Distribution, Data, Deployment

Software Architecture Convergence in Big Data Systems

Ian Gorton and John Klein, Software Engineering Institute

// Big data systems present many challenges to software architects. In particular, distributed software architectures become tightly coupled to data and deployment architectures. This causes a consolidation of concerns; designs must be harmonized across these three architectures to satisfy quality requirements. //

THE EXPONENTIAL GROWTH of data over the last decade has fueled a new specialization for software technology: data-intensive, or big data, software systems.[1] Internet-born organizations such as Google and Amazon are on this revolution's cutting edge, collecting, managing, storing, and analyzing some of the largest data repositories ever constructed. Their pioneering efforts,[2,3] along with those of numerous other big data innovators, have provided a variety of open source and commercial data management technologies that let any organization construct and operate massively scalable, highly available data repositories.

Addressing the challenges of software for big data systems requires careful design tradeoffs spanning the distributed software, data, and deployment architectures. It also requires extending traditional software architecture design knowledge to account for the tight coupling that exists in scalable systems. Scale drives a consolidation of concerns, so that distribution, data, and deployment architectural qualities can no longer be effectively considered separately. To illustrate this, we'll use an example from our current research in healthcare informatics.

The Challenges of Big Data

Data-intensive systems have long been built on SQL database technology, which relies primarily on vertical scaling—faster processors and bigger disks—as workload or storage requirements increase. SQL databases' inherent vertical-scaling limitations[4] have led to new products that relax many core tenets of relational databases. Strictly defined normalized data models, strong data consistency guarantees, and the SQL standard have been replaced by schemaless and intentionally denormalized data models, weak consistency, and proprietary APIs that expose the underlying data management mechanisms to the programmer. These *NoSQL* products[4] typically are designed to scale horizontally across clusters of low-cost, moderate-performance servers. They achieve high performance, elastic storage capacity, and availability by partitioning and replicating datasets across a cluster. Prominent examples of NoSQL databases include Cassandra, Riak, and MongoDB (see the sidebar "NoSQL Databases").

Distributed databases have funda-

0740-7459/15/$31.00 © 2015 IEEE

NOSQL DATABASES

The rise of big data applications has caused a significant flux in database technologies. While mature relational database technologies continue to evolve, a spectrum of databases called *NoSQL* has emerged over the past decade. The relational model imposes a strict schema, which inhibits data evolution and causes difficulties scaling across clusters. In response, NoSQL databases have adopted simpler data models. Common features include schemaless records, allowing data models to evolve dynamically, and horizontal scaling, by sharding (partitioning and distributing) and replicating data collections across large clusters. Figure A illustrates the four most prominent data models, whose characteristics we summarize here. More comprehensive information is at http://nosql-database.org.

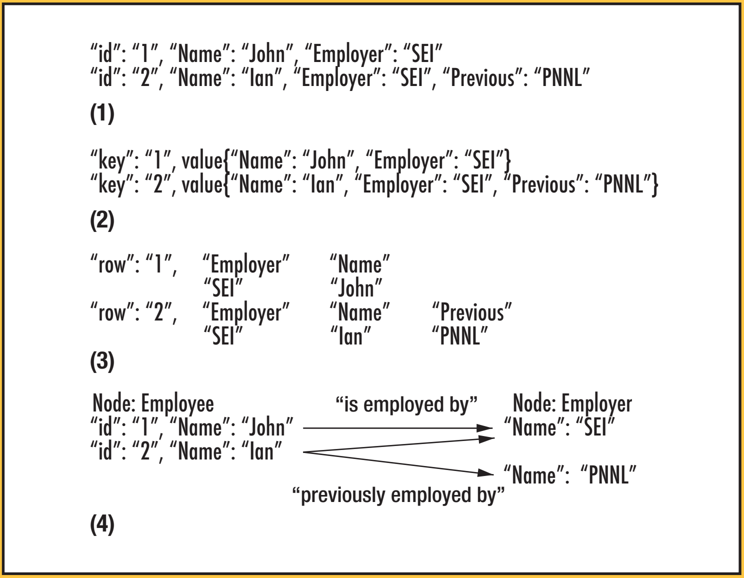
Document databases (see Figure A1) store collections of objects, typically encoded using JSON (JavaScript Object Notation) or XML. Documents have keys, and you can build secondary indexes on nonkey fields. Document formats are self-describing; a collection might include documents with different formats. Leading examples are MongoDB (www.mongodb.org) and CouchDB (http://couchdb.apache.org).

