SITEWIT CORPORATION: SQL OR NOSQL THAT IS THE QUESTION

Teaching Cases

Donald J. Berndt
College of Business
University of South Florida
dberndt@usf.edu

Ricardo Lasa
SiteWit Corporation
Ricardo.Lasa@SiteWit.com

James McCart*
SiteWit Corporation
James.McCart@SiteWit.com

Abstract

This teaching case focuses on a start-up company in the Web analytics and online advertising space, which faces a database scaling challenge. The case covers the rapidly emerging NoSQL database products that can be used to implement very large distributed databases. These are exciting times in the database marketplace, with a flock of new companies offering scalable database systems for the cloud. These products challenge existing relational database vendors that have come to dominate the market. The case outlines four potential solutions and asks students to make a choice or suggest a different alternative to meet database scaling goals.

Keywords: IS curriculum and education, database management systems (DBMS), software architecture

* Please feel free to contact any of the authors for a teaching note that supports the case.

Executive Summary

Is the time right? The time is right! And here is why…

Ricardo Lasa, CEO and co-founder of SiteWit Corporation, was always chastising his technical team that the "biggest risk facing the company is the engine." SiteWit provides cross-platform services aimed at helping small (or even medium-sized) business customers effectively advertise on search engines like Google (AdWords) and Bing (adCenter), as well as other online social networking or display advertising venues. At the core, SiteWit is a Web analytics company that tracks all the detailed organic and paid advertising traffic on client websites. SiteWit then uses this very detailed data to deliver software-as-a-service (SaaS) products that handle a variety of tasks from automated keyword bidding to campaign optimization. These products rely on a foundation of website analytic data warehousing and automated data mining, so data quality is of paramount concern.

Lasa and his team faced a critical technology challenge in scaling the core database systems to meet rapidly escalating data volumes. Should he stick with well-known relational database technologies? His core team was well versed in the Microsoft technology stack and had worked together for more than a decade on software-as-a-service (SaaS) applications. Or should he re-implement core components in newer, highly distributed NoSQL databases in search of competitive advantages? So, the decision could be concisely summarized as follows: SQL or NoSQL that is the question.

1. **Do nothing.** SiteWit Corporation is a lean startup with limited resources. Do we really need to add new technologies and more uncertainty at this stage?
2. **Proceed cautiously with NoSQL technology through limited experiments.** It might be reasonable to pick some component that could be implemented using NoSQL technology to gain experience and validate the technology.
3. **Develop a new product, alone or through a partnership, that makes use of NoSQL technologies.** A couple of potential SiteWit partners are experimenting with or even already resting firmly on NoSQL technologies, so one strategy might be to learn through collaboration.
4. **Take a leap of faith.** Again, SiteWit is an early stage company facing plenty of risk factors. Adding a few more for an important competitive advantage may be a reasonable tradeoff.

Whatever decision he reached, scaling the core database technology was an immediate concern. New customers were arriving daily in response to online advertising campaigns. It would not get any easier.

Introduction

Ricardo Lasa, CEO and co-founder of SiteWit Corporation knew the company was approaching a waypoint that may require a course correction. Things were going well. The company had already passed many critical points that can sink a startup. The core team had developed a complex product, which was already selling in the marketplace. Along the way, a beta version had allowed his company to raise money from angel investors and then from a small Series A funding round. With money in the bank and several products in the market, the company was adding a bit more development depth and focusing on growing the sales team. So, why was he again facing more sleepless nights?

The challenge was coming from a critical technical issue: scaling. At the core, SiteWit is an analytics company. A very large of amount of detailed Web analytics data is collected and processed as part of delivering online advertising services, such as keyword bidding, campaign optimization, and predictive analytics for re-marketing. The prospect of adding many more clients meant facing dramatic growth in the sheer volume of data being processed. While they had already faced several milestones and had re-engineered key processes to meet performance goals, explosive growth would certainly bring new challenges. Chief among these challenges would be to scale the core database technologies. SiteWit was already up and running with a robust architecture using cloud-based infrastructure services. However, the core database engines were standard commercial relational database management systems (in this case Microsoft SQL Server). It is certainly true that relational database vendors have added many features to support very large databases, but achieving Web scale is something different. Companies like Google

and Amazon had developed whole new infrastructures to support their businesses. More importantly, some other Web analytic startups had embraced next-generation distributed database technologies that grew out of earlier efforts by Google and other "big data" pioneers.

So, to continue the nautical theme, Lasa and his technical team were facing the need to choose a course. One course involved sticking with well-known relational database technologies with some tacks (in steady winds) along the way to adjust to growing demands. His core team was well versed in the Microsoft technology stack and had worked together for more than a decade on software-as-a-service (SaaS) applications. The other course was akin to a jibe in heavy winds, a high-risk and dramatic change in direction that involved re-implementing core components in newer, highly distributed NoSQL databases. So, the decision could be summarized as follows: SQL or NoSQL that is the question.

The Technology

The heart of SiteWit's technological challenge revolved around the core technologies for managing its data, or rather managing its increasingly large amounts of data. Since the 1980s, database systems had been dominated by a particular approach: the relational database. SiteWit was already up and running on a SQL-based relational platform that used cloud-based infrastructure services. While current relational database systems incorporated many features for scaling, more recent big-data companies were making use of newer highly-distributed database systems. To better understand the situation, it is useful to examine how database technology has evolved.

Early Database Approaches

Databases first began to appear in the late 1950s and early 1960s, spurred by two technological factors: the increasing reliability of computer processors and the expansion of secondary storage capacity in the form of tapes and disk drives. Early "databases" tended to be sequential or random access files that could be searched or processed by any program that knew the precise internal file structure. Eventually, standalone database systems that were application-independent evolved. For example, IBM developed the first true database model: the hierarchical data model. IBM's IMS system used this tree-like organization and, even today, the approach is used for some systems, such as the Microsoft Windows Registry.

