Couchbase

NoSQL Database Evaluation Guide

How Leading NoSQL Databases Compare Across the Eight Core Requirements

TABLE OF CONTENTS

Introduction

Digital Economy

The business world is undergoing massive change as industry after industry shifts to the Digital Economy. It's an economy powered by the Internet and other 21st-century technologies — the cloud, mobile, social media, and big data. At the heart of every Digital Economy business are its web, mobile, and Internet of Things (IoT) applications: they're the primary way companies interact with customers today, and how companies run more and more of their business. The experiences that companies deliver via those apps largely determine how satisfied — and how loyal — customers will be.

Building and running these web, mobile, and IoT applications has created a new set of technology requirements. Enterprise architecture needs to be far more agile than ever before, and requires an approach to real-time data management that can accommodate unprecedented levels of scale, speed, and data flexibility. Relational databases are unable to meet these new requirements, and enterprises are therefore turning to NoSQL database technology.

Three Phases of NoSQL Evolution and Adoption: From Grassroots to Mainstream

Enterprise adoption of NoSQL has unfolded in three overlapping phases. In phase I (which started around 2010), developers required flexibility to support the agile development of proof of concepts and small applications. In phase II (which began around 2013), enterprises required performance, scalability, and availability to develop and/or migrate targeted mission-critical services. In phase III (which is just starting in late 2015), both developers and enterprises require a general-purpose database that combines flexibility, performance, scalability, and availability with a comprehensive query language and powerful indexing to replatform all mission-critical applications and services for the Digital Economy.

About this Evaluation Guide

This guide defines and details the eight core requirements for an effective NoSQL database. Based on those requirements, the guide articulates how databases do or do not meet those requirements, and points out what to look for and what to avoid. It begins with Data Access because the key requirement for Phase III applications in the Digital Economy is the ability to query data with an expressive language that enables developers to query any type of data independent of how it is modeled.

Criteria

- **Data Access**
- **Performance**
- **Scalability**
- **Availability**
- **Multiple Data Centers**
- **Big Data Integration**
- **Administration**
- **Mobile**

The Eight Core Requirements for an Effective NoSQL Database:

High-Level Overview

While every company has its own specific set of requirements for the NoSQL database technology that best fits its use case(s), there's a core set of requirements that figure into most evaluations. Those requirements fall into eight categories, as defined below. This section provides a high-level overview of those eight core requirements. Later sections of the guide delve more deeply into each core requirement, followed by a comparison of leading NoSQL databases against the full set of requirements.

Data Access

An effective NoSQL database must support the agile development of interactive applications with a flexible data model and a comprehensive query language, separating how data is modeled from how it is accessed. These capabilities enable architects and developers to create and modify the data model independent of how the data is queried. The query language must be capable of expressing complex queries on nested and/or referenced data with the ability to NEST, UNNEST, and JOIN related data. Ideally, the query language extends SQL, the most powerful, proven, and well understood of all query languages. In addition, the database should support geospatial and full-text search, and data processing with MapReduce.

Performance

An effective NoSQL database must meet the user experience requirements of highly interactive applications and the SLAs of mission-critical services with consistent, high performance; efficient use of memory and persistent storage; asynchronous operations; concurrent reads and writes; and topology-aware clients. To be specific, it must be able to provide low latency read and write operations at high throughput. Performance is not only important to the user experience; it also has a direct impact on cost and complexity: higher-performance databases require less hardware and fewer nodes.

Scalability

In order to support interactive applications with large numbers of users, large amounts of data, or both — whether from the beginning or as the result of exponential growth — an effective NoSQL database must provide simple, efficient, and reliable scaling on demand and without delay. It should require little to no effort to add a single node or many, since the process and effort should not change as the database scales. And it should be possible to scale individual database services (querying, indexing, and storage) separately, in addition to scaling the database as a whole.

