Architecture Knowledge for Evaluating Scalable Databases

Ian Gorton, John Klein
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA
{igorton, jklein}@sei.cmu.edu

Albert Nurgaliev
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA, USA
anurgali@andrew.cmu.edu

***Abstract*—Designing massively scalable, highly available big data systems is an immense challenge for software architects. Big data applications require distributed systems design principles to create scalable solutions, and the selection and adoption of open source and commercial technologies that can provide the required quality attributes. In big data systems, the data management layer presents unique engineering problems, arising from the proliferation of new data models and distributed technologies for building scalable, available data stores. Architects must consequently compare candidate database technology features and select platforms that can satisfy application quality and cost requirements. In practice, the inevitable absence of up-to-date, reliable technology evaluation sources makes this comparison exercise a highly exploratory, unstructured task. To address these problems, we have created a detailed feature taxonomy that enables rigorous comparison and evaluation of distributed database platforms. The taxonomy captures the major architectural characteristics of distributed databases, including data model and query capabilities. In this paper we present the major elements of the feature taxonomy, and demonstrate its utility by populating the taxonomy for nine different database technologies. We also briefly describe QuABaseBD, a knowledge base that we have built to support the population and querying of database features by software architects. QuABaseBD links the taxonomy to general quality attribute scenarios and design tactics for big data systems. This creates a unique, dynamic knowledge resource for architects building big data systems.**

Keywords—scalable software systems, big data, software architecture knowledge base, feature taxonomy

I. Introduction

There has been no industry in the history of engineering that exhibits the rapid rate of change we see in software technologies. By their very nature, complex software products can be created and evolved much more quickly than physical products, which require redesign, retooling and manufacturing [1]. In contrast, the barriers to software product evolution are no greater than incorporating new functionality in to code, testing, and releasing a new build for download on the Internet.

For software engineers building modern applications, there exists a dizzying number of potential *off-the-shelf* components that can be used as building blocks for substantial parts of a solution [2]. This makes component selection, composition, and validation a complex software engineering task that has received considerable attention in the literature (e.g. [3], [4], [5], [6]). While there is rarely a single 'right' answer when selecting a complex component for use in an application, selection of inappropriate components can be costly, reduce downstream productivity due to extensive rework, and even lead to project cancelation [7].

A contemporary application domain where there is particular difficulty in component selection is that of massively scalable, big data systems [8]. The exponential growth of data in the last decade has fueled rapid innovation in a range of components, including distributed caches, middleware and databases. Internet-born organizations such as Google and Amazon are at the cutting edge of this revolution, collecting, storing, and analyzing the largest data repositories ever constructed. Their pioneering efforts, for example [9] and [10], along with those of numerous other big data innovators, have created a variety of open source and commercial technologies for organizations to exploit in constructing massively scalable, highly available data repositories.

This technological revolution has instigated a major shift in database platforms for building scalable systems. No longer are relational databases the *de facto* standard for building data repositories. Highly distributed, scalable "NoSQL" databases [11] have emerged, which eschew strictly-defined normalized data models, strong data consistency guarantees, and SQL queries. These features are replaced with schema-less data models, weak consistency guarantees, and proprietary APIs that expose the underlying data management mechanisms to the application programmer. Prominent examples of NoSQL databases include Cassandra, Riak, neo4j and MongoDB.

NoSQL databases achieve scalability through horizontally distributing data. In this context, distributed databases have fundamental quality constraints, as defined by Brewer's CAP Theorem [12]. When a network partition occurs ("P"- arbitrary message loss between nodes in the cluster), a system must trade consistency ("C" - all readers see the same data) against availability ("A" - every request receives a success/failure response).

The implications of the CAP theorem are profound for architects. To achieve high levels of scalability and availability, distribution must be introduced in all system layers. Application designs must then be aware of data replicas, handle inconsistencies from conflicting replica updates, and continue degraded operation in spite of inevitable failures of processors, networks, and software. This leads to new and emerging design principles, patterns and tactics based on established distributed systems theory, which must be adopted to successfully build scalable, big data systems [13].

This confluence of rapidly evolving technologies and (re-) emerging design principles and patterns makes designing big data systems immensely challenging. Application architectures and design approaches must exploit the strengths of available components and compose these into deployable, extensible and scalable solutions.

This is especially challenging at the data storage layer. The multitude of competing NoSQL database technologies creates a complex and rapidly evolving design space for an architect to navigate. Architects must carefully compare candidate database technologies and features and select platforms that can satisfy application quality and cost requirements. In the inevitable absence of up-to-date, unbiased technology evaluations, this comparison exercise is in practice a highly exploratory, unstructured task that uses an Internet search engine as the primary information gathering and assessment tool.

In this paper we introduce a detailed feature taxonomy that can be used to systematically compare the capabilities of distributed database technologies. This taxonomy was derived from our experiences in evaluating databases for big data systems in a number of applications domains (e.g. [14]). The feature taxonomy describes both the core architectural mechanisms of distributed databases, and the major data access characteristics that pertain to the data architecture a given database supports. We describe our experience populating this taxonomy with the features of nine different databases to demonstrate its efficacy. We also describe a dynamic knowledge base we have built to semantically encode the feature taxonomy so that it can be queried and visualized.

The major contributions of this paper are:

- The first presentation of a detailed, software and data architecture-driven feature taxonomy for distributed database systems.
- A demonstration of the efficacy of the taxonomy through its population with the features from nine different database technologies.
- A description of the semantic encoding and representation of the feature taxonomy to support efficient knowledge capture, query and visualization.

II. Related Work

Our work builds upon and extends established work in software architecture knowledge management [15]. Early research in this area includes Kruchten [16], which introduced an ontology describing architectural design decisions for software systems. The ontology can be used to capture project-specific design decisions, their attributes, and relationships to create a graph of design decisions and their interdependencies. Our feature taxonomy also describes a graph of related design alternatives and relationships, but the knowledge relates to the class of distributed databases as opposed to a specific project or system. Hence, the knowledge has applicability to the broad class of big data software systems.

Other research has focused on using knowledge models to capture project-specific architectural decisions [17] and annotate design artefacts using ontologies [18][19][20]. Ontologies for describing general architecture knowledge have also been proposed. These including defining limited vocabularies [21], formal definitions of architecture styles [22], and supporting reuse of architecture documentation [23]. However, the inherent complexity of these approaches has severely limited adoption in practice.

Formal knowledge models for capturing architecture-related decisions also exist, for example [24], [25], [26], [27], [28], and [29]. Shahin describes a conceptual framework for these approaches that demonstrates significant overlap between the proposed concepts [30]. Our representation of the feature taxonomy takes a conceptually similar approach in that it semantically codifies a collection of general capabilities of big data systems, allowing an architect to explore the conceptual design space.

[32] presents a workflow systems feature taxonomy for Grid Computing. This work is similar to our taxonomy in intent, but the features it describes in the taxonomy are specific to workflow systems, and hence have virtually no overlap with big data management systems. In addition, this taxonomy is only described in an academic paper, and cannot be queried to dynamically present tailored content for technology evaluation.

A primary use case for our knowledge base that semantically encodes the feature taxonomy is to provide decision support for evaluating alternative database technologies. Earlier work has demonstrated how a similar approach based on feature categorization of technology platforms can be effective in practice for middleware technology evaluation [4]. Our work extends this approach by reifying technology-specific knowledge as semantic relationships that enable querying of the knowledge to rapidly answer project-specific evaluation questions.