Key–value databases (see Figure A2) implement a distributed hash map. Records can be accessed primarily through key searches, and the value associated with each key is treated as opaque, requiring reader interpretation. This simple model facilitates sharding and replication to create highly scalable and available systems. Examples are Riak (http://riak.basho.com) and DynamoDB (http://aws.amazon.com/dynamodb).

Column-oriented databases (see Figure A3) extend the key–value model by organizing keyed records as a collection of columns, where a column is a key–value pair. The key becomes the column name; the value can be an arbitrary data type such as a JSON document or binary image. A collection can contain records with different numbers of columns. Examples are HBase (http://hbase.apache.org) and Cassandra (https://cassandra.apache.org).

Graph databases (see Figure A4) organize data in a highly connected structure—typically, some form of directed graph. They can provide exceptional performance for problems involving graph traversals and subgraph matching. Because efficient graph partitioning is an NP-hard problem, these databases tend to be less concerned with horizontal scaling and commonly offer ACID (atomicity, consistency, isolation, durability) transactions to provide strong consistency. Examples include Neo4j (www.neo4j.org) and GraphBase (http://graphbase.net).

NoSQL technologies have many implications for application design. Because there's no equivalent of SQL, each technology supports its own query mechanism. These mechanisms typically make the application programmer responsible for explicitly formulating query executions, rather than relying on query planners that execute queries based on declarative specifications. The programmer is also responsible for combining results from different data collections. This lack of the ability to perform JOINs forces extensive denormalization of data models so that JOIN-style queries can be efficiently executed by accessing a single data collection. When databases are sharded and replicated, the programmer also must manage consistency when concurrent updates occur and must design applications to tolerate stale data due to latency in update replication.



"id": "1", "Name": "John", "Employer": "SEI"
"id": "2", "Name": "Ian", "Employer": "SEI", "Previous": "PNNL"
(1)
"key": "1", value{"Name": "John", "Employer": "SEI"}
"key": "2", value{"Name": "Ian", "Employer": "SEI", "Previous": "PNNL"}
(2)
"row": "1", "Employer" "SEI" "Name" "John"
"row": "2", "Employer" "SEI" "Name" "Ian" "Previous" "PNNL"
(3)
Node: Employee
"id": "1", "Name": "John"
"id": "2", "Name": "Ian"
"is employed by"
Node: Employer
"Name": "SEI"
"Name": "PNNL"
"previously employed by"
(4)

FIGURE A. Four major NoSQL data models. (1) A document store. (2) A key–value store. (3) A column store. (4) A graph store.

mental quality constraints, defined by Eric Brewer's CAP (consistency, availability, partition tolerance) theorem.[5] When a network partition occurs (causing an arbitrary message loss between cluster nodes), a system must trade consistency (all readers see the same data) against availability (every request receives a success or failure response). Daniel Abadi's PACELC provides a practical interpretation of this theorem.[6] If a partition (P) occurs, a system must trade availability (A) against consistency (C). Else (E), in the usual case of no partition, a system must trade latency (L) against consistency (C).

Additional design challenges for scalable data-intensive systems stem from the following three issues.

Big data systems must be able to sustain write-heavy workloads.

First, achieving high scalability and availability leads to highly distributed systems. Distribution occurs in all tiers, from webserver farms and caches to back-end storage.

Second, the abstraction of a single system image, with transactional writes and consistent reads using SQL-like query languages, is difficult to achieve at scale.[7] Applications must be aware of data replicas; handle inconsistencies from conflicting replica updates; and continue operation in spite of inevitable processor, network, and software failures.