Availability

To ensure application and/or service uptime, an effective NoSQL database must maintain availability by providing a resilient architecture (i.e., no shared resources, no single point of failure) that leverages networking, topology, smart clients, and replication to survive unplanned outages and failures, regardless of scope. In addition, online operations such as backing up and restoring data, and upgrading the database, should be able to be performed while the database remains online without requiring any downtime. Finally, the database must be capable of leveraging multiple data centers in such a way that all operations can be immediately rerouted to a different data center without a noticeable delay.

Geographic Distribution

An effective NoSQL database must be able to leverage multiple data centers for high availability, fast disaster recovery, and high performance with local reads and writes. The ability to read and write any data to any data center on demand is critical to availability and performance. It not only enables applications to continue functioning should a data center fail, but it enables them to take advantage of a local database for faster reads and writes. In addition, the database must be flexible enough to support global operations with custom topologies that utilize unidirectional replication between some data centers and bidirectional replication between others.

Big Data Integration

An effective NoSQL database is the foundation of any real-time big data architecture. It must be capable of functioning as both a source and a destination of data by integrating not only with Hadoop for long-term storage and offline processing, but also with Spark, Storm, Kafka, Elasticsearch, Solr, and more to enable stream processing, high throughput messaging, distributed full-text search, and more. While it is important to support batch integration with Hadoop, the new requirement is streaming integration with platforms like Spark and Storm to enable low-latency analytics and iterative machine learning by (a) continuously streaming operational data to them and (b) storing their results so they can be accessed via web and mobile applications.

Administration

An effective NoSQL database must provide administrators with comprehensive management and monitoring capabilities via a powerful, full-featured administration console and API while at the same time automating the processes including, but not limited to, the distribution and replication of data. However, administrators must have the option to perform critical operations on demand with the push of a button, and without having to take the database offline — including rebalancing data when nodes are added to the cluster. In addition, administrators should be able to perform both cumulative and incremental backups and restores on demand, online, and regardless of node failures.

Mobile

An effective NoSQL database must provide mobile database capabilities, including fast and consistent access to data, with or without a network connection. With an embedded local database and built-in multi-master synchronization, the mobile database must allow all devices to continue to operate while disconnected from the global network. An effective mobile NoSQL solution must also provide security for data at rest, data in motion and data in the cloud.

The Eight Core Requirements for an Effective NoSQL Database: Detailed Breakdown

The following section explains in greater detail each of the eight core requirements for an effective NoSQL database. Each subsection concludes with an at-a-glance checklist of "What to look for" and "What to avoid."

Data Access

An effective NoSQL database must enaable developers to access data in different ways depending on application requirements and the data, and must have a full-featured query language based on the database industry standard, SQL. In addition, the database must provide clients in many languages from Java to Go and for mobile platforms, too.

SQL

SQL is the proven, de facto industry standard database query language, familiar to all developers. An effective NoSQL database must provide a full-featured query language based on SQL, which is both expressive and powerful. It enables applications to sort, filter, transform, aggregate, and combine data with a single query and little to no code. By contrast, databases with proprietary APIs or partial query languages lack features such as sorting, aggregation, and joins. These limitations require the application to work with inefficient, possibly ineffective "table per query" or "single table" data models.

MapReduce

An effective NoSQL database must provide incremental MapReduce, which enables applications to index, sort, filter, and aggregate data, and to do so faster. With incremental MapReduce, the first time a MapReduce function is performed, it processes the existing data and generates results. Then, when new data is added, the database automatically processes the new data and merges the new results with the previous results. By contrast, databases that can't perform MapReduce in increments must process the entire data set every time, slowing down as the size of the data set grows.

Geospatial

An effective NoSQL database must provide geospatial indexes, which enable applications to search not only based on location but on any dimension or multiple dimensions, and to search on location and multiple dimensions together (e.g., location + hours). In addition, a NoSQL database should support standards like GeoJSON, and be flexible enough to support arbitrary coordinates and numbers. By contrast, a database that's limited to just two-dimensional geospatial search can only search based on location. For example, it can find "all restaurants in a city," but it can't find "all restaurants in a city that are open after midnight."

