by your application. We used three types of single-purpose aggregation queries which we felt were basic and commonly needed. We summed the **num_employees**[1] field to return the total number of employees across all companies in the dataset. We grouped companies by the year they were founded in (**founded_year**), returning each existing year in the dataset and the total number of companies founded in the corresponding year. We also used the distinct operation to return each distinct **category_code**.

Projection can be explained through this example: Let ρ be a relation and let A, B, ..., C be attributes of ρ. Then the projection of ρ on (or over) those attributes, ρ{A,B,...,C}, is a relation with:

a.) heading: {A,B,...,C} and

b.) body: the set of all tuples *x*

[1] All mentions of actual fields from our documents will be bolded throughout this section.

such that there exists some tuple θ in ρ with A value equal to the A value in x, B value equal to the B value in x, ..., and C value equal to the C value in x. In terms of this project, our projection queries returned a list for each company containing their four URLs (**crunchbase_url**, **homepage_url**, **blog_url** and **blog_feed_url**).

Filtering involves scanning all documents and returning a smaller percentage of them based on a qualification metric. Filtering queries can be very useful for measuring query processing time, since it's relatively easy to tailor them to return certain percentages of the data. In our case, we queried the documents for a **founded_year** of 2010 or later. This could easily be changed to any other year by simply changing a couple characters in the code; querying for 2000 or later would return a larger percentage of the data, while querying for 2016 or later would return a smaller percentage.

We have so far covered the following 5 query decisions:

- Aggregation
    - Summing number of employees across all companies
    - Grouping companies by their founded year
    - Returning distinct category codes
- Projection
    - Returning the four URLs for each company
- Filtering
    - Returning companies founded on or after a given year

The next step was to implement similar queries, but to access embedded (or nested) documents in our data. In order to implement aggregation of embedded documents we used the same types of single purpose operations, just at a deeper level in the document hierarchy. We returned each company **name**, accompanied by the sum indicating the total amount of funding that the company has raised, summing the **raised_amount** nested within each of the **funding_rounds**. We grouped the documents by the cities they have **offices** in, accompanied by the count of total offices in each corresponding **city**. We also implemented the distinct operation by returning each distinct **price_currency_code** from the **acquisition** document. By implementing these three single-purpose aggregations to various levels in the JSON document

hierarchy we ensured that our tool has a complete evaluation of simple aggregation performance in each of our big data tools.

For projection we again returned the **name** of each company, but this time projected an attribute from within the embedded **image** document. The image document includes an **available_sizes** attribute containing a list of dimensions; we returned the smallest available image size. Projection is also used in the embedded document filter query. We project all company records with a high enough **price_amount** from the nested **acquisition** document. Specifying an acquisition price of $2,000,000,000 for example returned only a small handful of the companies, while lower prices returned much more.

In summary, our embedded documents queries were:

- Aggregation
    - Summing the funding amount raised by each company from all funding rounds
    - Counting the number of offices in each distinct city across all companies
    - Returning each distinct currency code used for company acquisitions
- Projection
    - Returning the smallest available company image size
- Filtering
    - Returning only the companies acquired at or above a specific price point

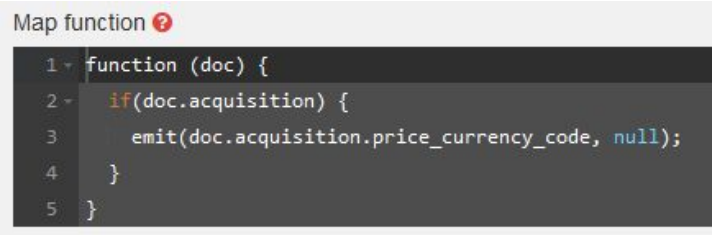
We did not include "write" actions in our queries; while MongoDB and CouchDB both support writes, Apache Spark does not and therefore it did not make sense to implement them on only two of our databases. See the conclusions section for more discussion on this topic

.

CouchDB Implementation

CouchDB utilizes MapReduce in JavaScript. Because of JavaScript's relatively lax coding conventions, the code snippets for Mapper functions are very short, and the Reduce functions don't even have to be user-written; CouchDB has built-in reducers that can be used. We did decide to write our own Reducers for the sake of the project, so an example of our CouchDB grouping by founded year query looks like this:

Accessing embedded documents is very, very simple in CouchDB, since JSON is the built-in data structure. Our distinct currency code query Mapper is just:



These queries, shown in CouchDB's Fauxton interface, create Views which are addressed in the CouchDB section. In order to access the actual output of the queries, we used simple cURL commands. The cURL command for the year founded query is:

```
curl http://localhost:5984/mqp/_design/year-founded/_view/year-founded?group=true&group_level=1 > output_query2.txt
```

Which provides output that looks something like this:

```
{"key":1980,"value":28},
{"key":1981,"value":25},
{"key":1982,"value":29},
{"key":1983,"value":36},
{"key":1984,"value":27},
{"key":1985,"value":34},
{"key":1986,"value":45},
{"key":1987,"value":42},
{"key":1988,"value":33},
{"key":1989,"value":46},
{"key":1990,"value":41},
{"key":1991,"value":46},
{"key":1992,"value":58},
{"key":1993,"value":81},
{"key":1994,"value":97},
{"key":1995,"value":145},
{"key":1996,"value":216},
{"key":1997,"value":200},
{"key":1998,"value":290},
{"key":1999,"value":533},
{"key":2000,"value":521},
{"key":2001,"value":464},
{"key":2002,"value":460},
{"key":2003,"value":576},
{"key":2004,"value":752},
{"key":2005,"value":961},
{"key":2006,"value":1423},
```