Third, each NoSQL product embodies a specific set of quality attribute tradeoffs, especially in terms of performance, scalability, durability, and consistency. Architects must diligently evaluate candidate database technologies and select databases that can satisfy application requirements. This often leads to *polyglot persistence*—using different database technologies to store different datasets in a single system, to meet quality attribute requirements.[8]

Furthermore, as data volumes grow to petascale and beyond, the required hardware resources grow from hundreds to tens of thousands of servers. At this deployment scale, many widely used software architecture patterns are unsuitable. Architectural and algorithmic approaches that are sensitive to hardware resource use can significantly reduce overall costs. For more on this, see the sidebar "Why Scale Matters."

Big Data Application Characteristics

Big data applications are rapidly becoming pervasive across a range of business domains. One example domain in which big data analytics looms prominently on the horizon is aeronautics. Modern commercial airliners produce approximately 0.5 Tbytes of operational data per flight.[9] This data can be used to diagnose faults in real time, optimize fuel consumption, and predict maintenance needs. Airlines must build scalable systems to capture, manage, and analyze this data to improve reliability and reduce costs.

Another domain is healthcare. Big data analytics in US healthcare could save an estimated $450 billion.[10] Analysis of petabytes of data across patient populations, taken from diverse sources such as insurance payers, public health entities, and clinical studies, can reduce costs by improving patient outcomes. In addition, operational efficiencies can extract new insights for disease treatment and prevention.

Across these and many other domains, big data systems share four requirements that drive the design of suitable software solutions. Collectively, these requirements represent a significant departure from traditional business systems, which are relatively well constrained in terms of data growth, analytics, and scale.

First, from social media sites to high-resolution sensor data collection in the power grid, big data systems must be able to sustain write-heavy workloads.[1] Because writes are costlier than reads, systems can use data sharding (partitioning and distribution) to spread write operations across disks and can use replication to provide high availability. Sharding and replication introduce availability and consistency issues that the systems must address.

The second requirement is to deal with variable request loads. Business and government systems experience highly variable workloads for reasons including product promotions, emergencies, and statutory deadlines such as tax submissions. To avoid the costs of overprovisioning to handle these occasional spikes, cloud platforms are elastic, letting applications add processing capacity to share loads and release resources when loads drop. Effectively exploiting this deployment mechanism requires architectures with application-specific strategies to detect increased loads, rapidly add new resources, and release them as necessary.

WHY SCALE MATTERS

Scale has many implications for software architecture; here we look at several.

The first implication focuses on how scale changes our designs' problem space. Big data systems are inherently distributed. Their architectures must explicitly handle partial failures, communication latencies, concurrency, consistency, and replication. As systems grow to thousands of processing nodes and disks and become geographically distributed, these issues are exacerbated as the probability of a hardware failure increases. One study found that 8 percent of servers in a typical datacenter experience a hardware problem annually, with disk failure most common.[1] Applications must also deal with unpredictable communication latencies and network connection failures. Scalable applications must treat failures as common events that are handled gracefully to ensure uninterrupted operation.

To deal with these issues, resilient architectures must fulfill two requirements. First, they must replicate data to ensure availability in the case of a disk failure or network partition. Replicas must be kept strictly or eventually consistent, using either master–slave or multimaster protocols. The latter need mechanisms such as Lamport clocks[2] to resolve inconsistencies due to concurrent writes.

Second, architecture components must be stateless, replicated, and tolerant of failures of dependent services. For example, by using the Circuit Breaker pattern[3] and returning cached or default results whenever failures are detected, an architecture limits failures and allows time for recovery.

Another implication is economics based. At very large scales, small optimizations in resource use can lead to very large cost reductions in absolute terms. Big data systems can use many thousands of servers and disks. Whether these are capital purchases or rented from a service provider, they remain a major cost and hence a target for reduction. Elasticity can reduce resource use by dynamically deploying new servers as the load increases and releasing them as the load decreases. This requires servers that boot and initialize quickly and application-specific strategies to avoid premature resource release.