Full-Text Search

An effective NoSQL database should include native full-text search and it should support integration with leading full-text search products like Elasticsearch and LucidWorks/Solr. Without such integration support, companies that already have these products installed won't be able to easily leverage them.

WHAT TO LOOK FOR

SQL-Based Query Language with JOINS

Aggregation

Incremental MapReduce

GeoJSON

Multi-Dimensional Geospatial Indexes

Native Full-Text Search

Elasticsearch and Solr Integration Support

WHAT TO AVOID

Proprietary Query Language

Inability to JOIN Data

Batch MapReduce

Two-Dimensional Geospatial Indexes

Unsupported Elasticsearch and Solr Integration

Performance

In order to meet high-throughput, low-latency requirements for reads ***and*** writes, an effective, high-performance NoSQL database must leverage memory, concurrency, and networking.

Caching

A database's caching architecture has a significant impact on performance. A database with a **managed object cache** and **write-through caching** delivers high performance by storing recent data in memory and by caching the data of individual reads ***and*** writes. By contrast, a database with a "block cache" is less efficient: it stores blocks of file data, and blocks may contain the data of multiple writes — i.e., writes that are not intended to be cached because their data is not being read. In addition, a database with "read-through caching" requires disk reads, because it does not cache data until it is read.

Locking

An effective NoSQL database should provide **fine-grained locking**, which can perform many reads and writes at the same time. Each write requires a lock; fine-grained locking capability provides many locks. By contrast, a database with "coarse-grained locking" limits the number of locks and therefore is limited to performing just a few writes at a time.

Clients

An effective NoSQL database should provide **topology-aware clients**, which ensure that all read and write requests require a single hop — from the client to the node. By contrast, if the database does not have topology-aware clients, read and write requests will require multiple hops — from the client, to the router/coordinator, to the node.

Writes

In order to perform writes without sacrificing consistency or performance, an effective NoSQL database should enable every node to contain **primary and replica data**, which utilizes every node. By contrast, databases where nodes contain either **primary or replica data** (not both) do not efficiently use every node; and databases that rely on **quorums** to maintain consistency require multiple nodes to perform a write. Both of those approaches negatively impact write performance.

Replication

An effective NoSQL database should provide **memory-to-memory** replication, which does not have to wait for data to be written to disk before replicating it. This architecture not only improves write performance, but it improves durability when replication is synchronous. An effective NoSQL database also enables **bidirectional geo-replication**, which can replicate data between multiple data centers to improve read and write performance and provide full data locality — applications can read and write any and all data to their data center.

WHAT TO LOOK FOR

Managed Object Cache

Write-Through Caching

Fine-Grained Locking

Topology-Aware Clients

Primaries with Replica Data

Memory-to-Memory Replication

Bidirectional Geo-Replication

WHAT TO AVOID

Block Cache

Read-Through Caching

Coarse-Grained Locking

Routers/Coordinators

Disk-to-Disk Replication

Unidirectional-Only Geo-Replication

Primaries without Replica Data

Quorums

Scalability

An effective NoSQL database must be highly scalable: Not only should it be able to increase capacity (data or throughput) or availability by adding nodes, but it should also be able to do so easily, on demand, and efficiently, by demonstrating linear scaling — i.e., when a node is added, its full capacity is added.

Nodes

An effective NoSQL database should be based on a single node type and a flat topology, which makes scaling easier, faster, and more efficient, because scaling is performed by adding one or more nodes on demand. In addition, it must be able to be deployed on commodity hardware or cloud infrastructure rather than expensive mainframes or appliances. By contrast, a database with multiple node types (e.g., primary/secondary/router) and a hierarchical topology is more complex and more difficult to manage: the process requires configuring a group of nodes and adding that group to the cluster. In addition, it may require moving nodes to different servers.

Services

An effective NoSQL database should provide elastic services for data storage, indexing, and querying, which can improve performance and resource utilization by running different services on different commodity hardware — for example, a fast processor is not required for every node, only those running the query service — and also make scaling faster and more efficient. For example, elastic services make it possible to scale the indexing and querying services to accommodate new features and more users without scaling the data storage service. As a result, it's not necessary to rebalance the data, or shift it around — a process that can impact overall performance until it is complete.