Many applications, such as the early airline reservation system, delivered high performance on hardware that would be considered primitive by today's standards using these early database designs. Using these designs, however, demanded significant concessions with respect to flexibility. Essentially, you had to know all the intended uses of your data before designing the data organization. If a different type of search or transaction was later desired, it was likely to prove highly inefficient, or even impossible to implement on an existing database. These early database models also required substantial skills from their users. They required each query to be written like a computer program. The languages used to form these queries were **procedural** languages, meaning that the details of how to produce the answer had to be specified in an unambiguous computer algorithm. While good database performance could be achieved, the effort involved was significant and demanded a highly skilled work force. These factors all drove demand for a more adaptable approach to data management.

Relational Databases: A Classic Success Story

E. J. Codd first proposed relational databases and the underlying theories in a 1970 paper (Codd 1970). In a subsequent Turing Award lecture titled "Relational Database: A Practical Foundation for Productivity," he highlighted the overall objective: to improve the productivity of database programmers (Codd 1982). The relational database model was a radical departure that rested upon a few powerful set-theoretic operations that combine separate data tables (or relations) to produce an answer set. The queries were specified using relational algebra or more commonly the standards-based Structured Query Language (SQL). SQL allowed a database user to express his or her query in a **declarative** form, without any detailed programmatic instructions. That is, the form of the results and inputs were specified, without any concern for how the results would be computed. So database users no longer needed to be

The authors talks about the problems of relational databases starting from the

skilled programmers or spend their time writing complicated query code – hence the gain in productivity.

The problem was that the early relational databases, while technically elegant, were horribly inefficient. Queries would simply take a long time to produce answers. This encouraged widespread research efforts on database optimization techniques that exploited heuristic rules, indexing structures, and statistics to create more efficient query execution plans. This approach substitutes computer-based analysis rather than handcrafted programming to generate the procedural steps necessary to efficiently answer a query. These research efforts were quite successful and led to a classic laboratory to marketplace transfer of technology, with companies such as Oracle, IBM, and later Microsoft offering robust relational database products.

In 2012, over thirty years after their commercial introduction, relational database systems (SQL technologies) remained at the core of virtually all corporate data infrastructures and power most day-to-day operational processes, from accounting systems to comprehensive ERP systems. There was some indication, however, that this situation might be ripe for change. In the emerging era of "big data" with an increasingly important role for data analytics, relational approaches were again being challenged by the problems of scaling and distributed transaction processing. In these areas, alternative database (NoSQL) technologies offer some interesting advantages.

Scaling Database Technologies

There are two fundamental approaches to scaling database systems: **vertical scaling** and **horizontal scaling** (Prichett 2008). Vertical scaling is the more straightforward strategy, relying on increasingly powerful computing infrastructures to meet demand. Of course, this strategy can become expensive as progressively more exotic machines lock you into pricey vendors. This path may also take you into database server clustering, with an extra layer of software complexity allowing multiple machines to focus on a single database.

Horizontal scaling takes a different approach, partitioning the data across multiple databases. While this approach is more complex, there are gains in flexibility and the potential to scale for big data applications. Horizontal scaling can be achieved along two dimensions. One dimension involves grouping the data by function and then spreading the functions across multiple databases. The second dimension involves splitting the data within a function across multiple databases (or *shards*). NoSQL platforms often offer built-in support for database sharding as a scaling strategy.

A Brief History of Transaction Processing

The dominant commercial relational database engines all provide sophisticated transaction processing capabilities. A **transaction** is a user-defined unit of work that typically comprises multiple database operations, such as individual queries or update statements. For instance, a simple human resource function may entail reading from several database tables and writing new information to other tables as necessary, all of which should be considered a single transaction. Relational databases provide important transaction processing guarantees, including atomicity, consistency, isolation, and durability (the so-called ACID properties). These fundamental ACID properties can be briefly defined as follows.

- **Atomicity.** All of the statements within a transaction are completed (or none are performed); no partially executed transactions.
- **Consistency.** The database is left in a consistent state after a transaction is executed.
- **Isolation.** A transaction is executed as if it is the only unit of work being processed by the database.
- **Durability.** Once completed (or committed), a transaction will never be reversed.

While industrial quality relational database systems often provide additional functions to speed up transaction processing, there are architectural limits on the degree of parallelism possible. Therefore, very large Internet-scale applications have sought out other big data solutions.

In order to scale traditional relational databases, a few additional servers can be "clustered" to function as

a single database engine, with protocols to keep the independent memory areas synchronized (the so-called cache coherency problem). However, to scale beyond these special-purpose clustered solutions requires a truly distributed collection of database engines, with data spread across all the systems. What happens if we try to process a transaction across database boundaries? In this case, a higher-level protocol must be used to make sure the pieces of a transaction are handled appropriately within each database, still guaranteeing the ACID properties above. For instance, most relational databases make use of an approach called the two-phase commit (2PC) protocol, even though it has some issues with regard to failures of participating distributed databases. The two phases consist of a voting phase in which all participants must indicate (to the query coordinator) that the assigned portion of a transaction can be completed, followed by a commit phase in which the query coordinator issues a commit or abort (depending on the previous votes). All participants must send a "yes" vote for a transaction to reach a commit point. Of course, this ensures consistency after each transaction, though at the cost of a fairly expensive protocol that requires the ACID properties to be met. How can we tradeoff consistency to gain availability and enhanced performance in big data environments?

Basic Availability Soft-State Eventual Consistency (BASE)

An alternative to the traditional ACID properties is captured by the acronym BASE, for basic availability soft-state eventual consistency (which we assume took several tries to discover). Rather than stark opposites, these different models of transaction processing anchor a continuum. Software architects can choose points along this continuum that best suit systems and associated business models. A BASE approach removes the strict focus on consistency after every detailed transaction in favor of achieving "eventual consistency" within a reasonable timeframe. In other words, approximations are fine and need not be based on every single data item.