III. Feature Taxonomy

Scalability in big data systems requires carefully harmonized data, software and deployment architectures [13]. In the data layer, scalability requires partitioning the data sets and their processing across multiple computing and storage nodes. This inherently creates a distributed software architecture in the data tier.

Contemporary database technologies adopt a variety of approaches to achieve scalability. These approaches can be distinguished by the data model that a database supports and the data distribution architecture it implements. Therefore, the selection of a specific database technology has a direct impact on the data and software architecture of an application.

Our feature taxonomy for distributed databases reflects these influences directly. It represents features in three categories related directly to the data architecture – namely *Data Model, Query Languages,* and *Consistency* – and four categories related directly to the software architecture – namely *Scalability, Data Distribution, Data Replication,* and *Security*. We decomposed each of these feature categories into a collection of specific features. Each feature has a set of allowed values representing the spectrum of design decisions that are taken in distributed databases. Depending on the database, some features may be assigned one or more of these values to fully characterize the database's capabilities.

In the following subsections we describe these feature categories and the spectrum of design decisions that are represented in our feature taxonomy. Space precludes a detailed description of each features. Instead we briefly describe the major classes of features and their implications on both the data and software architecture.

A. Data Model

The data model supported by a distributed database dictates both how application data can be organized, and to a large extent, how it can be queried. Our taxonomy, shown in Table 1, captures these data architecture issues, organizing features in three broad groups:

1. **Data Organization:** These features capture how a database platform enables data to be modeled, whether fixed schemas are required, and support for hierarchical data objects.
2. **Keys and Indexes:** Flexibility in data object key definition, including support for secondary and composite keys, can greatly influence application performance, scalability and modifiability. This collection of features describes how any given database support key definition.
3. **Query Approaches:** This collection of features describes the options available for querying a database, including key-based searching, text searching, and support for Map-Reduce based aggregation queries.

Table 1. Data Model Features

Feature	Allowed Values
Data Model	Column, Key-Value, Graph, Document, Object, Relational
Fixed Schema	Required, optional, none
Opaque Data Objects	Required, not required
Hierarchical Data Objects	Supported, not supported
Automatic Primary Key Allocation	Supported, not supported
Composite Keys	Supported, not supported
Secondary Indexes	Supported, not supported
Query by Key Range	Supported, not supported
Query by Partial Key	Supported, not supported
Query by Non-Key Value (Scan)	Supported, not supported
Map Reduce API	Builtin, integration with external framework, not supported
Indexed Text Search	Support in plugin (e.g. Solr), builtin proprietary, not supported

B. Query Languages

The query language features of a database directly affect application performance and scalability. For example, if a database does not return sorted result sets, the application itself must retrieve data from the database and perform the sort. For big data applications in which large results sets are common, this places a significant performance burden on an application, and uses resources (memory/CPU/network) that may be scarce under high loads.

Our feature taxonomy captures the broad query language characteristics, such declarative or imperative styles and languages supported, and the major detailed features that impart quality concerns. Table 2 illustrates these features. Note that for some features, for example *Languages Supported* and *Triggers*, multiple values may be assigned to the same feature for a database. This is a common characteristic that is seen across all categories in feature taxonomy.

Table 2. Query Language Features

Feature	Allowed Values
API-Based	Supported, Not Supported
Declarative	Supported, Not Supported
REST/HTTP-based	Supported, Not Supported
Languages supported	Java, C#, Python, C/C++, Perl, Ruby, Scala, Erlang, Javascript
Cursor-based queries	Supported, Not Supported
JOIN queries	Supported, Not Supported
Complex data types	Lists, maps, sets, nested structures, arrays, geospatial, none
Key matching options	Exact, partial match, wildcards, regular expressions
Sorting of query results	Ascending, descending, none
Triggers	Pre-commit, post-commit, none
Expire data values	Supported, Not Supported

C. Consistency

With the emergence of scalable database platforms, consistency has become a prominent quality of an application's data architecture. Transactional consistency properties that are standard in relational databases are rarely supported in NoSQL databases. Instead, a variety of approaches are supported for both transactional and replica consistency. This inevitably places a burden on the application to adopt designs that maintain strong data consistency or operate correctly with weaker consistency. A common design denormalizes data records so that a set of dependent updates can be performed in a single database operation. While this approach ensures consistency in the absence of ACID transactional semantics, denormalization also leads to increased data sizes due to duplication, and increased processing in order to keep duplicates consistent.

The features in Table 3 are grouped into those that support strong consistency, and those that support eventual (replica) consistency. Strong consistency features such as ACID and distributed transactions reduce application complexity at the cost of reduced scalability. Eventual consistency is a common alternative approach in scalable databases. Eventual consistency relies on storing replicas of every data object to distribute processing loads and provide high availability in the face of the database node failures. The taxonomy describes a range of features that constitute eventually consistent mechanisms, including conflict detection and resolution approaches.

D. Scalability

Evaluating qualities like performance and scalability in absolute terms require benchmarks and prototypes to establish empirical measures. However, the core architectural design decisions that underpin a database implementation greatly affect the scalability that an application can achieve. In our taxonomy, we capture some of these core scalability features, shown in Table 4.

Table 3. Consistency Features

Feature	Allowed Values
Object-level atomic updates	Supported, Multi-Value Concurrency Control, conflicts allowed
ACID transactions in a single database	Supported, lightweight transactions (e.g. test and set), not supported
Distributed ACID transactions	Supported, not supported
Durable writes	Supported, not supported
Quorum Reads/Writes (replica consistency)	In client API, in database configuration, in the datacenter configuration, not supported
Specify number of replicas to write to	In client API, in database configuration, not supported, not applicable – master-slave
Behavior when specified number or replica writes fails	Rollback at all replicas, No rollback, error returned, hinted handoffs, not supported
Writes configured to never fail	Supported, not supported
Specify number of replicas to read from	In client API, in database configuration, not supported, not applicable – master-slave
Read from master replica only	Not supported, in the client API, not applicable – peer-to-peer
Object level timestamps to detect conflicts	Supported, not applicable (single threaded), not applicable (master slave), not supported

Table 4. Scalability Features

Feature	Allowed Values
Scalable distribution architecture	Replicate entire database only; horizontal data partitioning; horizontal data partitioning and replication
Scaling out – adding data storage capacity	Automatic data rebalancing; manual database rebalancing; not applicable (single server only)
Request load balancing	HTTP-based load balancer required; client requests balanced across any coordinator; fixed connection to a request coordinator
Granularity of write locks	Locks on data object only; Table level locks; database level locks; no locks (single threaded); no locks (optimistic concurrency control); no locks (conflicts allowed)
Scalable request processing architecture	Fully distributed – any node can act as a coordinator; centralized coordinator but can be replicated; centralized coordinator (no replication); requires external oad balancer

Horizontal scaling spreads a data set across multiple nodes. In some databases, it is only possible to replicate complete copies of a database onto multiple nodes, which restricts scalability to the capacity of a single node - this is scaling up. Other databases support horizontal partitions, or sharding, to scale data on to multiple nodes.