Other strategies target the development tool chain to maintain developer productivity while decreasing resource use. For example, Facebook built HipHop, a PHP-to-C++ transformation engine that reduced the CPU load for serving Web pages by 50 percent.[4] At the scale of Facebook's deployment, this creates significant operational-cost savings. Other targets for reduction are software license costs, which can be prohibitive at scale. This has led some organizations to create custom database and middleware technologies, many of which have been released as open source. Leading examples of technologies for big data systems are from Netflix (http://netflix.github.io) and LinkedIn (http://linkedin.github.io).

Other implications of scale include testing and fault diagnosis. Owing to these systems' deployment footprints and the massive datasets they manage, comprehensively validating code before deployment to production can be impossible. Canary testing and Netflix's Simian Army are examples of the state of the art in testing at scale.[5] When problems occur in production, advanced monitoring and logging are needed for rapid diagnosis. In large-scale systems, log collection and analysis itself quickly becomes a big data problem. Solutions must include a low-overhead, scalable logging infrastructure such as Blitz4j.[6]

References

1. K.V. Vishwanath and N. Nagappan, "Characterizing Cloud Computing Hardware Reliability," *Proc. 1st ACM Symp. Cloud Computing* (SoCC 10), 2010, pp. 193–204.
2. L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM*, vol. 21, no. 7, 1978, pp. 558–565.
3. M.T. Nygard, *Release It! Design and Deploy Production-Ready Software*, Pragmatic Bookshelf, 2007.
4. H. Zhao, "HipHop for PHP: Move Fast," blog, 2 Feb. 2010; https://developers.facebook.com/blog/post/2010/02/02/hiphop-for-php--move-fast.
5. B. Schmaus, "Deploying the Netflix API," blog, 14 Aug. 2013; http://techblog.netflix.com/2013/08/deploying-netflix-api.html.
6. K. Ranganathan, "Announcing Blitz4j—a Scalable Logging Framework," blog, 20 Nov. 2012; http://techblog.netflix.com/search/label/appender.

The third requirement is to support computation-intensive analytics. Most big data systems must support diverse query workloads, mixing requests that require rapid responses with long-running requests that perform complex analytics on significant portions of the data collection. This leads to software and data architectures explicitly structured to meet these varying latency demands. Netflix's Recommendations Engine is a pioneering example of how to design software and data architectures to partition simultaneously between low-latency requests and requests for advanced analytics on large data collections, to continually enhance personalized recommendations' quality.[11]

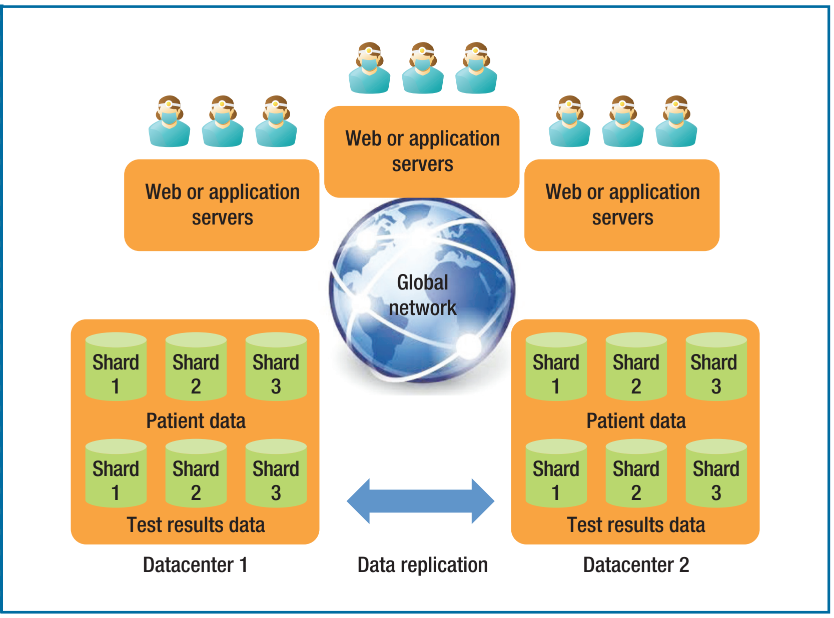


FIGURE 1. A MongoDB-based healthcare data management prototype. Geographically distributed datacenters increase availability and reduce latency for globally distributed users.