MongoDB Implementation

Queries can be implemented in MongoDB either by using their NoSQL commands from the Mongo shell or by using one of the many languages supported by MongoDB drivers. In the case of our project we first constructed the queries in NoSQL, but completed all testing using the MongoDB Java drivers for ease of collecting performance statistics. An example of our MongoDB NoSQL grouping by founded year query looks like this:

```
db.mainData.aggregate([{ "$group" : { _id: "$founded_year", count: {$sum:1}}}])
```

Similarly, an example of our MongoDB Java grouping by founded year query, including variables for latency calculation, looks like this:

```
// Form query with Document objects
Document count = new Document("$sum", 1);
Document id = new Document("_id", "$founded_year");
Document objectTwo = new Document("$group", id.append("count", count));
// Run & Time Query
startTime = System.nanoTime();
AggregateIterable<Document> outputTwo =
                this.coll.aggregate(Arrays.asList(objectTwo));
endTime = System.nanoTime();
duration = (endTime - startTime) / 1000000;
```

As can be seen in the two previous figures, there is a great similarity between the NoSQL and Java Driver implementations of MongoDB queries. Within the NoSQL command there are key-value pairs that are referenced in order to query the data from the JSON dataset. When translated to Java Driver syntax, the key-value pairs must be constructed using the *Document* object (from the org.bson.Document package). Once you have formed the components needed to implement the query, you pass them as an *ArrayList* to the *aggregate()* function which utilizes the MongoDB aggregation pipeline.

Not all queries must utilize the aggregation pipeline however, as there are much simpler commands that can be used to perform single purpose aggregation. For example, if you wanted to return each distinct value existing in a certain field (in this case the price currency code of the company acquisition) it would be as simple as this in MongoDB NoSQL and Java Drivers, respectively:

```
db.mainData.distinct('acquisition.price_currency_code')
```

```
ArrayList<String> distinctPriceCurrCode =
                this.coll.distinct("acquisition.price_currency_code", String.class)
                        .into(new ArrayList<String>());
```

We found that using queries with MongoDB had very simple syntax and once one learns how to implement them in NoSQL they are easily converted into Java Driver syntax.

Spark Implementation

Spark SQL is Apache Spark's module for working with structured data. This module allows for a seamless integration of SQL queries with Spark programs. Each query is a mixture of Scala and SQL. Here is an example of what our Apache Spark grouping by founded year query looks like:

```
val q2 = spark.sql("SELECT COUNT(founded_year), founded_year FROM people GROUP BY founded_year")
q2.show()
```

Querying nested documents is also supported. Our distinct currency code query is shown as:

```
val q1 = df.select(explode(df("acquisitions")))
val q2 = q1.select("col.price_currency_code")
test.createOrReplaceTempView("tempT")
val q3 = spark.sql("SELECT DISTINCT price_currency_code FROM tempT")
q3.show
```

Nested queries are extremely easy in Apache Spark due to the ability to print the database schema straight to the shell terminal:

```
scala> df.printSchema
root
 |-- _id: string (nullable = true)
 |-- acquisition: struct (nullable = true)
 |    |-- acquired_day: long (nullable = true)
 |    |-- acquired_month: long (nullable = true)
 |    |-- acquired_year: long (nullable = true)
 |    |-- acquiring_company: struct (nullable = true)
 |    |    |-- name: string (nullable = true)
 |    |    |-- permalink: string (nullable = true)
 |    |-- price_amount: long (nullable = true)
 |    |-- price_currency_code: string (nullable = true)
 |    |-- source_description: string (nullable = true)
 |    |-- source_url: string (nullable = true)
 |    |-- term_code: string (nullable = true)
 |-- acquisitions: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- acquired_day: long (nullable = true)
 |    |    |-- acquired_month: long (nullable = true)
 |    |    |-- acquired_year: long (nullable = true)
 |    |    |-- company: struct (nullable = true)
 |    |    |    |-- name: string (nullable = true)
 |    |    |    |-- permalink: string (nullable = true)
 |    |    |-- price_amount: double (nullable = true)
 |    |    |-- price_currency_code: string (nullable = true)
 |    |    |-- source_description: string (nullable = true)
 |    |    |-- source_url: string (nullable = true)
 |    |    |-- term_code: string (nullable = true)
 |-- alias_list: string (nullable = true)
 |-- blog_feed_url: string (nullable = true)
 |-- blog_url: string (nullable = true)
 |-- category_code: string (nullable = true)
 |-- competitions: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- competitor: struct (nullable = true)
 |    |    |    |-- name: string (nullable = true)
 |    |    |    |-- permalink: string (nullable = true)
 |-- created_at: string (nullable = true)
 |-- crunchbase_url: string (nullable = true)
 |-- deadpooled_day: long (nullable = true)
 |-- deadpooled_month: long (nullable = true)
```

This allows the user to see what queries are necessary to reach the desired view.