Think of accounting systems, which keep track of transactions as a business runs, but lag reality until the books are formally "closed" and reconciled for a given period. During much of the time, these accounting systems are used to produce management reports that are reasonable approximations rather than completed financial reports.

The CAP Theorem

Eric Brewer at the 2000 Symposium on Principles of Distributed Computing (PODC) first proposed the CAP theorem as a conjecture. There are many different discussions of his conjecture from both academic and practitioner perspectives, especially as it relates to NoSQL databases. However, the discussions almost always begin with a re-cap of CAP, highlighting three desirable properties of distributed systems: **consistency**, **availability**, and tolerance of network **partitions** (hence CAP). The conjecture is that distributed systems can embrace only two of the three properties, yielding three combinations that describe the underlying tradeoffs: consistent and available (CA), consistent and partition tolerant (CP), and available and partition tolerant (AP) as in Figure 1 below. Two MIT researchers, Gilbert and Lynch, published a proof of the conjecture establishing it as a theorem, though in a somewhat restricted form (Gilbert and Lynch 2002). However, it turns out that the design tradeoffs pursued in many NoSQL databases are somewhat subtler than a straightforward choice between consistency and availability.

Adopting a somewhat simpler perspective of these highly complex discussions, scaling based on any type of distributed system involves partitioning the data across machine boundaries, and therefore requires partition tolerance (P). Thus, highly scalable systems are typically trading off consistency or availability, giving us the CP or AP categories shown in Figure 1 (with some associated NoSQL systems listed). The consistent-available (CA) systems include the traditional relational database management systems (RDMSs), including offerings from companies such as Oracle and Microsoft (like the SQL Server engine being used by SiteWit).

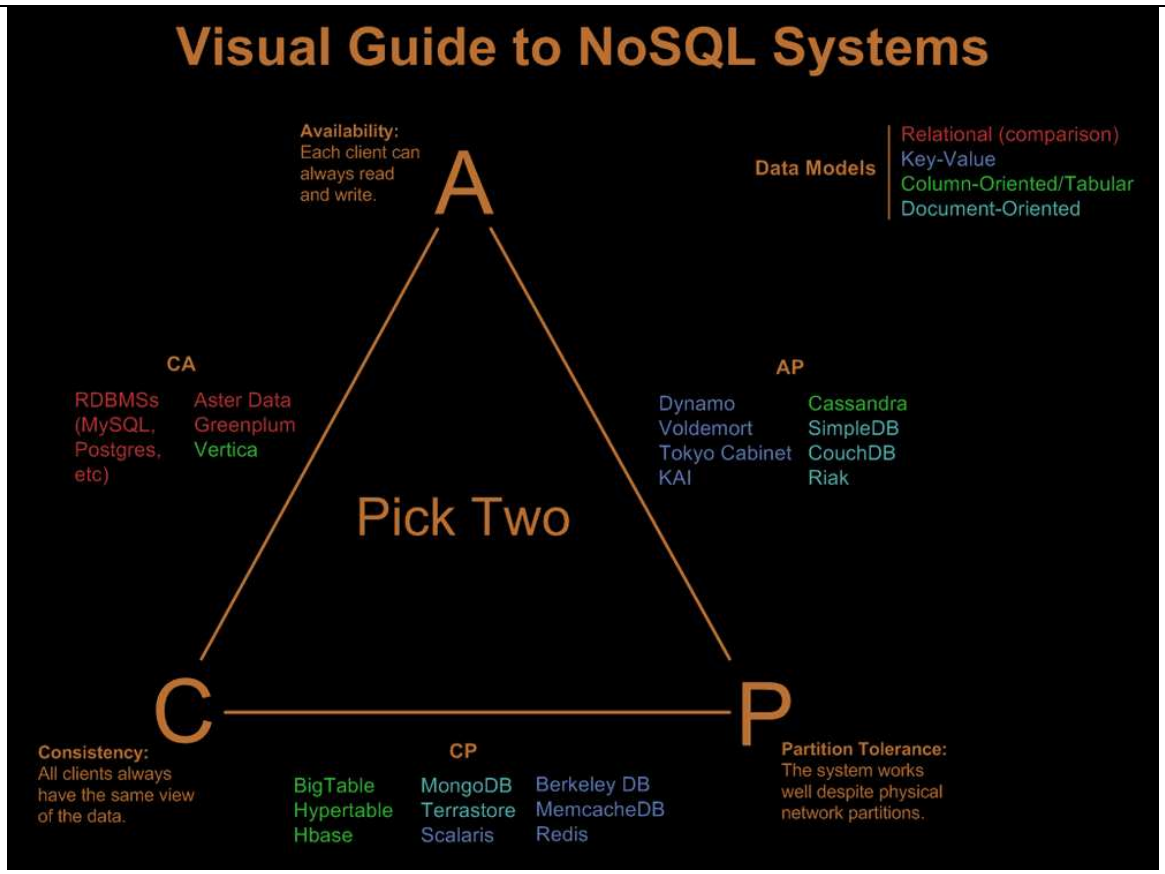

Figure 1. Consistency, Availability and Partition Tolerance Tradeoffs (Hurst 2010)

Google's BigTable

Google's BigTable provided an early and excellent example of these new highly distributed database systems (Chang et al. 2006). Because of its immense scale and innovative philosophy, Google has relied on custom-built infrastructure, including innovative data centers, inexpensive servers, and big data toolkits. BigTable was one of the first generation "Internet scale" highly distributed database systems. BigTable provided a simple data model for storing structured data across a large collection of commodity servers, thereby creating an efficient and cost effective data store at Web scale. By 2006, BigTable was the data store for many recognizable Google projects, such as Google Analytics, Google Finance, Orkut, Writely, and Google Earth. As described by several of the developers, "BigTable has achieved several goals: wide applicability, scalability, high performance, and high availability." What is not to like? In fact, BigTable was re-incarnated in projects such as the Apache Cassandra project (cassandra.apache.org), which have enabled many start-up companies to make use of big data on a solid foundation. For example, Netflix, Twitter, Constant Contact, Digg, and CloudKick are all Cassandra users.