The fourth requirement is high availability. With thousands of nodes in a horizontally scaled deployment, hardware and network failures inevitably occur. So, distributed software and data architectures must be resilient. Common approaches for high availability include replicating data across geographical regions,[12] stateless services, and application-specific mechanisms to provide degraded service in the face of failures.

These requirements' solutions crosscut the distribution, data, and deployment architectures. For example, elasticity requires

- processing capacity that can be acquired from the execution platform on demand,
- policies and mechanisms to appropriately start and stop services as the application load varies, and
- a database architecture that can reliably satisfy queries under an increased load.

This coupling of architectures to satisfy a particular quality attribute is common in big data applications. It can be regarded as a tight coupling of the process, logical, and physical views in the 4 + 1 view model.[13]

An Example of the Consolidation of Concerns

At the Software Engineering Institute, we're helping to evolve a system to aggregate data from multiple petascale medical-record databases for clinical applications. To attain high scalability and availability at low cost, we're investigating using NoSQL databases for this aggregation. The design uses geographically distributed datacenters to increase availability and reduce latency for globally distributed users.

Consider the consistency requirements for two categories of data in this system: patient demographics (for example, name and insurance provider) and diagnostic-test results (for example, blood or imaging test results). Patient demographic records are updated infrequently. These updates must be immediately visible at the local site where the data was modified ("read your writes"), but a delay is acceptable before the update is visible at other sites ("eventual replica consistency"). In contrast, diagnostic-test results are updated more frequently, and changes must be immediately visible everywhere to support telemedicine and remote consultations with specialists ("strong replica consistency").

We're prototyping solutions using several NoSQL databases. We focus here on one prototype using MongoDB to illustrate the architecture drivers and design decisions. The design segments data across three shards and replicates data across two datacenters (see Figure 1).

MongoDB enforces a master–slave architecture; every data collection has a master replica that serves all write requests and propagates changes to other replicas. Clients can read from any replica, opening an inconsistency window between writes to the master and reads from other replicas.

MongoDB allows tradeoffs between consistency and latency through parameter options on each write and read. A write can be unacknowledged (no assurance of durability, low latency), durable on the master replica, or durable on the master and one or more replicas (consistent, high latency). A read can prefer the closest replica (potentially inconsistent, low latency), be restricted to the master replica (consistent, partition intolerant), or require most replicas to agree on the data value to be read (consistent, partition tolerant).

The application developer must choose appropriate write and read options to achieve the desired performance, consistency, and durability and must handle partition errors to achieve the desired availability. In our example, patient demographic data writes must be durable on the primary replica, but reads can be directed to the closest replica for low latency. This makes patient demographic reads insensitive to network partitions at the cost of potentially inconsistent responses.

In contrast, writes for diagnostic-test results must be durable on all replicas. Reads can be performed from the closest replica because the write ensures that all replicas are consistent. This means writes must handle failures caused by network partitions, whereas reads are insensitive to partitions.

Today, our healthcare informatics application runs atop an SQL database, which hides the physical data model and deployment topology from developers. SQL databases provide a single-system-image abstraction, which separates concerns between the application and database by hiding the details of data distribution across processors, storage, and networks behind a transactional read/write interface.[14] In shifting to a NoSQL environment, an application must directly handle the faults that will depend on the physical data distribution (sharding and replication) and the number of replica sites and servers. These low-level infrastructure concerns, traditionally hidden under the database interface, must now be explicitly handled in application logic.