Testing Environment

In order to test our platforms properly, we needed to make sure that they'd be running on the exact same hardware. We toyed with the idea of using a virtual machine that had all three platforms installed, but ultimately decided to use Amazon Web Services' free tier as our testing environment. This had two major benefits for us: we could use identical cloud computers for consistency, and AWS's imaging process made it easy to back up our work and revert to previous versions of our machines if we made any errors.

Using AWS EC2, we initialized three identical instances(virtual machines) that we could access with ssh keys. The instances are "t2.micro" tier, and have the following specifications:

- 1 Intel Xeon vCPU

- 1GB RAM
- 8GB SSD
- 64-bit Ubuntu 16.04 LTS
- US-West-2 availability region (Oregon, US)
- Low to Moderate Network Performance rating[27]

Each platform was installed on its own instance, and latency testing took place on the instances to ensure uniformity across processing environments.

Metrics

The platforms were evaluated by both the latency of each query and their ease of use. For latency, each query was run 20 times and the average of the query times was used as the official measurement. Ease of use (EoU) was measured categorically, and each platform was ranked against the others with 1 being the best in that category and 3 being the worst, relatively speaking. Evaluated categories included installation of the tool, the overall procedure of using the tool, and the syntax of the queries written for the tool.

Results & Analysis

Data Tool	Installation	Operating Procedure	Syntax	Lines of Code	Average Latency
CouchDB	**1**	Even	2	2	3
MongoDB	**1**	Even	**1**	3	2
Spark	3	Even	3	**1**	**1**

The preceding table shows the rankings of each of our data platforms in the five categories described in the Metrics section. The following sections address each category in more detail by platform, and includes our analysis and comparisons.

Installation

CouchDB

Installing CouchDB from the command line on Linux machines is not the easiest thing, but isn't too bad if you're using a version prior to 2.0. Throughout most of the project, to familiarize ourselves with CouchDb, we were using version 2.0.0 on a 64-bit Windows 10 machine. For final testing we installed version 1.6.1 on the virtual instance. This decision was made because version 2 of CouchDB was optimized for ease-of-use, primarily through its graphical user interface and explorer-style installation. Installing it through the command line resulted in numerous errors that couldn't be easily resolved, whereas previous versions were designed with terminal installation more in mind. While version 2.0.0 made significant advances in user interaction, the processing time of the database itself was largely left unchanged, so this decision doesn't affect latency results. All EoU observations were made with respect to version 2.0.0.

Using version 1.6.1, installation was simply adding the proper PPA repository and then using an apt-get command to install CouchDB[28]. (Installing version 2.0.0 was so difficult due

to the requirement to manually install all dependencies, since the stable CouchDB PPA repository contains version 1.6.1.):

```
sudo apt-get install software-properties-common
sudo add-apt-repository ppa:couchdb/stable
sudo apt-get install couchdb
```

CouchDB's new users are automatically set up during this process, so some quick permissions changes are recommended so as to move a few files and directories away from root-only access:

```
sudo chown -R couchdb:couchdb /usr/bin/couchdb /etc/couchdb /usr/share/couchdb
sudo chmod -R 0770 /usr/bin/couchdb /etc/couchdb /usr/share/couchdb
```

This took care of CouchDB's numerous dependencies in in just a couple steps, and the database could then immediately be restarted and accessed through localhost port 5984:

```
sudo systemctl restart couchdb
curl localhost:5984
```

MongoDB

Installing MongoDB on Ubuntu was extremely simple and only took four commands from the terminal to complete. The first step is to import the public GNU Privacy Guard (GPG) key used by the MongoDB package management system[29]:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 0C49F3730359A14518585931BC711F9BA15703C6
```

This allows the Ubuntu package management system (ie. dpkg and apt) to ensure package consistency and authenticity. The next steps include creating a list file for MongoDB:

```
echo "deb [ arch=amd64,arm64 ] http://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.4 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.4.list
```

and reloading the local package database:

```
sudo apt-get update
```

The last step in the installation process is to install the MongoDB packages for the latest stable version:

```
sudo apt-get install -y mongodb-org
```

Given that this process requires zero prerequisites and can be completed with four commands ran from the terminal, MongoDB was awarded an EoU score of 3 for installation of the infrastructure. However, this only provides the ability to use the mongo shell. In order to use MongoDB with your application you must use one of its many available driver packages.

The Java drivers for MongoDB were also very simple to find and install, the only challenge being that multiple drivers are needed for basic use (mongodb-driver, mongodb-driver-core, and bson). Using an IDE such as Eclipse or IntelliJ makes it easy to add these jar files to the build path of your project, after which you should be ready to use the MongoDB Java drivers for your application.

Spark

Installing Spark on Ubuntu was a multi-step procedure and required the additional installations of Java and Scala. To first install Java, the following commands were used:

```
sudo apt-add-repository ppa:wedupd8team/java
sudo apt-get update
sudo apt-get install oracle-java7-installer
```

To install Scala, the following commands were used:

```
wget http://www.scala-lang.org/files/archive/scala-2.10.4.tgz
sudo mkdir /usr/local/src/scala
sudo tar xvf scala-2.10.4.tgz -C /usr/local/src/scala/
```

Next the .bashrc files needed to be updated to set the SCALA_HOME variable and path.