BigTable employed a simple data model and deliberately avoided providing complicated features such as general transaction management. Again echoing the developers, "The most important lesson we learned is the value of simple designs." BigTable did not implement a full relational database model, but goes beyond bare key-value pairs to provide a data store based on row keys, column keys (and column families), with timestamps to support versioning. Client applications interacted directly with BigTable via a lean API, while BigTable itself relied on other building blocks such as the Google File System (Ghemawat et al. 2003). BigTable partitioned a table into ranges of rows or "tablets," which are the fundamental units for distributing data. BigTable then relied on a single master server and many tablet

servers (perhaps thousands) to distribute and manipulate very large tables, with the bulk of all communications going directly through the bank of tablet servers. The early performance benchmarks were impressive, but the cost and sophistication required to create BigTable and the other building blocks necessary for the first wave of Web scale data all but eliminated small business entrants. Fortunately, the next wave of big data entrepreneurs had access to open source and commercial implementations of BigTable-like toolkits!

The NoSQL Landscape

Just as relational database management systems emerged out of a mix of theory and practice, NoSQL approaches were rapidly evolving from the same two influences. On the theory side, the CAP Theorem helped clarify the tradeoffs involved when building distributed "big data" applications. On the practice side, companies such as Google were applying NoSQL-like approaches to great effect and NoSQL databases like MongoDB were being applied by many Web start-ups.

By 2012, database systems that target big data applications were appearing at a rapid rate. Most of these involved spreading both data and processing across many machines, so that much larger amounts of computing power could be effectively harnessed. Nevertheless, bringing together large numbers of distributed machines for highly targeted tasks involved well-known challenges in communication, coordination, and even fault tolerance (Gelernter and Carriero 1992). Intermediate results often needed to be communicated between machines, certain processing steps could be dependent on each other (requiring a specific sequencing), and any machine might fail at the worst moment. All these challenges were made more difficult in the context of running non-stop (24-by-7) big data applications.

Although many new database systems were being branded as NoSQL, their underlying architecture could differ in fundamental ways from one another, greatly impacting how well the database system would work in a given environment. Researchers from Yahoo! Research outlined tradeoffs along four architectural dimensions (Cooper et al. 2010) that could be used to organize and compare NoSQL systems to one another. A brief description of the tradeoffs faced is provided below.

- **Read versus write performance.** How data is written to disk impacts the read and write performance of a system. Writing each record's change to a sequential log file increases write performance, but read performance suffers because all changes to the record must be found and merged from the log before returning the record. Writing the complete record, rather than incremental changes, into a sequential log file increases read performance, but does so at the detriment of write performance.
- **Data partitioning.** How data is accessed also affects performance. The two main methods used by NoSQL systems are row-based and column-based. Row-based partitioning stores all fields of one record contiguously on disk, increasing read/write performance when dealing with an entire record. Column-based partitioning stores a column or groups of columns for multiple records together on disk. Thus, accessing columns or subsets of columns (if grouped together) for multiple records results in better read/write performance than accessing records in their entirety.
- **Latency versus durability.** An important tradeoff also affecting write performance is whether an operation (INSERT, UPDATE, and DELETE) is considered complete once the data is written to memory or disk. Writing to memory decreases write latency, but does so at the risk of losing data if the machine crashes before the contents of memory are written to disk.
- **Synchronous versus asynchronous replication.** Replication of data and processes across multiple machines can impact system availability and performance, along with the risk of data loss. Synchronous replication requires changes to be propagated to all machines designated as replicas, which results in higher write latency. However, read latency and risk of data loss is reduced since all replicas have up-to-date data, and thus can be used to recover data and/or serve queries. Asynchronous replication does not require replicas have up-to-date data, which lowers write latency at the cost of increased read latency (assuming up-to-date data is required) and risk of data loss.

Table 1 provides characteristics for five NoSQL database systems. The characteristics of BigTable are

listed first followed by two database systems based on BigTable: Cassandra and HBase. All three of these database systems store data in column groupings. The final two database systems, Yahoo's hosted database system PNUTS and MongoDB (described in more detail below), store data in a row-based format, where a row represents a document in JavaScript Object Notation (JSON) or similar format (Chodorow and Dirolf 2010; Cooper et al. 2008).

Table 1. Characteristics of NoSQL Database Systems – Adapted from (Cooper et al. 2010)

System	Read / Write Optimized	Data Partitioning	Latency / Durability	Synchronous / Asynchronous Replications
BigTable	Write	Column	Durability	Synchronous
Cassandra	Write	Column	Tunable	Tunable
HBase	Write	Column	Latency	Asynchronous
PNUTS	Read	Row	Durability	Asynchronous
MongoDB	Write	Row	Tunable	Asynchronous

MongoDB

As noted on the project website (mongodb.org), "MongoDB (from "hu**mongo**us") is a scalable, high performance, open source NoSQL database." Records in MongoDB are stored as documents in collections (similar to tables). A document may have fields consisting of strings, integers, dates, arrays, other documents, etc. Unlike tables in a RDBMS, collections in MongoDB are schema-free; meaning (1) the structure of documents within a collection are not set a priori and (2) documents within the same collection may have a different structure from one another. MongoDB also differs from a RBDMS by not supporting joins between collections. Instead, all pertinent information is either (1) embedded within the document it is associated with, or (2) unique identifiers are provided within the document which link documents from different collections together. (Linking documents together requires client-side logic to issue additional queries to obtain all relevant documents and then combine the documents together.)

Figure 2 illustrates an overview of the architecture of MongoDB. The basic components of MongoDB consist of a configuration server, a routing program, shards, and replica sets. A brief description of each component is provided below in more detail (Chodorow and Dirolf 2010).