Queries

An effective NoSQL database should provide centralized querying, because it scales more efficiently and does not require every node to participate in performing queries. The data may be stored on many nodes for scalability and performance, while the query is performed on a single node. It then requests data only from nodes that contain part of the results. By contrast, a database that relies on distributed queries, or "scatter/gather" queries, performs the same query on every single node, which can slow down queries.

Clients

An effective NoSQL database should provide topology-aware clients, which allows the database to support a greater number of clients and applications, because more can be added without changing client configuration or scaling the database. By contrast, a database that relies on routers can run into issues when every application instance requires a local router and the number of routers overwhelms the database. If the routers are separated from the application instances, performance drops.

WHAT TO LOOK FOR

Single Node Type

Flat Topology

Topology-Aware Clients

Elastic Services

Centralized Querying

Heterogeneous Hardware Supported

WHAT TO AVOID

Multiple Node Types

Hierarchical Topology

Routers

Lack of Elastic Services

Scatter/Gather Queries

Homogenous Hardware Required

WHAT TO LOOK FOR

Shared-Nothing Architecture

No Single Point of Failure

Memory-to-Memory Replication

Rack Awareness

Primary Owners

Bidirectional, Cross–Data Center Replication

Full Write Locality

WHAT TO AVOID

Shared Resources

Required Nodes

Routers/Coordinators

Configuration Servers

Disk-to-Disk Replication

Quorums

Unidirectional-Only Cross–Data Center Replication

Limited Write Locality

Availability

To deliver high availability, an effective NoSQL database should implement a shared-nothing architecture with replication, rack awareness, replication across multiple data centers, and no single point of failure.

Architecture

An effective NoSQL database should be designed with a **shared-nothing architecture** and **no single point of failure**: with this architecture, there are no required nodes and no shared resources between nodes — the database can therefore tolerate the failure of any node or resource. By contrast, a database with required nodes (such as routers, proxies, or configuration servers) or with shared resources (storage, memory, or processors) can lose availability if any of them fail.

Replication

An effective NoSQL database should include automatic, configurable **memory-to-memory replication**, which ensures availability and consistency while maintaining write performance: the data is replicated to multiple nodes, and it is fast. If a node fails, the data remains available. By contrast, a database that requires disk IO for replication can only provide one or the other — availability and consistency, or write performance.

Consistency

An effective NoSQL database should be designed with **primary owners**, which ensures availability and consistency by routing reads and writes of the same data to the same node. While the data is replicated, only a single node needs to be available to perform writes. By contrast, a database that relies on quorums requires the majority of quorum members to be available. If not, the data becomes unavailable.

Geographic Distribution

An effective NoSQL database provides **bidirectional, cross–data center replication** (XDCR) between independent data centers, which ensures availability by enabling applications to read and write all data to any data center on demand and without a noticeable delay — all data centers can read and write all data. This capability not only improves availability, but it improves performance — all reads and writes are performed locally — and enables disaster recovery with the ability to recover data from a remote data center. By contrast, a database with only unidirectional replication must perform a failover first — resulting in a temporary loss of availability.

Clients

An effective NoSQL database should provide **topology-aware clients**, which ensures availability because clients are aware of node failures. If a node fails, clients will be made aware of it and can route reads and writes to a different node. By contrast, a database without topology-aware clients requires routers or proxies between clients and nodes. If a router or proxy fails, the database can become unavailable because the clients cannot reach it. In addition, an effective NoSQL solution should provide an embedded database for mobile platforms with built-in, automatic synchronization that is always available, regardless of whether or not the remote database is available.

Geographic Distribution

In order to deliver high availability, fast disaster recovery, and high performance, an effective NoSQL database must leverage cross–data center replication and multiple data centers.

Locality

An effective NoSQL database should provide **bidirectional cross–data center replication**, which enables applications to read and write all data to any data center not only for high availability and disaster recovery, but for performance — applications can perform local reads and writes. By contrast, a database with only unidirectional replication cannot always perform local writes, because either (a) only one data center performs writes, or (b) each data center can only perform writes for a subset of the data.