```
vi .bashrc
export SCALA_HOME=/usr/local/src/scala/scala-2.10.4
export PATH=$SCALA_HOME/bin”$PATH
```

The .bashrc file was then restarted by using the command:

```
. .bashrc
```

Since the building of Apache Spark depends on git, the following command was used to install git:

```
sudo apt-get install git
```

The next step is to download and install the preconfigured Apache Spark and Hadoop .tar file itself. For this project, Spark 2.0.1 was downloaded and extracted along with Hadoop 2.7.

```
wget http://d3kbcqa49mib13.cloudfront.net/spark-2.0.1-bin-hadoop2.7.tgz
tar xvf spark-2.0.1-bin-hadoop2.7.tgz
```

Due to large amount of commands for installation, Apache Spark was rated the most complex of the three in the metrics.

Overall Procedure

CouchDB

CouchDB, once installed on a machine, runs on startup unless specified otherwise, and a simple terminal command will get it running if it isn't already:

```
couchdb start
```

The only difficulty in procedure comes with learning to use cURL commands, if the user is not familiar with them to begin with. Simple cURL commands using the HTTP API can provide the user with all the data they need, from version to query syntax to advanced usage statistics. Data and views can be added using POST commands. For example, the JSON data set we used was added to the database with this one-line command:

```
curl -H "Content-Type:application/json" -d @MYFILE.json -v POST
"http://localhost:5984/mqp/_bulk_docs"
```

From there on out, we utilized a combination of JSON files, cURL commands, bash scripts and a python script to do all testing. Because CouchDB saves views as design documents, the map and reduce code was put into JSON object format and loaded using a cURL command. To avoid excess typing, we bash scripted the the view creation, as well as the cURL command that then queried the view. Likewise, we bash scripted the "delete view" process, making use of a simple python script that parsed the output string from a GET call in order to delete the current revision number of a document, since delete calls require both the document ID and revision number.

MongoDB

A single command is needed to start MongoDB services, enabling a user to locally enter the mongo shell or connect to the database using one of the driver languages:

```
sudo service mongod start
```

Once the mongo services were started it was very straightforward to enter the mongo shell and create a database, as well as a collection which are the MongoDB equivalent to an RDBMS database and table, respectively. With the services started it is also possible to query from the hosted databases, locally or remotely.

Spark

The spark-shell can be launched by navigating to the appropriate directory and entering the command:

```
./spark-shell
```

Once the spark-shell is open, commands can be entered to load data files and execute queries interactively. This allows for immediate feedback regarding syntax and results.

Syntax

CouchDB

The query syntax for CouchDB itself is incredibly simple, as long as the user is familiar with MapReduce practices. As shown in the Queries section, they're written in JavaScript code and and can easily include CouchDB's built in reducers. When actually loaded as JSON objects through cURL commands, our queries looked like this:

```
{
    "id": "_design/distinct_tags",
    "views" : {
    "distinct_tags": {
    "map": "function (doc) {if(doc.category_code)emit(doc.category_code, null);}",
    "reduce": "_count"
    }
    }
}
```

These JSON docs could then be loaded directly into **/_design/VIEWNAME** as binary data.

Users who are less familiar with the MapReduce key-value process would likely take a little more time to understand how to use CouchDB's query structure. In that particular case, MongoDB's NoSQL syntax might be more intuitive.

MongoDB

Using MongoDB requires a good understanding of JSON data format, as it uses a binary-encoded serialization of JSON documents. The built-in NoSQL functions of the mongo shell are very intuitive to the structure of JSON documents, and also make it very easy to perform complex combinations of aggregation functions. Using the Java driver language is also very straightforward as you can connect to the local database on the default port by using a function that takes no arguments.

Some of the queries were simple enough to pass a specific key within the JSON dataset as an argument, such as the distinct operation, but more complicated queries needed to be constructed using multiple key-value pairs. Just as you construct the MongoDB NoSQL queries by using key-value pairs encapsulated in curled brackets, the Java queries are constructed by using Document objects in which the key-value pair is defined in the constructor. Combinations of these Document objects are then passed to query functions such as projection and aggregation.

Spark

The JSON data file is then loaded and a temporary view is created through these commands:

```
Val df = spark.read.json(“filename.json”)
df.createOrReplaceTempView("View")
```

Spark SQL makes it extremely convenient to query as SQL is used within a Scala command. For example, selecting the sum of Employees from the data set can be done like shown:

```
Val q1 = spark.sql(SELECT SUM(number_of_employees) AS EmpSum FROM View”)
```

The command q1.show() is used to show the outcome of the q1 query.

For nested queries, nested data can be accessed by traversing the schema in the SQL “SELECT” statement. For example, if we want to access the “raised_amount” field that is nested within the “funding_rounds” field, the command is executed like so:

```
val q1 = spark.sql("SELECT name, funding_rounds.raised_amount FROM View")
```

If a column contained an array that needed to be aggregated, the explode() function which creates a new row for each element in the given array or map column. To explode the raised_amount column, the following command is used:

```
val q2 = q1.withColumn("tag",
org.apache.spark.sql.functions.explode(q1.col("raised_amount")));
```

The above command will create an additional column named “tag” and a new row for each element with the raised_amount array. To sum all the values within the raised_amount array would be the same as summing all the values in the “tag” column and group by “name”. This command can be executed like so:

```
val q3 = spark.sql("SELECT name, SUM(tag) as Sum FROM newT GROUP BY name")
```

Lines of Code

CouchDB

CouchDB queries are short. Even in the full JSON format, creating a view would require at most 5 classifiers: the ID, view classifier and chosen view name, the required map code and then the reduce code if any is required. This can all even be written in one line if desired as long as everything is bracketed properly. The final line then required is the GET request needed to query the view.

MongoDB

The lines of code needed to use MongoDB differed between the NoSQL and Java driver implementations. MongoDB NoSQL queries are run using a single command, but this command is sometimes composed of multiple key-value pairs which perform sub-operations. This

sometimes complicated query type is still considered a single line, as it is a single command, but it is good practice to construct the query in a text editor organized hierarchically as you would with blocks of code.

The Java drivers, however, require these sometimes complicated combinations of key-value pairs to be constructed using Document objects, as mentioned in the syntax section. This increases the number of lines needed for the query by the number of Document objects that need to be created. Given the similarity in syntax between MongoDB NoSQL and Java drivers, the additional lines of code are very intuitive and should not be viewed as a negative aspect of the tool.

Spark

When using the spark-shell, queries can be executed in a line by line basis. Assuming an additional view does not need to be created, Spark SQL can query a JSON data file in one line. This allows for very efficient efficient query creation. An additional line of code is required for creating additional views and queries. For the ten queries performed in this study, the shortest was one line, and the longest was five lines (not including the line for loading the JSON file and the line for creating the view).

Latency

Query (Latency Measured in Milliseconds)			
	CouchDB	*MongoDB*	*Spark*
acq_price	14.90	2.43	11.65
after2010	56.65	2.42	8.85
count_employees	14.55	0.52	6.20
distinct_currency	17.10	44.97	11.00
distinct_tags	14.20	37.66	1.95
funding	43.90	0.52	13.35
image_size	819.45	0.52	14.70
offices	14.55	0.51	7.30
urls	1010.75	1.80	1.90
year_founded	14.55	0.52	3.95
Average:	202.06	9.19	8.09

[2]

CouchDB

Measuring latency for CouchDB is a unique process. Queries are actually run on views, which were explained earlier as being design documents that store essentially pre-queried data to make it quick and easy to run the same queries very often. This is one of the main functionalities of CouchDB that make it very useful if you're running a website that always has to retrieve the same data sets.

Views are posted to the database but not actually created until the first time they're queried. The very first time a query is run on a new view, it's *very* slow. For our testing environment and 80MB dataset, first-time queries took an average of 12.6 seconds. For larger datasets, say 10+GB, first-time queries could take multiple hours. After the first run queries become much faster. We originally thought that it would be more accurate to create the view, run the query, delete the view, and then repeat in order to run latency measurements. However, because CouchDB is a revision-only databased, deleted design documents that define views aren't truly deleted, only marked as deleted, unless the database is purged. Purging is an arduous process, so it was decided that the most accurate way to measure latency was to leave the views intact and simply make a note of the first-time query run being significantly longer.

[2] For full-verbosity latency measurements, see the Appendix.

Because cURL queries were run on the terminal line, either by hand or by bash script, the built-in Linux system time command was used to measure latency. It's also important to mention that CouchDB is actually not very fast query-wise. This is because it depends on HTTP, which is not a very low-latency protocol, and is the reason CouchDB appears to be so outperformed by MongoDB and Spark.

MongoDB

The time taken to run each of our queries using MongoDB was much lower than that of CouchDB, and in most cases lower Spark's, mainly due to the infrastructure's extensive use of RAM. MongoDB maps all data into RAM and leaves the task of memory management to the operating system. In the case of a very large dataset, MongoDB has to at least have the indexes for each of the databases in RAM to prevent I/O performance bottlenecks[30]. Using the distinct operation, however, caused MongoDB to have a significantly greater query latency, to the point that the average latency of MongoDB queries became greater than that of Spark.

Given the fact that we only used a single database and collection with 76 MB of data, most of the data, if not all, was likely able to be mapped into RAM. This further explains the difference in latency between the big data infrastructures in use, as MongoDB was able to access all the needed data from RAM. The table below shows the latency statistics, measured in microseconds, for each of our queries run on MongoDB. The last row shows the average runtime for the specified query in bold.

The reason why MongoDB was evaluated using microseconds rather than seconds or milliseconds is that many of the queries consistently ran in under a single millisecond. These latency measurements don't account for the time needed to connect to the database from the Java driver code. Connecting to the database adds a slight overhead to overall latency, but is only done when accessing data using a driver language. Since the pre-query processes for each infrastructure differed, the time to complete such operations was not measured or compared.

Given the fact that the dataset used in this project is smaller than the amount of RAM on most systems, MongoDB would be better evaluated with an amount of data that exceeds the available RAM on the system being used. I/O operations are the most common performance

bottleneck because MongoDB, when there is more data than available RAM, has to push memory to disk rather than keeping it all available in RAM.

Spark

Apache Spark had the lowest overall average query time and an average data-load time of 1.02 seconds, which was expected due to the fact the platform uses in-memory computation. To calculate latency, a time function written in scala was utilized within the shell. For example, to measure the latency of the count_employees query, this script was entered within the Spark-shell:

```
def time[A](f: => A) = {
  val s = System.nanoTime
  val ret = f
  println("time: "+(System.nanoTime-s)/1e6+"ms")
  ret
}
time {
val q1 = spark.sql("SELECT SUM(number_of_employees) AS EmpSum FROM test")
}
```

This function recorded the latency of all executions with the time brackets; in this case, all SparkSQL commands. When the execution was completed, the latency value was returned within the shell. The query executing the selecting of the four urls for each company had the lowest latency for Apache with an average latency of 1.9 milliseconds:

```
def time[A](f: => A) = {
  val s = System.nanoTime
  val ret = f
  println("time: "+(System.nanoTime-s)/1e6+"ms")
  ret
}
time {
val q4 = spark.sql("SELECT crunchbase_url, homepage_url, blog_url, blog_feed_url FROM test")
}
```

The Apache Spark query with the highest latency, with an average value of 14.7 milliseconds, was the selection of each company name with their smallest available image size. This query required multiple view creations and multiple sub-queries resulting in a longer latency, as shown:

```
def time[A](f: => A) = {
  val s = System.nanoTime
  val ret = f
  println("time: "+(System.nanoTime-s)/1e6+"ms")
  ret
}
time {
val q1 = spark.sql("SELECT name, screenshots.available_sizes FROM test")
val q2 = q1.withColumn("tag", org.apache.spark.sql.functions.explode(q1.col("available_sizes")));
val q3 = q2.withColumn("tag2", org.apache.spark.sql.functions.explode(q2.col("tag")));
q3.createOrReplaceTempView("newT")
val q4 = spark.sql("SELECT name, MIN(tag2) as min FROM newT GROUP BY name")
}
```

Apache Spark did not have the lowest query time in our study, however it did have the lowest average query time. Apache Spark had a difference in query times of 12.8 milliseconds, whereas MongoDB has a difference of 44.46 milliseconds, and CouchDB at 996.55 milliseconds. In regards to data-load time, Apache Spark loaded all 75MB of data in an average of 1.02 seconds which is faster than CouchDB by 10.98 seconds. This load time is very useful when loading large datasets, as our study concluded datasets can be loaded over 10x faster than CouchDB.

Conclusions

CouchDB is definitely more targeted towards ease of use for web application developers than pure speed. The static data views make it very effective for providing data in predetermined formats, and the HTTP API makes those formats quick and simple to access. When using the Futon or Fauxton user interfaces, creating queries and managing the infrastructure is remarkably intuitive. However because we did our testing purely through command-line arguments, we didn't benefit from that intuitiveness. Additionally, CouchDB falls behind when evaluated quantitatively. The HTTP protocol is simply not fast, and CouchDB specifically has trouble with queries that require a larger amount of data be returned. In our case, that was clearly evident from the latency measurements of our projection queries.

MongoDB was very simple to install and get running on Ubuntu machines. The syntax was also very intuitive to the structure of JSON documents and the type of query being performed. This simplicity paired with MongoDB's ultra-low latency makes it a top competitor among data storage and processing engines. However, a major shortcoming of MongoDB comes when performing the *distinct* operation, as it performs a full scan of the dataset and can cause I/O performance bottlenecks due to its extensive use of RAM.

Apache Spark is an efficient platform in terms of writing and executing queries. In-memory data processing allowed for the avoidance of costly disk access, and for queries to be performed at memory speeds. Despite the positive performance remarks, Apache Spark has several required dependencies, which made the installation process relatively complex. Overall, the combination of the Spark-shell and Spark SQL enable concise query creation and low-latency execution.

Although there is no clear cut best platform, each of them has strengths and weaknesses that lend themselves to different areas of use. CouchDB uses HTTP connections to transfer data, therefore making it most useful in web applications. MongoDB has many use cases and ultra-low latency, making it a top choice for any application that requires a flexible data schema. Apache Spark excels in its abilities to process and analyze data with ultra-low latency, making it a top choice for any application that needs to provide real-time statistics and machine learning capabilities. Future directions of this study would involve:

- An increase in data size, moving into the range of multiple gigabytes. This would be more representative of enterprise-level datasets and further distinguish performance differences between the platforms.
- Cluster analysis, as opposed to the single-instance performance measured on AWS EC2 instances.
- A greater number of queries, including variations in filtering percentages, different levels of nesting, and both read and write queries where applicable.

Given the timeline of this study, our research acts as a starting point for a more detailed benchmarking of these three platforms.

Acknowledgements

We would like to thank our advisor, Mohamed Eltabakh, for his guidance and support throughout the duration of the project.

References

[1] Ghemawat, S., Gobioff, H., & Leung, S. (n.d.). The Google File System (Rep.). Retrieved https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

[2] What is Apache Hadoop? (n.d.). Retrieved January 13, 2017, from https://hortonworks.com/apache/hadoop/

[3] Apache Software Foundation. (2017, January 26). Welcome to Apache™ Hadoop®! Retrieved from http://hadoop.apache.org/

[4] Ting, D. (2017, January 24). PoweredBy - Hadoop Wiki. Retrieved January 17, 2017, from https://wiki.apache.org/hadoop/PoweredBy#N

[5]Rose Technologies. (2012, November 26). Hadoop Architecture and Deployment - Rose Technologies. Retrieved from http://www.rosebt.com/blog/hadooparchitecture-and-deployment

[6]MacLean, D. (2011). A Very Brief Introduction to MapReduce. Retrieved from Stanford University website: http://hci.stanford.edu/courses/cs448g/a2/files/map_reduce_tutorial.pdf

[7] Dean, J., & Ghemawat, S. (n.d.). MapReduce: Simplified Data Processing on Large Clusters. Retrieved from Google, Inc. website: https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

[8] Exploring CouchDB. (2009, March 31). Retrieved February 28, 2017, from http://www.ibm.com/developerworks/opensource/library/os-couchdb/index.html

[9] Jackson, J. (2010, July 14). CouchDB NoSQL Database Ready for Production Use. Retrieved February 28, 2017, from http://www.pcworld.com/article/201046/article.html

[10] The Apache Software FoundationBlogging in Action. (n.d.). Retrieved February 28, 2017, from https://blogs.apache.org/couchdb/entry/welcome_bigcouch

[11] Erlang whitepaper. (n.d.). Retrieved February 28, 2017, from https://web.archive.org/web/20111025022940/http://ftp.sunet.se/pub/lang/erlang/white_paper.html

[12] Cassandra vs MongoDB vs CouchDB vs Redis vs Riak vs HBase vs Couchbase vs OrientDB vs Aerospike vs Neo4j vs Hypertable vs ElasticSearch vs Accumulo vs VoltDB vs Scalaris vs RethinkDB comparison. (n.d.). Retrieved February 28, 2017, from https://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis

[13] Eventual Consistency. (n.d.). Retrieved February 28, 2017, from http://guide.couchdb.org/draft/consistency.html

[14] Hadoop Tool: CouchDB Assignment Help. (n.d.). Retrieved February 28, 2017, from http://www.myassignmenthelp.net/couchdb-assignment-help

[15] Kalla, R. (2012, February 21). How does MongoDB compare to CouchDB? What are the advantages and disadvantages of each? - Quora. Retrieved from https://www.quora.com/How-does-MongoDB-compare-to-CouchDB-What-are-the-advantages-and-disadvantages-of-each

[16] MongoDB, Inc. (2017). Do What You Could Never Do Before | MongoDB. Retrieved from https://www.mongodb.com/what-is-mongodb

[17] MongoDB, Inc. (2017). MongoDB at Scale | MongoDB. Retrieved from https://www.mongodb.com/mongodb-scale

[18] MongoDB, Inc. (2017). MongoDB Architecture | MongoDB. Retrieved from https://www.mongodb.com/mongodb-architecture

[19] MongoDB, Inc. (2017). Use Cases | MongoDB. Retrieved from https://www.mongodb.com/use-cases

[20] Zaharia, Matei; Chowdhury, Mosharaf; Franklin, Michael J.; Shenker, Scott; Stoica, Ion. Spark: Cluster Computing with Working Sets (PDF). USENIX Workshop on Hot Topics in Cloud Computing (HotCloud).

[21] Zaharia, Matei; Chowdhury, Mosharaf; Das, Tathagata; Dave, Ankur; Ma, Justin; McCauley, Murphy; J., Michael; Shenker, Scott; Stoica, Ion. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing (PDF). USENIX Symp. Networked Systems Design and Implementation.

[22] Armbrust, M., Ghodsi, A., Zaharia, M., Xin, R. S., Lian, C., Huai, Y., . . . Franklin, M. J. (2015). Spark SQL. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15. doi:10.1145/2723372.2742797