- **Configuration server** – contains information about what data is stored on which shard.
- **Routing program (Mongos)** – a program which interfaces with the client by routing requests to the proper shard and then aggregating responses back from the shards.
- **Shard** – a Mongo database instance which holds a portion of a collection's data. MongoDB supports autosharding which automatically splits data up according to the shard key and rebalances data amongst shards as needed. The shard key is a field in a collection selected by the database administrator. For example, the name of a user may be used as a shard key and data would then be distributed to shard 1 for names starting with A-L and shard 2 for M-Z.
- **Replica set** – a replication mechanism that consists of a cluster of machines specific to one shard. A replica set consists of one primary node and one or more secondary nodes that asynchronously synch with the primary node. If the primary node goes down the replica set will automatically failover and one of the secondary nodes will be promoted as the new primary node. Once the old primary node is brought back up it will be returned to the replica set as a secondary node and then re-synced with the new primary node.

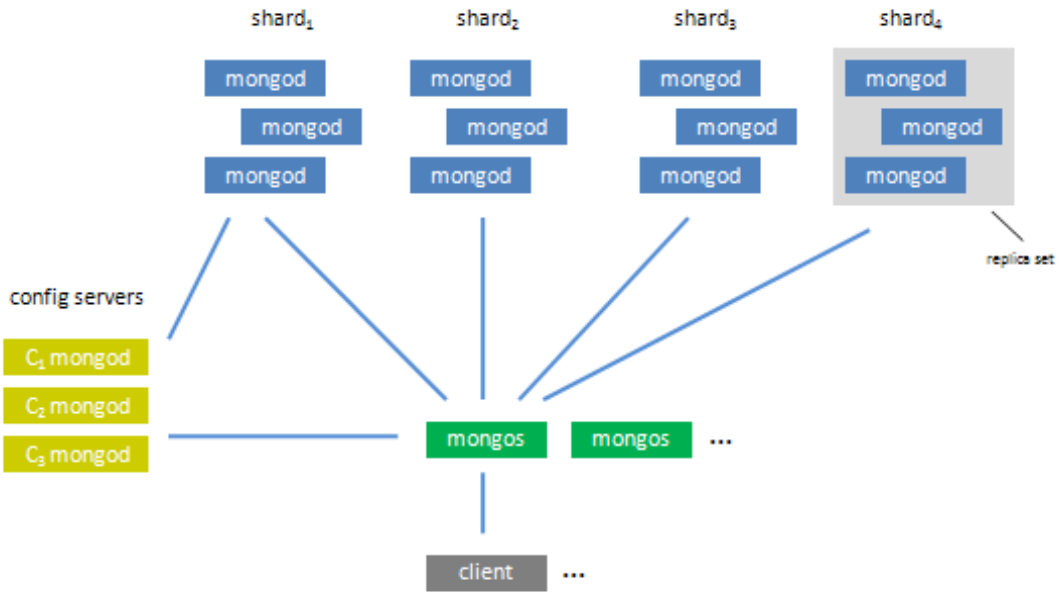

Figure 2. A Distributed Set of MongoDB Servers or Shards (from mongodb.org)

Elliot Horowitz, the CTO of *10gen*—the original developer of the product—described the philosophy behind MongoDB design on the mongodb.org website:

> "MongoDB wasn't designed in a lab. We built MongoDB from our own experiences building large scale, high availability, robust systems. We didn't start from scratch, we really tried to figure out what was broken, and tackle that. So the way I think about MongoDB is that if you take MySQL, and change the data model from relational to document based, you get a lot of great features: embedded docs for speed, manageability, agile development with schema-less databases, easier horizontal scalability because joins aren't as important. There are lots of things that work great in relational databases: indexes, dynamic queries and updates to name a few, and we haven't changed much there. For example, the way you design your indexes in MongoDB should be exactly the way you do it in MySQL or Oracle, you just have the option of indexing an embedded field."

The Competitive Landscape

SiteWit was a participant in the broader marketplace known as Web analytics. As companies became more and more dependent upon the Web for communications and customer support, the need to understand what customers were looking at and, even more importantly, what website characteristics influenced customer decisions became critical. Web analytics, broadly defined, studied Web traffic in an attempt to understand website effectiveness.

According to a 2011 report by the Gartner Group (Gassman 2011), the Web analytics marketplace could be divided into three broad segments, only one of which seems likely to offer much revenue generation potential:

- **Low end**, where basic traffic is measured but little sophisticated analysis is performed. The free standard version of Google Analytics, and various open source tools serve the needs of this market segment.
- **Middle**, where companies have attempted to interpret basic measures in terms of their business

value. Typically, free tools are used to gather these metrics. In some cases, however, organizations in this category acquire additional applications, or may move to the high-end tier once such value has been demonstrated.

- **High end**, where businesses make a systematic study of website value and are willing to invest in such value once it can be measured. Gassman provides examples that include:
    - Automated processes to optimize online campaigns and behavior on the website.
    - Ability to target landing page content to suit the context of visitors and to customize content to a visitors' behavior throughout their visits.
    - Ability to mash Web analytics data with other data, including transaction, master customer and third-party data and social media metrics.

According to the Gartner study, four companies dominated the high-end customer segment. These companies were:

1. *Adobe*: Reporting about 6000 customers accounting for almost $500 million in revenue. The company had, through internal development and acquisition, acquired a full suite of products including tools for social analytics, one of the most rapidly growing areas of interest.
2. *Google*: By far the largest competitor, with a reported installed base of over 200,000 customers—most of whom used the free standard version of Google Analytics. In 2011, it introduced a premium service with vastly expanded capabilities. According to Gassman, a subscription to the premium service typically cost $150,000/year.
3. *IBM*: Entered the Web analytics market in 2010 by acquiring two mid-size players with a combined customer base estimated to be somewhere around 3000 clients.
4. *Webtrends*: A private company reported to have around 3500 customers. It was also growing through acquisition of smaller companies.