Another key determinant of scalability is the approach to distributing client requests across database nodes. Bottlenecks in the request processing path for reads and writes can rapidly become inhibitors of scalability. These bottlenecks are typically request or transaction coordinators that cannot be distributed and replicated, or processes that store configuration state that must be accessed frequently during request processing.

E. Data Distribution

There are a number of software architecture alternatives that a database can adopt to achieve data distribution. These alternatives can greatly affect the quality attributes of the resulting system. To this end, the features in this category capture how a given database coordinates access to data that is distributed over deployment configurations ranging from single clusters to multiple geographically distributed data centers, shown in Table 5.

The mechanisms used to locate data and return results to requesting clients are an important aspect of distributed data access that affects performance, availability and scalability. Some databases provide a central coordinator that handles all requests and passes them on to other nodes where the data is located for processing. A more scalable solution is provided by databases that allow any database node to accept a request and act as the request coordinator.

Table 5. Data Distribution

Feature	Allowed Values
Data distribution architecture	Single database only; master-single slave; master-multiple slaves; multimaster
Data distribution method	User specified shard key; assigned key ranges to nodes; consistent hashing; not applicable (single server only)
Automatic data rebalancing	Failure triggered; new storage triggered; scheduled rebalancing; manual rebalancing; no applicable (single server only)
Physical data distribution	Single cluster; rack-aware on single cluster; multiple co-located clusters; multiple data centers
Distributed query architecture	Centralized process for key lookup; distributed process for key lookup; Direct replica connection only
Queries using non-shard key values	Secondary indexes; non-indexed (scan); not supported
Merging results from multiple shards	Random order; sorted order; paged from server; not supported

F. Data Replication

Data replication is necessary to achieve high availability in big data systems. This feature category is shown in Table 6. Replication can also enhance performance and scalability by distributing database read and write requests across replicas, with the inevitable trade-off of maintaining replica consistency. All databases that support replication adopt either a master-slave or peer-to-peer (multi-master) architecture, and typically allow a configurable number of replicas that can be geographically distributed across data centers.

Replication introduces the requirement on a database to handle replica failures. Various mechanisms, ranging from fully automated to administrative, are seen across database for replica failure and recovery. Recovery is complex, as it requires a replica to 'catch up' from its failed state and become a true replica of the current database state. This can be done by replaying transaction logs, or by simply copying the current state to the recovered replica.

G. Security

Security is necessary in the data tier of an application to ensure data integrity and prevent unauthorized access. This feature category is shown in Table 7. Our taxonomy captures the approaches supported by a database for authentication, which is often a key factor determining how a data platform can be integrated into an existing security domain. We also capture features such as *roles* that greatly ease the overheads and complexity of administering database security, and support for encryption – an important feature for applications requiring the highest levels of data security.

Table 6. Data Replication Features

Feature	Allowed Values
Replication Architecture	Master-slave; peer-to-peer
Replication for backup	Supported; not supported
Replication across data centers	Supported by data center aware features; Supported by standard replication mechanisms; Enterprise edition only
Replica writes	To master replica only; to any replica; to multiple replicas; to specified replica (configurable)
Replica reads	from master replica only; from any replica; from multiple replicas; from specified replica (configurable)
Read repair	Per query; background; not applicable
Automatic Replica Failure Detection	Supported; not supported
Automatic Failover	Supported; not supported
Automatic new master election after failure	Supported; not supported; not applicable
Replica recovery and synchronization	Performed by administrator; supported automatically; not supported

Table 7. Security Features

Feature	Allowed Values
Client authentication	Custom user/password; X509; LDAP; Kerberos; HTTPS
Server authentication	Shared keyfile; server credentials
Credential store	In database; external file
Role-based security	Supported; not supported
Security role options	Multiple roles per user; role inheritance; default roles; custom roles; not supported
Scope of rules	Cluster; database; collection; object; field
Database encryption	Supported; not supported
Logging	Configurable event logging; configurable log flush conditions; default logging only

IV. Knowledge Base Overview

QuABaseBD (Quality Attributes at Scale Knowledge Base, or QuABaseBD - pronounced *"k-baseBD'*).) is a linked collection of computer science and software engineering knowledge created specifically for designing big data systems with scalable database technologies. As depicted in Figure 1, QuABaseBD is presented to a user through a Web-based wiki interface. QuABaseBD is built upon the Semantic MediaWiki (SMW) platform (https://semantic-mediawiki.org/), which adds dynamic, semantic capabilities to the base MediaWiki implementation (used for Wikipedia).

In contrast to a typical wiki such as Wikipedia, the pages in QuABaseBD are dynamically generated from information that users enter into a variety of structured forms. This significantly simplifies content authoring for QuABaseBD and ensures internal consistency, as newly-added content is automatically included in summary pages and query results without needing to manually add links. Form-based data entry structures knowledge capture when populating the knowledge base, which ensures that the new content adheres to the underlying knowledge model. Hence the dynamic, structured nature of the QuABaseBD ensures it can consistently capture and render knowledge useful for software architects exploring the design space for big data systems.

QuABaseBD exploits these semantic capabilities to implement a model representing fundamental software architecture design knowledge for big data systems. The initial version of QuABaseBD populates this knowledge model specifically for designing the data layer of an application.

QuABaseBD links two distinct areas of knowledge through an underlying semantic model. These areas are:

1. **Software design principles for big data systems:** Knowledge pertaining to specific quality attribute scenarios and design tactics for big data systems.
2. **Database feature taxonomy:** Knowledge pertaining to the specific capabilities of NoSQL and NewSQL databases to support database evaluation and comparison, as described in the previous section.

In the following, we describe how we link these two areas in the QuABaseBD knowledge model, and how we use the features of the SMW platform to populate and query the feature taxonomy.

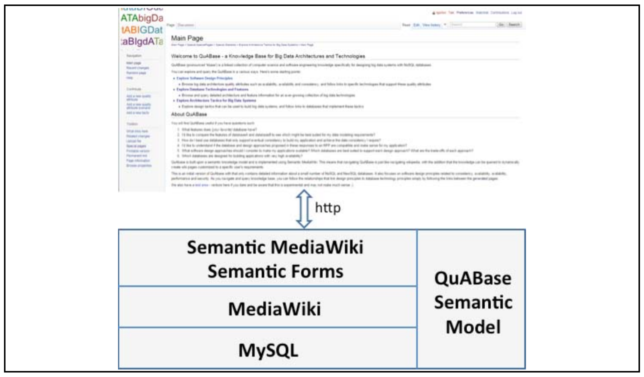


Figure 1. Conceptual Architecture of QuABaseBD

A. Semantic Knowledge Model

The SMW platform supports semantic annotation of information as *Categories* and *Properties*. These annotations can be applied in an *ad hoc* manner as markup in the wikitext allowing the semantic structure to emerge from the contributed content. In contrast, QuABaseBD takes a structured approach to knowledge representation, as discussed above. All content is created using a form and rendered using a template, and the set of forms and templates embodies the structure of the semantic knowledge model.

The knowledge model is split into two main sections. One section represents software architecture concepts related to quality attributes, quality attribute scenarios, and architecture tactics. This section of the knowledge model is not intended to be a complete representation of general software design knowledge, but instead, it represents a growing collection of concepts and properties needed to reason about big data systems design and database technology selection. The purposes of this section of the model are to support the definition of architecturally significant requirements, to identify the quality attribute tradeoffs that are inherent in distributed data-intensive systems, and to describe design tactics to achieve particular architecture requirements. The second section of the knowledge model represents the feature taxonomy described above.