[23] "Cooper, Brian F; et al. "Benchmarking cloud serving systems with YCSB" (PDF). Yahoo Research.

[24] Kamat, G. (2015, August 31). YCSB, the Open Standard for NoSQL Benchmarking, Joins Cloudera Labs - Cloudera Engineering Blog. Retrieved November 27, 2016, from http://blog.cloudera.com/blog/2015/08/ycsb-the-open-standard-for-nosql-benchmarking-joins-cloudera-labs/

[25] Cooper, B. F. (2016, September 19). Brianfrankcooper/YCSB. Retrieved November 27, 2016, from https://github.com/brianfrankcooper/YCSB/releases/tag/0.11.0

[26] JudCON 2013. (2013). JudCON: JBoss Users & Developers Conference. Retrieved November 27, 2016, from https://www.jboss.org/dms/judcon/2013india/presentations/day1track2session2.pdf

[27] Amazon. (n.d.). EC2 Instance Types – Amazon Web Services (AWS). Retrieved from https://aws.amazon.com/ec2/instance-types/

[28] Pyasi, A. (2016, June 3). How To Install CouchDB and Futon on Ubuntu 16.04. Retrieved from http://linoxide.com/linux-how-to/install-couchdb-futon-ubuntu-1604/

[29] MongoDB, Inc. (2016). Install MongoDB Community Edition on Ubuntu — MongoDB Manual 3.4. Retrieved from https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/

[30] Farrugia, C. (2012, November 5). MongoDB Performance Pitfalls - Behind The Scenes. Retrieved from https://www.revulytics.com/blog/mongodb-performance-pitfalls-behind-the-scenes

Appendix

Run #	CouchDB Query (Latency Measured in Milliseconds)										
	acq_price	after2010	count_employees	distinct_currency	distinct_tags	funding	image_size	offices	urls	year_founded	
1	16	49	15	18	15	31	789	15	977	17	
2	15	46	14	17	14	28	819	14	965	14	
3	15	62	14	17	14	43	823	15	1030	14	
4	15	57	14	17	14	44	805	15	1008	14	
5	15	58	16	17	14	45	831	14	1005	14	
6	15	58	14	17	15	45	827	15	999	14	
7	15	56	14	17	15	46	825	14	1010	14	
8	15	57	16	16	14	45	814	14	1007	17	
9	15	57	14	17	14	45	812	14	1011	14	
10	14	57	20	18	14	45	819	14	1009	14	
11	15	58	14	17	14	44	832	14	1017	14	
12	15	57	14	17	15	46	822	16	1023	15	
13	14	58	14	17	14	45	826	14	1024	14	
14	15	58	14	17	14	45	824	15	1006	14	
15	15	58	14	16	14	46	822	14	1018	18	
16	15	58	14	18	14	53	818	14	1024	14	
17	15	58	14	17	14	46	825	14	1028	14	
18	15	57	14	18	14	44	819	14	1013	14	
19	15	57	14	17	14	46	821	18	1020	14	
20	14	57	14	17	14	46	816	14	1021	14	
	14.90	**56.65**	**14.55**	**17.10**	**14.20**	**43.90**	**819.45**	**14.55**	**1010.75**	**14.55**	

Run #	MongoDB Query (Latency Measured in Milliseconds)									
	acq_price	after2010	count_employees	distinct_currency	distinct_tags	funding	image_size	offices	urls	year_founded
1	2.36	2.33	0.53	44.44	36.38	0.51	0.51	0.51	1.83	0.54
2	2.42	2.43	0.52	45.71	36.46	0.55	0.53	0.53	1.80	0.50
3	2.36	2.39	0.51	44.74	37.07	0.51	0.50	0.51	1.83	0.51
4	2.34	2.40	0.51	44.52	36.88	0.53	0.52	0.51	1.79	0.51
5	2.36	2.45	0.50	47.71	38.02	0.53	0.53	0.51	1.83	0.51
6	2.37	2.37	0.54	44.03	38.50	0.54	0.51	0.55	1.81	0.52
7	2.41	2.38	0.54	45.34	37.76	0.54	0.53	0.51	1.74	0.52
8	2.37	2.36	0.52	45.60	37.98	0.55	0.55	0.51	1.82	0.50
9	2.43	2.37	0.51	45.71	37.69	0.52	0.50	0.51	1.78	0.53
10	2.42	2.40	0.53	44.27	38.04	0.51	0.51	0.51	1.86	0.50
11	2.33	2.35	0.54	45.63	38.66	0.51	0.52	0.51	1.80	0.54
12	2.87	2.39	0.52	44.31	37.23	0.50	0.51	0.50	1.80	0.51
13	2.83	2.42	0.52	44.30	38.25	0.52	0.51	0.52	1.84	0.51
14	2.34	2.45	0.51	45.20	38.04	0.55	0.52	0.53	1.79	0.56
15	2.43	2.41	0.51	45.94	37.59	0.50	0.52	0.52	1.81	0.50
16	2.37	2.48	0.52	44.13	36.57	0.52	0.52	0.51	1.74	0.50
17	2.30	2.49	0.52	44.22	38.60	0.54	0.50	0.50	1.89	0.51
18	2.40	2.85	0.55	44.74	37.29	0.50	0.51	0.54	1.74	0.54
19	2.42	2.30	0.53	44.07	38.74	0.50	0.50	0.50	1.75	0.50
20	2.40	2.44	0.52	44.77	37.41	0.51	0.54	0.52	1.74	0.52
	2.43	**2.42**	**0.52**	**44.97**	**37.66**	**0.52**	**0.52**	**0.51**	**1.80**	**0.52**

Run #	Spark Query (Latency Measured in Milliseconds)									
	acq_price	after2010	count_employees	distinct_currency	distinct_tags	funding	image_size	offices	urls	year_founded
1	11	9	7	12	2	14	15	8	2	4
2	10	8	7	11	2	16	14	7	1	4
3	10	7	6	10	3	13	13	8	3	5
4	14	6	7	14	3	14	16	7	3	4
5	14	9	6	13	2	13	15	7	2	4
6	12	8	6	12	1	14	15	8	2	3
7	10	9	5	11	2	14	14	7	1	4
8	9	8	6	10	3	13	15	9	2	4
9	15	8	5	10	2	13	14	8	1	3
10	12	10	6	11	2	15	15	7	2	4
11	9	11	6	11	1	11	15	6	2	5
12	14	9	7	10	1	12	13	7	2	4
13	9	9	6	12	2	14	16	7	1	5
14	10	10	6	11	1	13	13	6	2	4
15	12	11	5	10	3	13	14	7	1	4
16	12	8	6	10	2	14	14	8	1	3
17	13	8	7	10	2	12	15	7	2	4
18	11	9	6	11	1	11	15	7	3	5
19	14	10	7	10	2	14	17	8	2	3
20	12	10	7	11	2	14	16	7	3	3
	11.65	**8.85**	**6.20**	**11.00**	**1.95**	**13.35**	**14.70**	**7.30**	**1.90**	**3.95**