All told, it seemed likely that the existing global Web analytics market was well in excess of $1 billion. It was also evident that all four of the largest participants were growing through aggressive acquisition of much smaller firms.

Along with SiteWit, there was a group of next-generation companies that targeted the high-end Web analytic space, but focused on managing and/or optimizing online campaigns on platforms such as Google Adwords, Bing adCenter, and Facebook. These companies provided sophisticated services in areas not yet fully served by large competitors at different price points. Some firms targeted small-to-medium sized businesses or advertising agencies, while others focused at the top end. These companies included Acquisio (www.acquisio.com), Clickable (www.clickable.com), Kenshoo (www.kenshoo.com), Lexity (lexity.com), Marin Software (www.marinsoftware.com), WordStream (www.wordstream.com), and a few others.

The Firm

SiteWit, headquartered in Tampa, was at the leading edge of the market for online predictive analytics and paid search optimization software. SiteWit provided an online marketing optimization and predictive analytics platform that allowed online marketers to optimize their Google AdWords and Bing adCenter campaigns, with Facebook soon to follow. Pay-per-click campaign management was available within the SiteWit.com software-as-a-service (SaaS) platform, along with predictive analytics that segmented and scored website traffic. The company offered a "freemium" model, with all website monitoring, traffic reports, and predictive analytics available at no cost. Website traffic monitoring relied on a comprehensive revenue attribution model that used first click, last click, and multi-click attribution to better understand how multiple visitor sessions affected purchasing and other e-commerce actions. Active campaign management was offered at a flat fee, rather than based on a percentage of ad spend.

The CEO

Ricardo Lasa was originally from Madrid, Spain. He grew up around businesses as his father Jose Luis Lasa built a large and successful real estate development firm. Lasa came to the United States to finish his undergraduate degree in MIS at the University of South Florida (USF), and went on to complete both a Masters degree in MIS and an MBA. He stays active in the technical community, serving on the Advisory Board of the Information Systems Department at USF, as well as in organizations such as the Tampa Bay Technology Forum (tbtf.org) and Tampa Bay WaVE (tampabaywave.org). He has been the CEO and founder of several other technology startups, including Web Piston (webpiston.com), a do-it-yourself website builder (the subject of another case (Gill and Lasa 2010)) and Rivergy, Inc. a leading Web developer in the Tampa Bay area. Ricardo Lasa gained critical experience in understanding the software-as-a-service (SaaS) business model through Web Piston, selling thousands of websites via online sales. Web Piston relied heavily on online advertising, running campaigns around the world. It was this experience that led him to co-found SiteWit Corporation, using the early version of the service to optimize his own Web Piston campaigns. Ricardo Lasa is a CEO with a lot of technical depth and he helped develop many of the core SiteWit components along with a small group of programmers that have worked together for a long time (building both Rivergy and Web Piston).

The Software Architecture

SiteWit was designed and developed for cloud computing from the outset. SiteWit ran on Amazon Web Services (AWS), though other cloud vendors had also been used during development. Cloud computing offered a flexible and cost effective infrastructure for the data intensive Web analytic tasks that underpinned SiteWit functionality. The high-level architecture (shown in Figure 3) was specifically designed with high availability and scalability in mind.

Availability

The cloud computing environment offered some tremendous advantages with respect to cost and on-demand resources, but virtualized servers brought challenges related to somewhat unpredictable I/O latencies and discrete failures. To meet availability goals, SiteWit layered more traditional database recovery and availability strategies on top of the cloud-based components. In particular, the core relational database servers were mirrored with failover capabilities. These databases were also used to refresh the development environment with real data. Finally, the lowest level Web log data was continually archived to a separate database instance. All other services were provided using easily replicated commodity servers for redundancy and performance gains through coarse-grained parallelism. The cloud computing infrastructure made it easy to provision new servers to meet changing demands.

Scalability

Given the data intensive nature of the SiteWit feature set, one of the most important aspects of the architecture was scalability. Careful consideration was given to the location of computationally demanding tasks, leaving some within the core database servers and locating others on commodity application servers. SiteWit used several groups of such servers for data collection, session processing (on application servers), and reporting. Dedicated Web servers that record the low-level page hit data handled data collection. Most importantly, the very intensive processes used to group sessions into threads for visitor histories, compute the many session attributes for predictive modeling, and handle cost and revenue attribution all took place on a collection of dedicated application servers that could be easily expanded to meet escalating demands. SiteWit maintained three attribution models: first click, last click, and multi-click (even across funnel) attribution. An extensive process status and queuing system was used to distribute tasks across this server group. Another demanding task was creating the aggregated summary data used for reporting. Again, a collection of reporting servers was used, incrementally pulling low-level data and producing the various aggregations necessary for presentation via SiteWit Web servers. The core database servers coordinated the activities of these satellite server groups and handled specialized tasks, such as training predictive models for visitor scoring and segmentation.