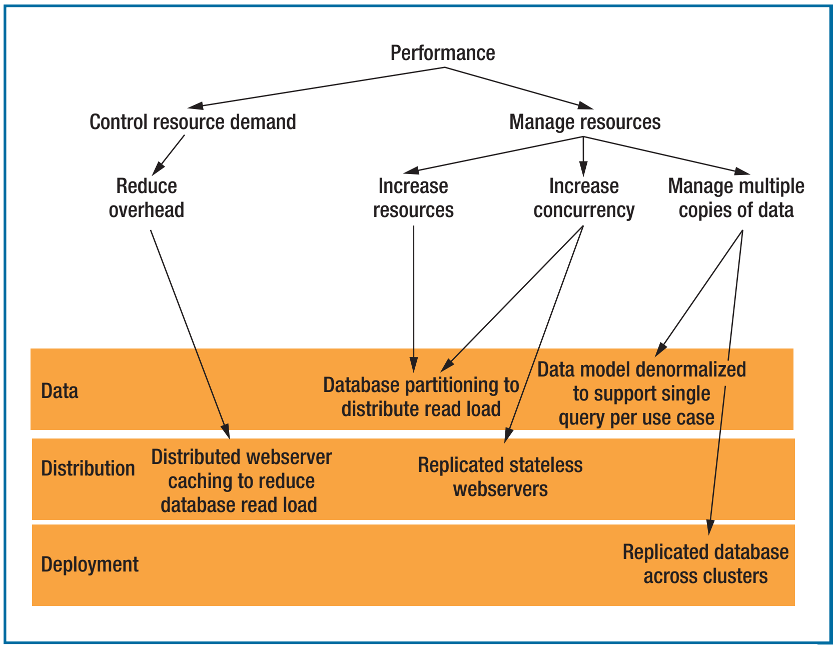

Performance
Control resource demand
Manage resources
Reduce overhead
Increase resources
Increase concurrency
Manage multiple copies of data
Data
Database partitioning to distribute read load
Data model denormalized to support single query per use case
Distribution
Distributed webserver caching to reduce database read load
Replicated stateless webservers
Deployment
Replicated database across clusters

FIGURE 2. Performance tactics for big data systems. The design decisions span the data, distribution, and deployment architectures.

Systematic Design Using Tactics

In designing an architecture to satisfy quality drivers such as those in this healthcare example, one proven approach is to systematically select and apply a sequence of architecture tactics.[15] Tactics are elemental design decisions, embodying architectural knowledge of how to satisfy one design concern of a quality attribute. Tactic catalogs enable reuse of this knowledge. However, existing catalogs don't contain tactics specific to big data systems.

Figures 2 and 3 extend the basic performance and availability tactics[15] to big data systems. Figure 4 defines scalability tactics, focusing on the design concern of increased workloads. Each figure shows how the design decisions span the data, distribution, and deployment architectures.

For example, achieving availability requires masking faults that inevitably occur in a distributed system. At the data level, replicating data items is an essential step to handle network partitions. When an application can't access any database partition, another tactic to enhance availability is to design a data model that can return meaningful default values without accessing the data. At the distributed-software layer, caching is a tactic to achieve the default-values tactic defined in the data model. At the deployment layer, an availability tactic is to geographically replicate the data and distributed application software layers to protect against power and network outages. Each of these tactics is necessary to handle the different types of faults that threaten availability. Their combined representation in Figure 3 provides architects with comprehensive guidance to achieve highly available systems.