The novelty of the QuABaseBD knowledge model is the linkage between the two sections through the relationship of an instance of a tactic to the instances of the features of a particular database that implement that tactic. This is shown in the extract of the knowledge model shown in Figure 2.

Here we see that a *Quality Attribute* is represented using a *General Scenario*. The general scenario includes only those stimuli, responses, and response measures that are relevant in big data systems, in contrast to the abstract general scenarios presented by Bass and colleagues [31]. A general scenario is a prototype that generates many *Quality Attribute Scenarios*, each of which combines a stimulus and response in the context of a big data system. A Quality Attribute Scenario covers a specific situation, and so we can identify the *Tactics* that can be employed to achieve the desired scenario response. Tactics represent tradeoffs – each tactic promotes at least one quality attribute, and may inhibit other quality attributes. Although not represented in Figure 2, the knowledge model also includes “anti-tactics”, representing design approaches that prevent the desired response from being achieved.

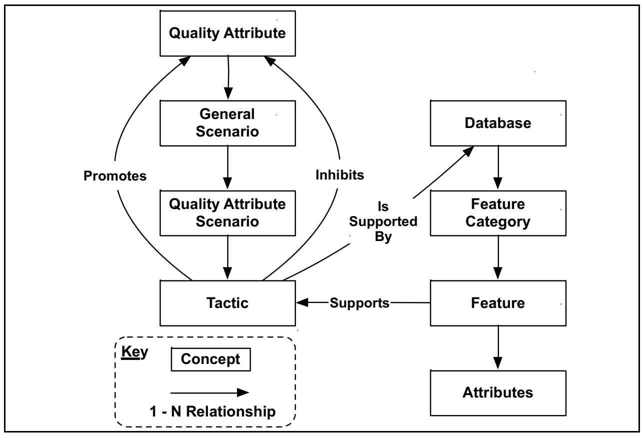
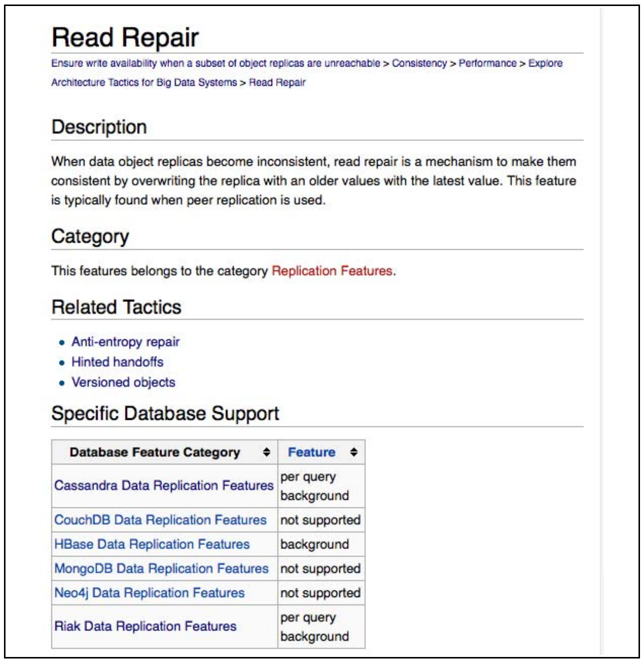


Figure 2. Extract from QuABaseBD Knowledge Model

Tactics also represent specific design decisions that can be realized by a *Database* implementation, so we can say that a database supports a collection of tactics. This support is provided by one or more *Features*, which are grouped into *Feature Categories*. Finally, a feature has one or more *Attributes*, which represent the allowable values for the feature.

This relationship between features and tactics allows an architect to reason architecture qualities. For example, an architect may reason about the need for certain database features in order to achieve a particular system quality, or how different implementations of a feature in different databases will affect system qualities.

An example of how QuABaseBD uses a template to link tactics to database implementations is shown in Figure 3. This illustrates the feature page for the *Read Repair* feature, which is in the feature category *Replication Features*. The related tactics that are supported by this feature are selected during the content creation process, and the table showing which databases implement the feature is generated dynamically by querying the knowledge base content.


Read Repair

Ensure write availability when a subset of object replicas are unreachable > Consistency > Performance > Explore Architecture Tactics for Big Data Systems > Read Repair

Description

When data object replicas become inconsistent, read repair is a mechanism to make them consistent by overwriting the replica with an older values with the latest value. This feature is typically found when peer replication is used.

Category

This features belongs to the category Replication Features.

Related Tactics

- Anti-entropy repair
- Hinted handoffs
- Versioned objects

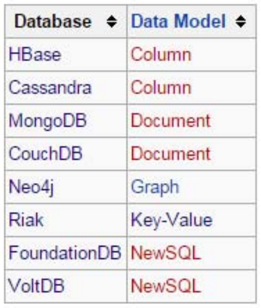
Specific Database Support

Database Feature Category	Feature
Cassandra Data Replication Features	per query background
CouchDB Data Replication Features	not supported
HBase Data Replication Features	background
MongoDB Data Replication Features	not supported
Neo4j Data Replication Features	not supported
Riak Data Replication Features	per query background

Figure 3. Linking from Tactics to Implementations and Features

B. *QuABaseBD Implementation of Feature Taxonomy*

From the main QuABaseBD page, users can choose to explore the knowledge base content for specific database technologies. Figure 4 shows an extract from the main database page. This table is a dynamically generated list (the result of the SMW query shown in Figure 5) of the databases for which QuABaseBD currently provides information.



Database	Data Model
HBase	Column
Cassandra	Column
MongoDB	Document
CouchDB	Document
Neo4j	Graph
Riak	Key-Value
FoundationDB	NewSQL
VoltDB	NewSQL

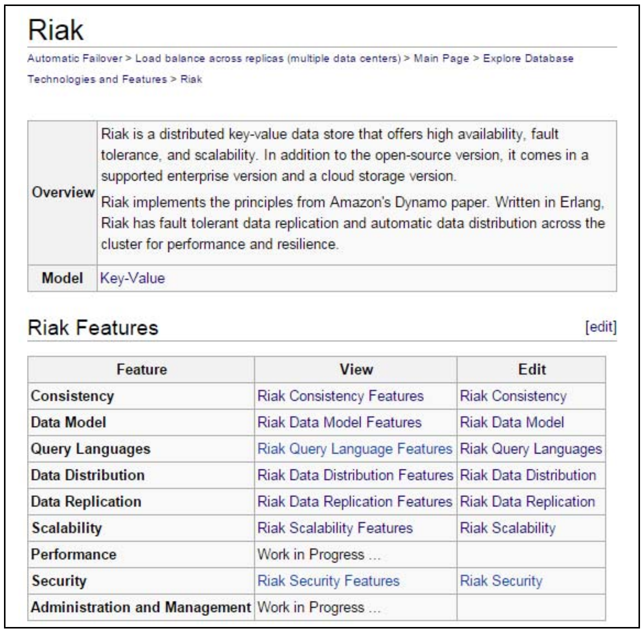
Figure 4. QuABaseBD Database Knowledge

```
{{#ask: [[Category:Database]]
|intro=Select any of the database below to get
information on their features and the tactics they
support
| mainlabel=Database
| ?Has DB Model=Data Model
| sort=Has DB Model
| order=asc
}}
```

Figure 5. Query to List Databases and Data Model Types

When a user navigates to the content for a specific database, for example Riak, they see a page that gives a brief overview of the database and a table listing the feature categories from the feature taxonomy, as shown in Figure 6.