Topology

An effective NoSQL database, which provides bidirectional replication between **independent clusters**, has the flexibility required to support a variety of topologies: ring (sequence of one-to-one), hub and spoke (one-to-many or many-to-one in parallel), mesh (many-to-many in parallel), and tiered or mixed combinations. By contrast, a database with bidirectional replication and a single cluster is limited to a mesh topology; and a database with unidirectional replication and a single cluster is limited to a hub and spoke topology.

Control

An effective NoSQL database should enable a dedicated **cross–data center replication implementation**, which not only provides advanced functionality, but is also easier to manage. Administrators can **pause, resume, or cancel** replication on demand; configure **filtering** to limit replication based on application, tenant, geography, and more; and use it to perform **data recovery**. By contrast, a database that relies on standard replication, without dedicated cross–data center replication, provides limited functionality and is difficult to manage because it is not separate from intra-cluster replication.

Replication

In order to reduce replication latency and thus the window of inconsistent data, an effective NoSQL database should use **memory-to-memory replication** to avoid disk IO. By contrast, a database with disk-to-disk replication can suffer from large windows of inconsistent data, because it has to wait for data to be written to disk before replicating it.

WHAT TO LOOK FOR

Bidirectional Replication

Independent Clusters

Memory-to-Memory Replication

Optimized Cross–Data Center Replication

Filtered Replication

Pause/Resume

Data Recovery

WHAT TO AVOID

Unidirectional Replication

Single Cluster

Standard Replication

Full Replication Only

Big Data Integration

An effective NoSQL database must support integration with big data, analytics, and reporting infrastructure with both batch and streaming data flows.

Hadoop

In order to import data from and export data to Hadoop, an effective NoSQL database should provide a **certified Sqoop plugin**. By exporting data to Hadoop, data can be processed by multiple MapReduce or Spark jobs — i.e., the data is in Hadoop. By contrast, a database that relies on a MapReduce input source forces Hadoop to ingest all of the data every time a job is run — i.e., the data remains in the database.

Spark

An effective NoSQL database should provide complete Spark integration, enabling the database to be used as a source of data for **Spark**, **Spark Streaming**, and **Spark SQL**, in addition to being used to persist the results. By contrast, a database that's limited to Spark and Spark SQL integration can be used as an input and/or output source for offline analytics, but it can't be used as an input for Spark Streaming for real-time analytics.

Kafka

An effective NoSQL database should provide complete **Kafka** integration, which enables the database to be used as a **producer** (source of messages) and/or **consumer** (destination of messages). When the database is used as a producer, data is published to a message queue as soon as it is written to the database. When the database is used as a consumer, data is written to the database as soon as it is published to the message queue.

ETL/BI/Reporting

An effective NoSQL database should come with standard, full-featured **ODBC/JDBC drivers** and a query language based on SQL that can be used with analytics and data integration tools without requiring a custom connector or adapter. By contrast, a database without ODBC/JDBC drivers, or with drivers that do not wrap a SQL-based language, provides limited functionality and performance because it has to implement query logic within the driver.

WHAT TO LOOK FOR

- **Certified Sqoop Plugin**
- **Spark Input/Output**
- **Spark SQL Input**
- **Spark Streaming Input**
- **Kafka Consumer and Producer**
- **ODBC/JDBC Drivers**
- **SQL-Based Query Language**

WHAT TO AVOID

- **No Sqoop Plugin**
- **No Spark SQL Input**
- **No Spark Streaming Input**
- **No Kafka Integration**
- **No Supported Full-Text Search Integration**
- **No SQL-Based Query Language**

Administration

An effective NoSQL database should give administrators access to a complete administrative console and API that provides them with all the tools necessary to easily manage and monitor deployments of any size and scale.