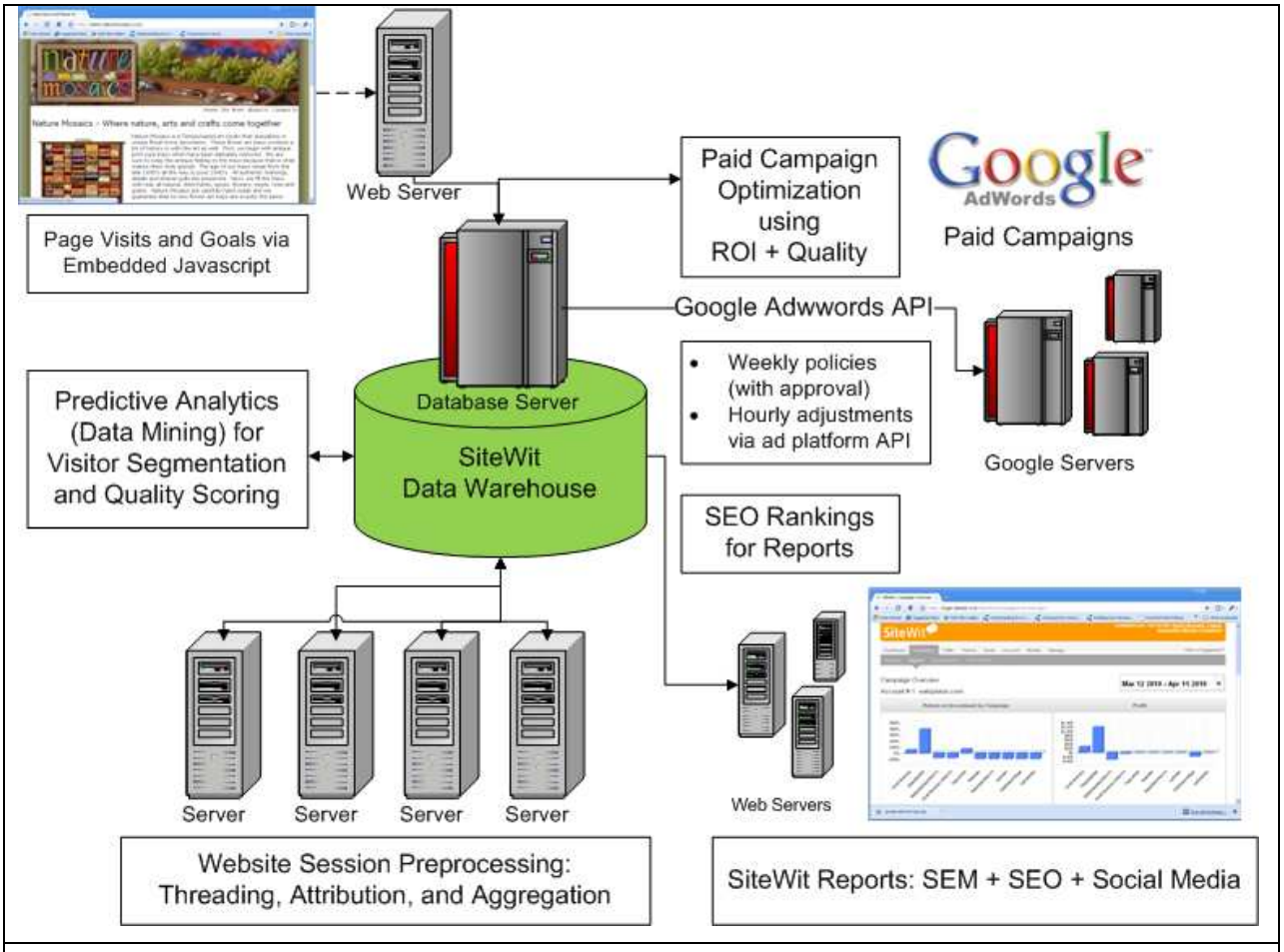

Figure 3. The SiteWit Software Architecture

NoSQL at SiteWit

The technical team at SiteWit had already looked at some alternative NoSQL databases, even running some preliminary tests. In addition, two corporate partners had gained some experience with specific systems. One partner had already made the leap, building their system using Citrusleaf (a NoSQL platform). Their products had extremely high performance demands and their experience was very positive. The other partner had completed some experiments with NoSQL systems, such as MongoDB. In fact, SiteWit engineers had joined their staff at a recent MongoDB conference.

Chris Lord, CTO and Matt Munday, Chief Software Architect (CSA) both attended a MongoDB conference and were evaluating other NoSQL technologies as well. While there was a lot of positive hype surrounding many of the platforms, all technologies make tradeoffs and have limitations. In this arena, it is critical to understand the advantages and disadvantages of these new platforms, applying the appropriate tool to an appropriate task. Matt Munday (always the skeptic) had done some digging around looking for some outside opinions on NoSQL databases and MongoDB in particular. He had recently found a blog post that provided a fairly in-depth review of MongoDB (excerpted here).

To be fair, it must be acknowledged that MongoDB is popular, and that there are valid reasons for its popularity.

** It is remarkably easy to get running.*

** Schema-free models that map to JSON-like structures have great appeal to developers (they fit our brains), and a developer is almost always the individual who makes the platform decisions when a project is in its infancy.*

** Maturity and robustness, track record, tested real-world use cases, etc, are typically more important to sysadmin types or operations specialists, who often inherit the platform long after the initial decisions are made.*

** Its single-system, low concurrency read performance benchmarks are impressive, and for the inexperienced evaluator, this is often The Most Important Thing.*

However, the post went on to warn about some serious problems.

But if you're intending to really run a large scale system on Mongo, one that a business might depend on, simply put:

1. MongoDB issues writes in unsafe ways *by default* in order to win benchmarks

2. MongoDB can lose data in many startling ways

3. MongoDB requires a global write lock to issue any write

Under a write-heavy load, this will kill you.

4. MongoDB's sharding doesn't work that well under load

5. mongos is unreliable

6. MongoDB actually once deleted the entire dataset

7. Things were shipped that should have never been shipped

8. Replication was lackluster on busy servers

Please take this warning seriously.

In a subsequent design discussion, Matt Munday made a point of saying that "he would not want his bank to be running on a NoSQL database." In truth, he was reluctant to move any of the SiteWit billing processes to a NoSQL platform. So, there are definitely strengths and weaknesses to consider. It is in this context that Lasa and his team considered the leap to NoSQL technology.

The Decision

The decision faced by the SiteWit team would affect all of the products and services offered by the company. At the core, SiteWit was an analytics company, capturing every click on a website and processing them into visitor sessions at various levels of abstraction. This detailed data was then used in analyses that adjust bids for search terms, develop suggestions for optimizing advertising campaigns, and create predictive models for scoring or segmenting website visitors.