Knowledge creators can edit the values for the feature taxonomy for each database. As discussed above, all content creation is done using SMW forms – clicking on an 'Edit' link displays a form for the associated feature category, as shown in Figure 7 for the data replication category.The forms render the elements of the feature taxonomy as fields organized in tabs, along with a valid set of values that a knowledge creator can select from for each feature. Using forms in this manner, the QuABaseBD implementation ensures a consistent set of values for each feature, thus greatly facilitating ease of comparison across databases.



Riak

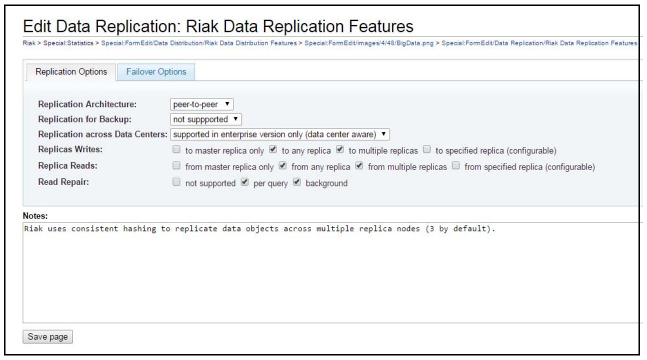
Automatic Failover > Load balance across replicas (multiple data centers) > Main Page > Explore Database Technologies and Features > Riak

Overview	Riak is a distributed key-value data store that offers high availability, fault tolerance, and scalability. In addition to the open-source version, it comes in a supported enterprise version and a cloud storage version. Riak implements the principles from Amazon's Dynamo paper. Written in Erlang, Riak has fault tolerant data replication and automatic data distribution across the cluster for performance and resilience.
Model	Key-Value

Riak Features [edit]

Feature	View	Edit
Consistency	Riak Consistency Features	Riak Consistency
Data Model	Riak Data Model Features	Riak Data Model
Query Languages	Riak Query Language Features	Riak Query Languages
Data Distribution	Riak Data Distribution Features	Riak Data Distribution
Data Replication	Riak Data Replication Features	Riak Data Replication
Scalability	Riak Scalability Features	Riak Scalability
Performance	Work in Progress ...	
Security	Riak Security Features	Riak Security
Administration and Management	Work in Progress ...	

Figure 6. QuABaseBD Knowledge for Riak



Edit Data Replication: Riak Data Replication Features

Riak > Special:Statistics > Special:FormEdit/Data Distribution/Riak Data Distribution Features > Special:FormEdit/images/4/48/BigData.png > Special:FormEdit/Data Replication/Riak Data Replication Features

Replication Architecture:	peer-to-peer
Replication for Backup:	not suppported
Replication across Data Centers:	supported in enterprise version only (data center aware)
Replicas Writes:	☐ to master replica only ✓ to any replica ✓ to multiple replicas ☐ to specified replica (configurable)
Replica Reads:	☐ from master replica only ✓ from any replica ✓ from multiple replicas ☐ from specified replica (configurable)
Read Repair:	☐ not supported ✓ per query ✓ background

Notes:

```
Riak uses consistent hashing to replicate data objects across multiple replica nodes (3 by default).
```

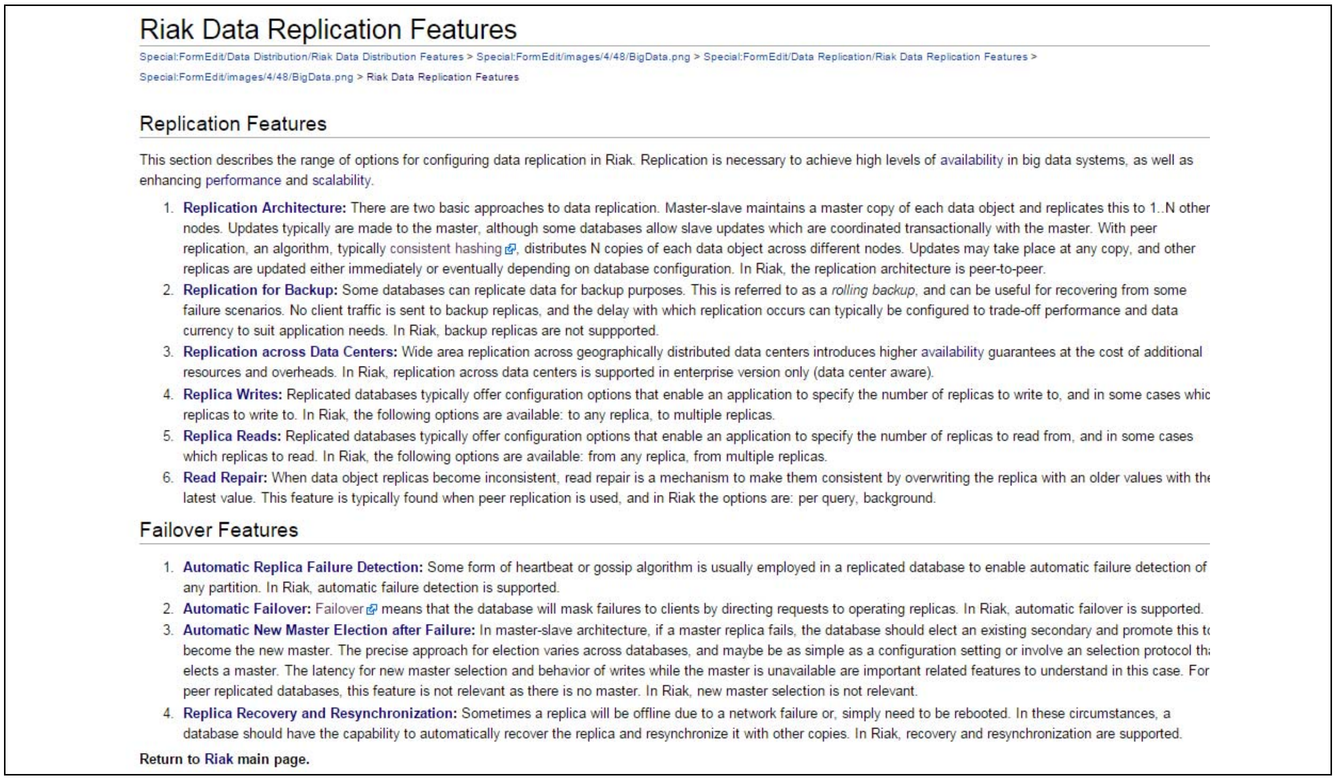
Figure 7. Populating the QuABaseBD Feature Taxonomy for Riak Replication

When a form is saved by a knowledge creator, an associated SMW template performs two actions:

1. It associates the selected feature values with a semantic property for each feature, creating a collection of RDF triples with the general form *{Feature, Has Value, Value}* to populate the semantic feature taxonomy
2. It generates a wiki page for knowledge consumers by substituting the selected feature values into a text template that describes each feature and the associated value for each database.