Management

An effective NoSQL database should provide a full-featured administration console and API giving administrators complete control of all database operations — including the ability to add or remove nodes, rebalance data, perform backups, restore data, failing over nodes, restore failed nodes, and more — all on demand. By contrast, databases with limited administration consoles require administrators to perform some tasks manually, either by editing configuration files or performing command line operations, or in some cases by performing tasks automatically without the option for administrative control.

Monitoring

A critical part of managing both small and large deployments is a full complement of statistics and data for monitoring. An effective NoSQL database must therefore give administrators access to hundreds of metrics, both cluster-wide and node-specific. By contrast, a database with limited metrics may not provide administrators with information such as swap usage, CPU utilization, the number of connections, disk reads per second, the percent of data resident in memory, replication queue size, geo-replication status, and much more.

Backup/Restore

In order to efficiently restore the data of a cluster, an effective NoSQL database must enable administrators to perform cumulative and incremental backups and restores. This includes taking cumulative and incremental backups, as well as restoring data from cumulative or incremental backups. By contrast, a database that's limited to snapshots forces administrators to restore all of the data, even if only a small percentage of it needs to be restored. In addition, administrators should be able to perform backups on demand, as well as restore data regardless of whether or not nodes have failed.

WHAT TO LOOK FOR

- **Manual Rebalancing**
- **On-Demand Backups**
- **Operations via Admin Console**
- **Cumulative and Incremental Backup and Restore**
- **Configuration UI**
- **200+ Metrics**
- **Cluster-Wide Log Aggregation**
- **Online Restore**

WHAT TO AVOID

- **No Manual Rebalancing**
- **No On-Demand Backups**
- **Operations via Command Line**
- **Scheduled Backups**
- **Configuration Files**
- **Limited Metrics**
- **Per-Node Log Analysis**
- **Snapshot Backup and Restore**
- **Offline Restore**

Mobile Database

An effective NoSQL database must provide mobile database capabilities, including fast and consistent access to data, with or without a network connection.

Local Database

An effective mobile NoSQL solution must provide an embedded database that runs on the device and has a small footprint. The database must provide a flexible data model, perform fast queries against data, and publish change events that allow applications to listen/observe for data changes.

Synchronization

An effective mobile NoSQL solution must provide built-in multi-master synchronization that allows for secure synchronization between local and remote databases. It should support flexible deployment topologies — including Star, Tree, and Mesh — and allow different parts of the system (in addition to the devices) to operate while disconnected from the global network.

Security

An effective mobile NoSQL solution must provide security for customizable user authentication, fine-grained data read/write access, data transport over a secure channel, encrypted data storage on device, and encrypted data storage in the cloud.

WHAT TO LOOK FOR

- **Offline Data Access**
- **Flexible Data Model**
- **Fast Queries**
- **Change Events**
- **Multi-Master Replication**
- **Flexible Deployment Topology**
- **Security**
- **Managed Synchronization**

WHAT TO AVOID

- **Reliant on Network**
- **Inflexible Data Model**
- **Key/Value Only**
- **Polling for Changes**
- **Cache and Write Queues**
- **Only Support for Star Topology**
- **DIY Security**
- **DIY Synchronization**

Conclusion

The process of evaluating a NoSQL database begins with identifying its architecture and understanding its features, both the capabilities and the limitations. The architecture and features have to meet developer, enterprise, and application requirements. If they do, the next step is to perform a hands-on evaluation — install the database, build a proof of concept or migrate a simple application, and see how well it does or does not perform under load.

Appendix A: NoSQL Database Evaluation Checklist

The following checklist evaluates Couchbase Server, MongoDB, and Cassandra (DataStax Enterprise). However, it can be used as a framework for evaluating other NoSQL databases as well.