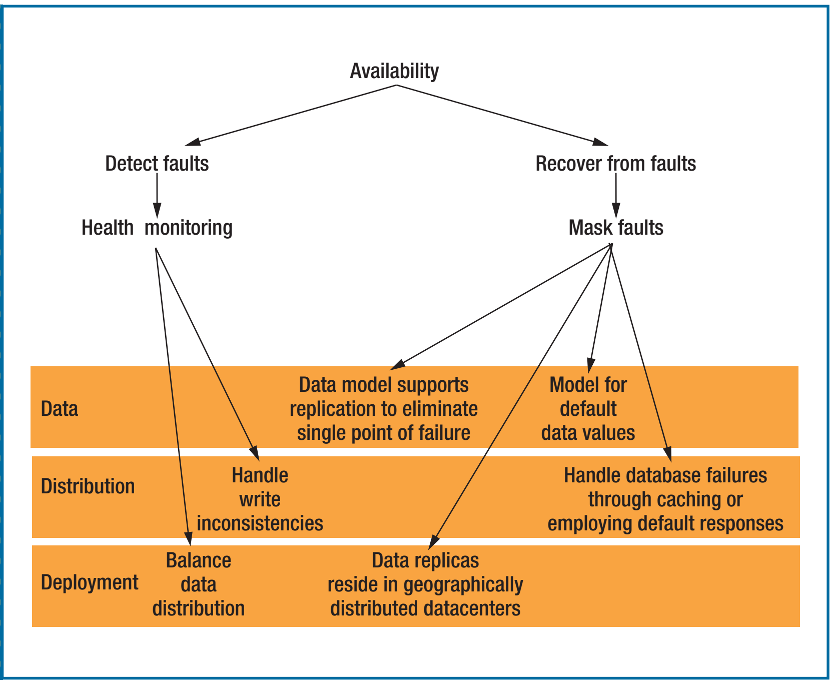

Availability
Detect faults
Recover from faults
Health monitoring
Mask faults
Data
Data model supports replication to eliminate single point of failure
Model for default data values
Distribution
Handle write inconsistencies
Handle database failures through caching or employing default responses
Deployment
Balance data distribution
Data replicas reside in geographically distributed datacenters

FIGURE 3. Availability tactics for big data systems. The tactics' combined representation provides architects with comprehensive guidance to achieve highly available systems.

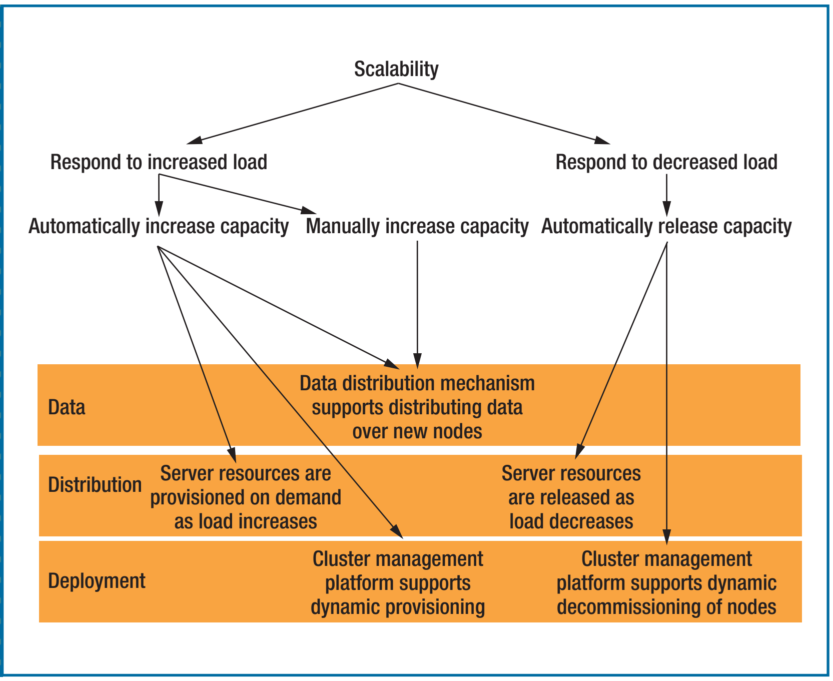

Scalability
Respond to increased load
Respond to decreased load
Automatically increase capacity
Manually increase capacity
Automatically release capacity
Data
Data distribution mechanism supports distributing data over new nodes
Distribution
Server resources are provisioned on demand as load increases
Server resources are released as load decreases
Deployment
Cluster management platform supports dynamic provisioning
Cluster management platform supports dynamic decommissioning of nodes

FIGURE 4. Scalability tactics for big data systems. These tactics focus on the design concern of increased workloads.

Big data applications are pushing the limits of software engineering on multiple horizons. Successful solutions span the design of the data, distribution, and deployment architectures. The body of software architecture knowledge must evolve to capture this advanced design knowledge for big data systems.