An example of the resulting generated wiki page is shown in Figure 8. Therefore, to a knowledge consumer, the mechanics of selecting alternate feature values is completely hidden. They simply see a description of each feature and how it is realized in the associated database.

A major advantage of associating each feature with a semantic property is that it facilitates fine grain searching across the different features for each database. Figure 9 illustrates the search facilities available to users for each feature category. By default, if no specific feature values are selected in the query form (top left), then all the databases in the QuABaseBD that have a completed feature page for that category are displayed (bottom right). The generated table facilitates a direct comparison of the databases in the QuABaseBD. The user can then choose specific values in the query form for the various features they are interested in and generate customized result sets to answer their specific questions..



Riak Data Replication Features

Special:FormEdit/Data Distribution/Riak Data Distribution Features > Special:FormEdit/images/4/48/BigData.png > Special:FormEdit/Data Replication/Riak Data Replication Features > Special:FormEdit/images/4/48/BigData.png > Riak Data Replication Features

Replication Features

This section describes the range of options for configuring data replication in Riak. Replication is necessary to achieve high levels of availability in big data systems, as well as enhancing performance and scalability.

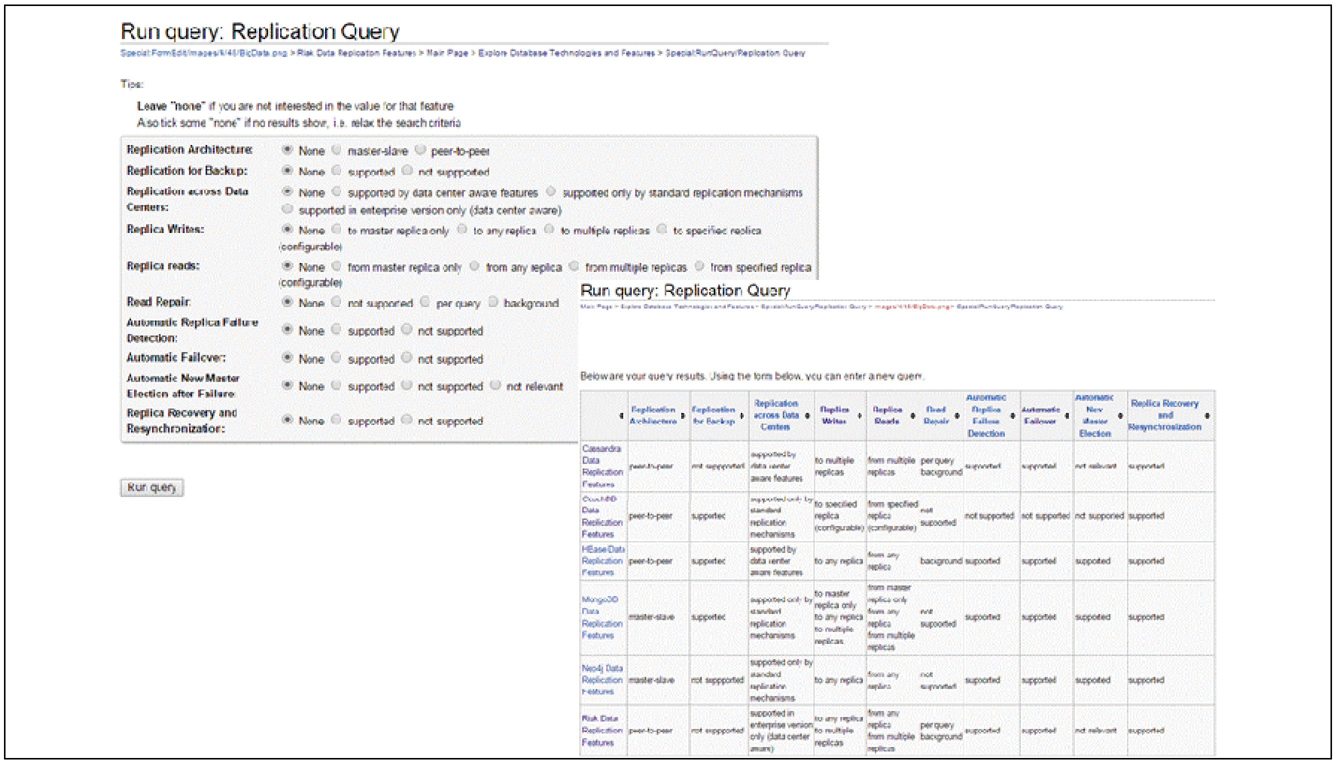
1. **Replication Architecture:** There are two basic approaches to data replication. Master-slave maintains a master copy of each data object and replicates this to 1..N other nodes. Updates typically are made to the master, although some databases allow slave updates which are coordinated transactionally with the master. With peer replication, an algorithm, typically consistent hashing, distributes N copies of each data object across different nodes. Updates may take place at any copy, and other replicas are updated either immediately or eventually depending on database configuration. In Riak, the replication architecture is peer-to-peer.
2. **Replication for Backup:** Some databases can replicate data for backup purposes. This is referred to as a *rolling backup*, and can be useful for recovering from some failure scenarios. No client traffic is sent to backup replicas, and the delay with which replication occurs can typically be configured to trade-off performance and data currency to suit application needs. In Riak, backup replicas are not suppported.
3. **Replication across Data Centers:** Wide area replication across geographically distributed data centers introduces higher availability guarantees at the cost of additional resources and overheads. In Riak, replication across data centers is supported in enterprise version only (data center aware).
4. **Replica Writes:** Replicated databases typically offer configuration options that enable an application to specify the number of replicas to write to, and in some cases whic replicas to write to. In Riak, the following options are available: to any replica, to multiple replicas.
5. **Replica Reads:** Replicated databases typically offer configuration options that enable an application to specify the number of replicas to read from, and in some cases which replicas to read. In Riak, the following options are available: from any replica, from multiple replicas.
6. **Read Repair:** When data object replicas become inconsistent, read repair is a mechanism to make them consistent by overwriting the replica with an older values with the latest value. This feature is typically found when peer replication is used, and in Riak the options are: per query, background.

Failover Features

1. **Automatic Replica Failure Detection:** Some form of heartbeat or gossip algorithm is usually employed in a replicated database to enable automatic failure detection of any partition. In Riak, automatic failure detection is supported.
2. **Automatic Failover:** Failover means that the database will mask failures to clients by directing requests to operating replicas. In Riak, automatic failover is supported.
3. **Automatic New Master Election after Failure:** In master-slave architecture, if a master replica fails, the database should elect an existing secondary and promote this to become the new master. The precise approach for election varies across databases, and maybe be as simple as a configuration setting or involve an selection protocol th: elects a master. The latency for new master selection and behavior of writes while the master is unavailable are important related features to understand in this case. For peer replicated databases, this feature is not relevant as there is no master. In Riak, new master selection is not relevant.
4. **Replica Recovery and Resynchronization:** Sometimes a replica will be offline due to a network failure or, simply need to be rebooted. In these circumstances, a database should have the capability to automatically recover the replica and resynchronize it with other copies. In Riak, recovery and resynchronization are supported.

Return to Riak main page.