Ricardo Lasa often repeated, especially within earshot of the technical staff, that the "biggest risk facing the company is the engine." He was basically emphasizing that the biggest risks in failing to deliver quality services, as well as scaling for future growth, were related to the core data collection and processing infrastructure. Among these risks, a few specific threats stood out.

1. The key to providing high quality campaign optimization services and predictive models was having the fine-grained data necessary for analyses. Whenever the data collection and processing systems failed, most other services also need to be paused (directly affecting the customer experience).
2. Even when the processing services were running, most of the customer experience was driven by the availability of insightful reports that were challenging to compute. Slow downs in the data infrastructure meant delays in delivering reports and a poor customer experience.
3. An important competitive advantage for SiteWit was the highly automated implementation of even complex tasks, such as predictive modeling. This enabled the delivery of sophisticated

services at very affordable prices. Any issues that needed to be resolved through highly (or even moderately) skilled labor cut deeply into profits.

4. The key to long-term success for SiteWit was a very large customer base with low prices and low costs. This meant that the core data intensive tasks needed to grow much larger in scale. Internet giants, including Google, Amazon, and Facebook had clearly embraced big data and blazed a trail to successful Internet-scale performance.

The Choices

So, with these factors in mind, Lasa and his team faced an interesting set of choices. Broadly speaking the choices fell into four categories.

1. **Do nothing.** SiteWit Corporation was a lean startup with limited resources. As a startup, simply surviving the initial growth stage and establishing product-market fit with early adopters had been a challenge. Do we really need to add new technologies and more uncertainty at this stage? Perhaps the most prudent course was to focus on refining the products and gaining valuable early customers before worrying about scaling (especially with a largely unknown and unproven technology). After all, SiteWit did not have the resources of a Google to spend whatever it might take to overcome the inevitable difficulties in such a technological endeavor.
2. **Proceed cautiously with NoSQL technology through limited experiments.** Even though SiteWit was an early stage company, it boasted a culture of research since the products rested on a foundation of big data, analytics, and machine learning. In addition, a data-driven approach was taken in the development process as part of the lean startup philosophy, including "innovation accounting" and the learning cycle (Ries 2011). It might be reasonable to pick some component that could be implemented using NoSQL technology to gain experience and validate the technology (and better understand the specific benefits within the SiteWit context). In fact, it might also be possible to build a parallel implementation of a component that would enable a very realistic benchmarking comparison.
3. **Develop a new product, alone or through a partnership, that makes use of NoSQL technologies.** As it turned out, a couple of existing SiteWit partners were experimenting with or already resting firmly on NoSQL technologies. In some cases, the technology was a bit different than what would be used within SiteWit. Nonetheless, the technologies were certainly close enough to shed light on any potential benefits. One strategy might be to identify a partnership opportunity that would make use of a NoSQL database, learning both from the partner and the experience of developing a real system (with shared value). Of course, this option would require careful collaboration and entail some loss of control. As it happened, there were a few such opportunities on the radar screen.
4. **Take a leap of faith.** Again, SiteWit was an early stage company facing plenty of risk factors. Adding a few more for an important competitive advantage could be a reasonable tradeoff. Since development resources were limited, it would pay to put the best engineers on the critical NoSQL database project that was the heartbeat of all product offerings. Splitting the attention of key technical staff would likely be a recipe for disaster, with the possibility of poorly implementing both SQL and NoSQL databases. In addition, there were several analytics-oriented startups that had successfully implemented NoSQL platforms and had grown quickly with the confidence to scale. As the Nike slogan goes: just do it.

Acknowledgements

The authors gratefully acknowledge the helpful guidance of T. Grandon Gill in preparing this case, as well as the opportunity to participate in his NSF-sponsored STEM case project.

References

Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R.E. 2006. "Bigtable: A Distributed Storage System for Structured Data," in *Seventh Symposium on Operating System Design and Implementation (OSDI)*, Berkeley, CA, pp. 205-218.

Chodorow, K. and Dirolf, M. 2010. *MongoDB: The Definitive Guide*, O'Reilly Media, Inc.

Codd, E.F. 1970. "A Relational Model of Data for Large Shared Data Banks," *Communications of the ACM* (13:6), pp. 377-387.

Codd, E.F. 1982. "Relational Database: A Practical Foundation for Productivity." *Communications of the ACM* (25:2), pp. 109-117.

Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H., Puz, N., Weaver, D., and Yerneni, R. 2008. "PNUTS: Yahoo!'s Hosted Data Serving Platform," in *34th International Conference on Very Large Data Bases*, Auckland, New Zealand.

Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. 2010. "Benchmarking Cloud Serving Systems with YCSB," in *First ACM Symposium on Cloud Computing (SoCC)*, Indianapolis, IN.

Gassman, B. 2012. "Web Analytics Market Update, 2012," *The Gartner Group*, 17 November 2011, ID:G00224599.

Gelernter, D. and Carriero, N. 1992. "Coordination Languages and their Significance," *Communications of the ACM* (35:2), pp. 97-107.

Ghemawat, S., Gobioff, H., and Leung, S. 2003. "The Google File System," in *ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY.

Gilbert, S. and Lynch N. 2002. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services," *ACM SIGACT News* (33:2).

Gill, T.G., and Lasa, R. 2010. "Web Piston: Choosing a New Strategy," *ICIS 2010 Proceedings*, Paper 132, http://aisel.aisnet.org/icis2010_submissions/132.

Hurst, N. 2010. "Visual Guide to NoSQL Systems," retrieved from http://blog.nahurst.com/visual-guide-to-nosql-systems.

Prichett, D. 2008. "BASE: An ACID Alternative," *ACM Queue* (6:3), pp. 48-55.

Ries, E. 2011. *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*, Crown Publishing Group, Random House, Inc.