This article is a first step on this path. Our research is proceeding in two complementary directions. First, we're expanding our collection of architecture tactics and encoding them in an environment that supports navigation between quality attributes and tactics, making crosscutting concerns for design choices explicit. Second, we're linking tactics to design solutions based on specific big data technologies, enabling architects to rapidly relate a particular technology's capabilities to a specific set of tactics.

Acknowledgments

This material is based on research funded and supported by the US Department of Defense under contract FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. References herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, doesn't necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute. This material has been approved for public release and unlimited distribution. DM-0000810.

References

1. D. Agrawal, S. Das, and A. El Abbadi, "Big Data and Cloud Computing: Current State and Future Opportunities," *Proc. 14th Int'l Conf. Extending Database Technology* (EDBT/ICDT 11), 2011, pp. 530–533.

2. W. Vogels, "Amazon DynamoDB—a Fast and Scalable NoSQL Database Service Designed for Internet Scale Applications," blog, 18 Jan. 2012; www.allthingsdistributed.com/2012/01/amazon-dynamodb.html.
3. F. Chang et al., "Bigtable: A Distributed Storage System for Structured Data," *ACM Trans. Computing Systems*, vol. 26, no. 2, 2008, article 4.
4. P.J. Sadalage and M. Fowler, *NoSQL Distilled*, Addison-Wesley Professional, 2012.
5. E. Brewer, "CAP Twelve Years Later: How the "Rules" Have Changed," *Computer*, vol. 45, no. 2, 2012, pp. 23–29.
6. D.J. Abadi, "Consistency Tradeoffs in Modern Distributed Database System Design: CAP Is Only Part of the Story," *Computer*, vol. 45, no. 2, 2012, pp. 37–42.
7. J. Shute et al., "F1: A Distributed SQL Database That Scales," *Proc. VLDB Endowment*, vol. 6, no. 11, 2013, pp. 1068–1079.
8. M. Fowler, "PolyglotPersistence," blog, 16 Nov. 2011; www.martinfowler.com/bliki/PolyglotPersistence.html.
9. M. Finnegan, "Boeing 787s to Create Half a Terabyte of Data per Flight, Says Virgin Atlantic," *Computerworld UK*, 6 Mar. 2013; www.computerworlduk.com/news/infrastructure/3433595/boeing-787s-to-create-half-a-terabyte-of-data-per-flight-says-virgin-atlantic.
10. B. Kayyali, D. Knott, and S. Van Kuiken, "The 'Big Data' Revolution in Healthcare: Accelerating Value and Innovation," McKinsey & Co., 2013; www.mckinsey.com/insights/health_systems_and_services/the_big data_revolution_in_us_health_care.
11. X. Amatriain and J. Basilico, "System Architectures for Personalization and Recommendation," blog, 27 Mar. 2013; http://techblog.netflix.com/2013/03/system-architectures-for.html.
12. M. Armbrust, "A View of Cloud Computing," *Comm. ACM*, vol. 53, no. 4, 2010, pp. 50–58.
13. P.B. Kruchten, "The 4 + 1 View Model of Architecture," *IEEE Software*, vol. 12, no. 6, 1995, pp. 42–50.
14. J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993.
15. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed., Addison-Wesley, 2012.

ABOUT THE AUTHORS

IAN GORTON is a senior member of the technical staff on the Carnegie Mellon Software Engineering Institute's Architecture Practices team, where he investigates issues related to software architecture at scale. This includes designing large-scale data management and analytics systems and understanding the inherent connections and tensions between software, data, and deployment architectures. Gorton received a PhD in computer science from Sheffield Hallam University. He's a senior member of the IEEE Computer Society. Contact him at igorton@sei.cmu.edu.

JOHN KLEIN is a senior member of the technical staff at the Carnegie Mellon Software Engineering Institute, where he does consulting and research on scalable software systems as a member of the Architecture Practices team. Klein received an ME in electrical engineering from Northeastern University. He's the secretary of the International Federation for Information Processing Working Group 2.10 on Software Architecture, a member of the IEEE Computer Society, and a senior member of ACM. Contact him at jklein@sei.cmu.edu.