Figure 8. Template-generated Wiki Page for Riak Replication



Run query: Replication Query

Special:FormEdit/images/4/48/BigData.png > Riak Data Replication Features > Main Page > Explore Database Technologies and Features > Special:RunQuery/Replication Query

Tips:

Leave "none" if you are not interested in the value for that feature
Also tick some "none" if no results show, i.e. relax the search criteria

Replication Architecture:	[✓ None] master-slave peer-to-peer
Replication for Backup:	[✓ None] supported not suppported
Replication across Data Centers:	[✓ None] supported by data center aware features supported only by standard replication mechanisms supported in enterprise version only (data center aware)
Replica Writes:	[✓ None] to master replica only to any replica to multiple replicas to specified replica (configurable)
Replica reads:	[✓ None] from master replica only from any replica from multiple replicas from specified replica (configurable)
Read Repair:	[✓ None] not supported per query background
Automatic Replica Failure Detection:	[✓ None] supported not supported
Automatic Failover:	[✓ None] supported not supported
Automatic New Master Election after Failure:	[✓ None] supported not supported not relevant
Replica Recovery and Resynchronization:	[✓ None] supported not supported

Run query: Replication Query

Main Page > Explore Database Technologies and Features > Special:RunQuery/Replication Query > images/4/48/BigData.png > Special:RunQuery/Replication Query

Below are your query results. Using the form below, you can enter a new query.

	Replication Architecture	Replication for Backup	Replication across Data Centers	Replica Writes	Replica Reads	Read Repair	Automatic Replica Failure Detection	Automatic Failover	Automatic New Master Election	Replica Recovery and Resynchronization
Cassandra Data Replication Features	peer-to-peer	not suppported	supported by data center aware features	to multiple replicas	from multiple replicas	per query background	supported	supported	not relevant	supported
CouchDB Data Replication Features	peer-to-peer	supported	supported only by standard replication mechanisms	to specified replica (configurable)	from specified replica (configurable)	not supported	not supported	not supported	not supported	supported
HBase Data Replication Features	peer-to-peer	supported	supported by data center aware features	to any replica	from any replica	background	supported	supported	supported	supported
MongoDB Data Replication Features	master-slave	supported	supported only by standard replication mechanisms	to master replica only to any replica to multiple replicas	from master replica only from any replica from multiple replicas	not supported	supported	supported	supported	supported
Neo4j Data Replication Features	master-slave	not suppported	supported only by standard replication mechanisms	to any replica	from any replica	not supported	supported	supported	supported	supported
Riak Data Replication Features	peer-to-peer	not suppported	supported in enterprise version only (data center aware)	to any replica to multiple replicas	from any replica from multiple replicas	per query background	supported	supported	not relevant	supported

Figure 9. Querying the Database Features

V. Demonstrating Feature Taxonomy Efficacy

The feature taxonomy and the initial implementation of QuABaseBD were co-developed during an evaluation of four NoSQL databases, namely MongoDB, Cassandra, Neo4j and Riak [14]. During this project, QuABaseBD was populated with information about those databases by the authors of this paper. The feature taxonomy and the allowed values were based on the capabilities of the four databases. As each of these databases represented a different NoSQL data model, we expected that the feature taxonomy we developed would be sufficiently general to facilitate differentiation amongst a much larger collection of technologies.

To assess the efficacy of the feature taxonomy, we further populated QuABaseBD with information about five other databases: Accumulo, Hbase, CouchDB, FoundationDB, and VoltDB. Graduate students helped perform this *content curation*[1]: Each contributor located and reviewed the product documentation for the database published on the Internet by the vendor or open source developer, and used the QuABaseBD SMW forms to enter the relevant information to populate the feature taxonomy.

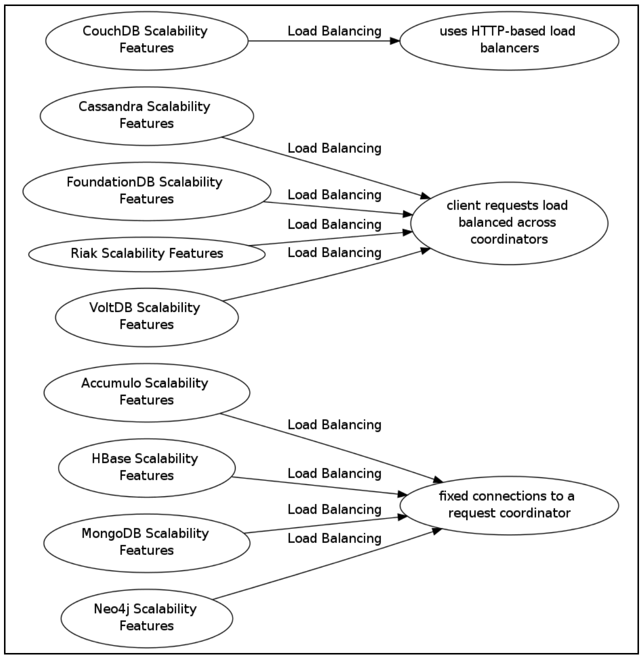

CouchDB Scalability Features
Load Balancing
uses HTTP-based load balancers
Cassandra Scalability Features
FoundationDB Scalability Features
Riak Scalability Features
VoltDB Scalability Features
client requests load balanced across coordinators
Accumulo Scalability Features
HBase Scalability Features
MongoDB Scalability Features
Neo4j Scalability Features
fixed connections to a request coordinator

Figure 10. Clustering of Implementations for the Load Balancing Feature

Throughout this curation process, we observed no additional features were needed to describe the functionality delivered by any of these databases, and curators were able to map all product features to one or more of the taxonomy features. For several features, we incorporated new "allowed values" to more precisely capture a particular database's capabilities. For example, CouchDB eschews object locking on updates in favor of Multi-Version Concurrency Control (MVCC). This approach was not supported in our initial set of databases, and hence we added MVCC as an allowed value to for the *Object Level Atomic Updates* feature. Additional allowed values for a number of features were also necessary to describe the capabilities of two so-called 'NewSQL' databases that were incorporated into QuABaseBD, namely VoltDB and FoundationDB. These databases are distributed, scalable implementations of relational databases, supporting large subsets of the standard SQL query language. Encouragingly, the feature taxonomy was sufficiently rich to enable us to capture the capabilities of these NewSQL databases in QuABaseBD.

Figure 10 is a simple visualization of the populated feature taxonomy that we can generate by querying QuABaseBD. It shows how each of the nine databases in the knowledge base relate to a value for the *Load Balancing* feature in the *Scalability* category. This visualization clearly shows how the databases cluster around the different feature values. Being able to convey such information visually is important as it makes it straightforward for architects to focus on features of interest and see how databases compare. As the number of databases in QuABaseBD grows, we will develop further visualizations to help convey the information is a concise and informative manner.

VI. Further Work and Conclusions

We are currently working towards finalizing QuABaseBD for public release. To this end we are testing the knowledge base functionality, and validating with experts on each database that our curated values for the database features are valid. We believe that as a curated scientific knowledge base, there are high expectations that the content in QuABaseBD is trustworthy at all times. To this end, we are working to design a systematic curation process where a small cohort of experts will be responsible for changes to the content. We anticipate that visitors to the knowledge base will be able to suggest changes through associated comments pages, and these proposals will be assessed for inclusion by the curators.

The SMW platform has provided a usable interface for these architects to derive answers to questions in a short amount of time. The platform's semantic metamodel, combined with the forms for content entry and templates for content rendering, allowed us to represent a novel domain knowledge model for big data architecture and technology in a form (a wiki) that users are familiar with.