PERFORMANCE

	Couchbase	mongoDB	DATASTAX
Managed Object Cache	YES	NO	YES
Write-Through Caching	YES	YES	NO
Fine-Grained Locking	YES	MMAP (NO), WiredTiger (YES)	YES
Topology-Aware Clients	YES	NO	YES
Primaries with Replica Data	YES	NO	YES
Asynchronous Indexing	YES	NO	NO
Memory-to-Memory Replication	YES	NO	NO
Bidirectional Geo-Replication	YES	NO	YES
TOTAL	8	1-2	5

AVAILABILITY

	Couchbase	mongoDB	DATASTAX
Shared-Nothing Architecture	YES	NO	YES
No Single Point of Failure	YES	NO	YES
Memory-to-Memory Replication	YES	NO	NO
Primary Owner	YES	YES	NO
Rack Awareness	YES	YES	YES
Bidirectional Geo-Replication	YES	NO	YES
Full Write Locality	YES	NO	YES
TOTAL	7	2	5

Appendix A: NoSQL Database Evaluation Checklist (Continued)

MULTIPLE DATA CENTERS

	Couchbase	mongoDB	DATASTAX
Independent Clusters	YES	NO	NO
Unidirectional Geo-replication	YES	YES	NO
Bidirectional Geo-replication	YES	NO	YES
Memory-to-Memory Replication	YES	NO	NO
Geo-replication Implementation	YES	NO	NO
Filtering	YES	NO	NO
Pause / Resume	YES	NO	NO
Data Recovery	YES	NO	NO
Ring	YES	NO	NO
Hub and Spoke	YES	YES	NO
Mesh	YES	NO	YES
Always Read Local	YES	YES	YES
Always Write Local	YES	NO	YES
TOTAL	13	3	4

SECURITY

	Couchbase	mongoDB	DATASTAX
Authentication	YES	YES	YES
Authorization	YES	YES	YES
Auditing	YES	YES	NO
Encrypted Access	YES	YES	YES
Alerts	YES	YES	YES
TOTAL	5	5	4

Appendix A: NoSQL Database Evaluation Checklist (Continued)

DATA ACCESS

	Couchbase	mongoDB	DATASTAX
Schemaless	YES	YES	NO
SQL-based Query Language	YES	NO	NO
JOINS	YES	NO	NO
Compound Indexes	YES	YES	NO
Subqueries	YES	NO	NO
Ad-hoc Sorting	YES	YES	NO
Aggregation	YES	YES	NO
Pagination	YES	YES	YES
Incremental MapReduce	YES	NO	NO
GeoJSON	YES	YES	NO
Two-Dimensional Geospatial Indexes	YES	YES	NO
Multi-Dimensional Geospatial Indexes	YES	NO	NO
Native Full Text Search	YES	YES	YES
Supported Elasticsearch Integration	YES	NO	NO
Supported Solr Integration	YES	NO	NO
REST API	YES	NO	NO
TOTAL	16	7	2

SCALABILITY

	Couchbase	mongoDB	DATASTAX
Single Node Type	YES	NO	YES
Flat Topology	YES	NO	YES
Elastic Services	YES	NO	NO
Centralized Querying	YES	NO	NO
Topology Aware Clients	YES	NO	YES
TOTAL	5	0	3

Appendix A: NoSQL Database Evaluation Checklist (Continued)

ADMINISTRATION

	Couchbase	mongoDB	DATASTAX
Controlled Rebalancing	YES	NO	YES
Push-Button Rebalancing	YES	NO	YES
200+ Metrics	YES	NO	NO
Backup / Restore	YES	YES	YES
Cumulative Backup	YES	NO	NO
Cumulative Restore	YES	NO	NO
Incremental Backup	YES	YES	YES
Incremental Restore	YES	NO	NO
Incremental Backup On Demand	YES	NO	NO
Full Backup On Demand	YES	NO	YES
Live Restore	YES	NO	YES
Live Restore, Different Topology	YES	NO	NO
TOTAL	12	2	6

BIG DATA REPLICATION

	Couchbase	mongoDB	DATASTAX
Certified Sqoop Plugin	YES	NO	NO
Spark Input	YES	YES	YES
Spark Output	YES	YES	YES
Spark SQL Input	YES	NO	YES
Spark Streaming Input	YES	NO	NO
Kafka Consumer	YES	NO	NO
Kafka Producer	YES	NO	NO
ODBC/JDBC Drivers	YES	YES	YES
SQL Query Language	YES	NO	NO
TOTAL	8	3	4