After deployment, we will continue to expand the QuABaseBD content, and have identified several areas for future work. Manual creation and maintenance of content is inefficient, as the scope of the content is expanded to cover more of the database product landscape. Automation of these tasks is therefore needed, using technology such as machine learning to extract content from product documentation.

The terminology used in the database feature taxonomy needs further study. We chose terms that are abstract and general, rather than adopting implementation-specific terms. In some cases, adding alternative terminology for feature values may increase usability. We also anticipate expanding and

[1] http://en.wikipedia.org/wiki/Content_curation

restructuring the taxonomy as we receive feedback from the community.

We hope that QuABaseDB can become an enduring resource to support the design of big data systems. We also hope it will demonstrate the potential of curated, dynamic knowledge bases as sources of highly reliable scientific knowledge, as well as the basis for a next generation of software engineering decision support tools.

Acknowledgment

This material is based upon work funded and supported by the Department of Defense under Contract No. 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, trade mark, manufacturer, or otherwise, does not 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-0002076.

References

[1] M. Lehman. "Programs, life cycles, and laws of software evolution." *Proc. IEEE* 68.9 (1980): 1060-1076.

[2] J. Bosch. "From software product lines to software ecosystems." In *Proc. of the 13th Intl. Software Product Line Conf.*, 2009.

[3] A. S. Jadhav and R. M. Sonar. "Framework for evaluation and selection of the software packages: A hybrid knowledge based system approach," *J. Syst. Softw*. 84, 8 (August 2011), 1394-1407.

[4] A. Liu, I. Gorton. "Accelerating COTS middleware acquisition: the i-Mate process," *Software*, vol.20, no.2, pp.72,79, Mar/Apr 2003.

[5] J. Zahid, A. Sattar, and M. Faridi. "Unsolved Tricky Issues on COTS Selection and Evaluation." *Global Journal of Computer Science and Technology*, 12.10-D (2012).

[6] C. Becker, M. Kraxner, M. Plangg, et al. "Improving decision support for software component selection through systematic cross-referencing and analysis of multiple decision criteria". In *Proc. 46th Hawaii Intl. Conf. on System Sciences* (HICSS), pp. 1193-1202, 2013.

[7] N. Rozanski & E. Woods. *Software systems architecture: working with stakeholders using viewpoints and perspectives*. Addison-Wesley, 2011.

[8] D. Agrawal, S. Das, and A. El Abbadi, "Big data and cloud computing: current state and future opportunities," in *Proc. 14th Int. Conf. on Extending Database Technology (EDBT/ICDT '11)*, Uppsala, Sweden, 2011, pp. 530-533.

[9] G. DeCandia, D. Hastorun, M. Jampani, et al., "Dynamo: Amazon's Highly Available Key-value Store," in *Proc. 21st ACM SIGOPS Symp. on Operating Systems Principles (SOSP '07)*, Stevenson, WA, USA, 2007, pp. 205-220. doi: 10.1145/1294261.1294281.

[10] F. Chang, J. Dean, S. Ghemawat, et al., "Bigtable: A Distributed Storage System for Structured Data," *ACM Trans. Comput. Syst.,* vol. 26, no. 2, 2008.

[11] P. J. Sadalage and M. Fowler, *NoSQL Distilled.* Addison-Wesley Professional, 2012.

[12] E. Brewer, "CAP twelve years later: How the "rules" have changed," *Computer,* vol. 45, no. 2, pp. 23-29, February 2012.

[13] I. Gorton, I and J. Klein "Distribution, Data, Deployment: Software Architecture Convergence in Big Data Systems." *Software*, accepted (2014).

[14] J. Klein, I. Gorton, N. Ernst, et al., "Performance Evaluation of NoSQL Databases: A Case Study," in *Proc. of 1st Workshop on Performance Analysis of Big Data Systems (PABS 2015)*, Austin, TX, USA, 2015.

[15] M. A Babar, T. Dingsøyr, P. Lago, H. van Vliet. *Software architecture knowledge management: theory and practice*, Springer-Verlag. 2009

[16] P. Kruchten. "An ontology of architectural design decisions in software intensive systems." In *Proc. 2nd Groningen Workshop on Software Variability*. 2004.

[17] Art Akerman, and Jeff Tyree. "Using ontology to support development of software architectures," *IBM Systems Journal,* 45:4 (2006): 813-825.

[18] F. de Almeida, F. Ricardo, R. Borges, *et al*. "Using Ontologies to Add Semantics to a Software Engineering Environment." SEKE. 2005.

[19] H-J. Happel and S. Seedorf. "Applications of ontologies in software engineering." In *Proc. Workshop on Sematic Web Enabled Software Eng.* (SWESE) on the ISWC. 2006.

[20] H. Knublauch,. "Ontology-driven software development in the context of the semantic web: An example scenario with Protege/OWL." In *Proc. 1st Intl. Workshop on the Model-driven Semantic Web* (MDSW2004). Monterey, California, USA. 2004.

[21] L. T. Babu, et al. "ArchVoc-towards an ontology for software architecture." In *Proc. 29th Intl. Conf. on Software Eng. Workshops,* IEEE Computer Society, 2007.

[22] C. Pahl, S. Giesecke, and W. Hasselbring. "An ontology-based approach for modelling architectural styles." *Software Architecture* (2007): 60-75.

[23] C. A. Welty and D. A. Ferrucci. "A formal ontology for re-use of software architecture documents." In *Proc. 14th IEEE Intl. Conf. on Automated Software Eng.*, 1999.

[24] P. Kruchten, R. Capilla, and J.C. Dueas. "The decision view's role in software architecture practice." *Software*, 26:2 (2009): 36-42.

[25] P. Kruchten, P. Lago and H. van Vliet, "Building up and Reasoning about Architectural Knowledge". In *Proc. 2nd International Conference on the Quality of Software Architectures* (QoSA), pp. 39-47, 2006.

[26] J.S. van der Ven, A. Jansen, J. Nijhuis, et al, "Design decisions: The Bridge between Rationale and Architecture", In *Rationale Management in Software Engineering*, A.H. Dutoit, et al., eds, Springer, pp. 329-346, 2006.

[27] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions", In *Proc. 5th Working IEEE/IFIP Conf. on Software Arch.* (WICSA), pp. 109-120, 2005.

[28] R.C. de Boer, R. Farenhorst, P. Lago, et al, "Architectural Knowledge: Getting to the Core". In *Proc. 3rd Intl. Conf. on the Quality of Software Architectures* (QoSA), pp. 197-214, 2007.

[29] A. Jansen, J.S. van der Ven, P. Avgeriou, et al, "Tool Support for Architectural Decisions". In *Proc. 6th Working IEEE/IFIP Conf. on Software Arch.* (WICSA), pp. 44-53, 2007.

[30] M. Shahin, P. Liang, M.R. Khayyambashi, "Architectural design decision: Existing models and tools," In *Proc. WICSA/ECSA 2009*, pp. 293-296, 2009.

[31] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice, 3rd Edition.* Addison-Wesley, 2013.

[32] Jia Yu and Rajkumar Buyya, A Taxonomy of Workflow Management Systems for Grid Computing, Journal of Grid Computing, Volume 3, Numbers 3-4, Pages: 171-200, Springer Science+Business Media B.V., New York, USA, Sept. 2005.