Wahl ohne Qual

NoSQL Reality Check

Dirk Bangel, Kai Weingärtner

Relationale Datenbanken haben sich fast überall zur Datenverwaltung durchgesetzt. Seit kurzem werden aber auch andere Konzepte unter dem Begriff NoSQL diskutiert und vor allem bei Unternehmen schon erfolgreich eingesetzt. Es existieren derzeit mehr als zwanzig NoSQL-Datenbanken mit teilweise unterschiedlichen Konzepten. Bisher wurde aber kaum beantwortet, wie NoSQL-Datenbanken zu den Anforderungen und Architekturen typischer Geschäftsanwendungen passen. Es ist also an der Zeit, Einsatzszenarien für Datenverwaltungssysteme neu zu bewerten. In diesem Beitrag werden Entscheidungshilfen für den Einsatz in Geschäftsanwendungen gegeben.

Kernkonzepte

Mit dem vorhandenen Java-Handwerkzeug im Datenbankumfeld kann man sich bei der Entwicklung von Geschäftsanwendungen beinahe entspannt zurücklehnen. Ob einfache Datenbankabfragen mittels Frameworks wie JDBC oder das Persistieren von POJOs mittels JPA. Viele leistungsstarke Frameworks prägen den Java-Alltag. Doch inzwischen haben sich die Zeiten geändert: Die Datenmenge steigt rasant und die Datenmodelle entwickeln sich ständig weiter.

NoSQL-Datenbanken setzen genau bei dieser Problematik an. Vieles läuft aber in der NoSQL-Welt anders als bei relationalen Datenbanken. Besonders die Datenmodellierung und Abfragen unterscheiden sich voneinander. Um einen Vergleich zu relationalen Datenbanken ziehen zu können, ist ein grundlegendes Verständnis der Kernkonzepte der NoSQL-Datenbanken notwendig. Die zwei verbreitetsten NoSQL-Konzepte sind

- ColumnFamily Datastores und
- Document Datastores.

Diese werden hier vorgestellt und mit dem bekannten relationalen Modell verglichen.

ColumnFamily Datastores

Die ColumnFamily Datastores realisieren das Google-BigTable-Konzept [Chang06]. Obwohl viele Bezeichnungen mit denen von relationalen Datenbanken identisch sind, wird kein rein relationales Modell beschrieben. Tabellen unterteilen sich in ColumnFamilies und dann erst in Spalten.

Die Spalten von ColumnFamily Datastores werden als Qualifier bezeichnet und können dynamisch ergänzt oder entfernt werden. Die ColumnFamilies sind hingegen statisch und mit relationalen Tabellen vergleichbar. Die Tabelle inklusive aller zugehörigen ColumnFamilies (die BigTable) ist wie eine relationale Datenbank zu verstehen. Aus relationalen Gesichtspunkten ist die BigTable dann ein logisches und physisches Modell in der ersten Normalform.

Wie bei relationalen Datenbanken besitzt jede Zeile einen eindeutigen Schlüssel (RowKey). Anders als bei relationalen Datenbanken gibt es aber nur einen RowKey pro Zeile, der als Primärschlüssel dient (s. Abb. 1).

Eine weitere Besonderheit von ColumnFamily Datastores liegt in der einzelnen Zelle. Jede Änderung erzeugt eine neue Version der Zelle. So können jederzeit auch ältere Stände einer Row abgefragt werden, wie Abbildung 2 zeigt. Abgesehen von den Versionseigenschaften sind Zellen uninterpretierte Byte-Arrays. Es existieren somit keine Datentypen im Datastore. Datentypen können aber programmatisch durch die jeweiligen Datastore-APIs oder durch Java-Mittel im Client hergestellt werden.

Soweit zum Grundkonzept der ColumnFamily Datastores. Für die weiteren Vergleiche des ColumnFamily-Konzepts werden weiter unten die beiden freien Java-Implementierungen HBase und Cassandra aufgegriffen.

Document Datastores

In Document Datastores werden Daten als strukturierte, hierarchische und schemalose Datenstrukturen gespeichert, den Dokumenten. Die bekanntesten NoSQL Document Datastores sind CouchDB und MongoDB. Sie nutzen zur Definition des Datenmodells das JSON-Format.

Document Datastores werden als schemalos bezeichnet, weil potenziell jedes Dokument eine eigene Datenstruktur haben kann. Was auf den ersten Blick nach Verlust des Datenmodells aussieht, schafft auf den zweiten Blick viel Flexibilität bei der Änderung des Domänenmodells. So können bei einzelnen

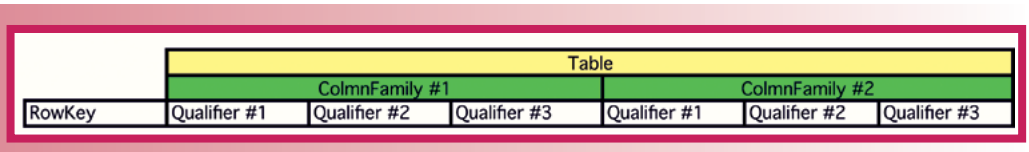

Abb. 1: Grundstruktur ColumnFamily

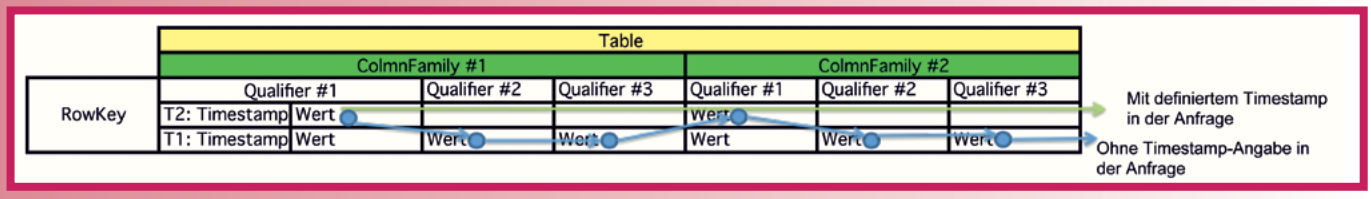

Abb. 2: Timestamps (Versionen) bei ColumnFamily Datastores

Dokumenten jederzeit Attribute ergänzt oder entfernt werden, ohne dass dazu (wie bei relationalen Datenbanken) ein gemeinsames Schema angepasst werden muss.

Ein Dokument kann wiederum eingebettete Kind-Dokumente enthalten. Dokumente haben außerdem immer einen eindeutigen Identifikator, über den sie in anderen Dokumenten referenziert werden können. Es besteht also zur Abbildung von Beziehungen immer die Wahl zwischen dem Einbetten oder Referenzieren von anderen Dokumenten. Jedes eingebettete Dokument wird immer mit dem Eltern-Dokument geladen. Referenzierte Dokumente können hingegen separat geladen werden.

Vergleicht man Dokumente mit Tabellen einer relationalen Datenbank, so bildet eine Haupttabelle das Dokument und die abhängigen Tabellen die eingebetteten Dokumente ab. Beziehungen zu anderen Haupttabellen stellen die Referenzen aus Dokumenten dar. Hier ein Beispiel für die Abbildung einer Bestellung mit eingebetteten Bestellpositionen und referenzierten Produkten:

```
Bestellung: {
  id: 123456,
  datum : new Date(2010, 1, 1),
  positionen : [
    {produktId : 1234, anzahl : 2},
    {produktId : 2345, anzahl : 1}
  ]
}
```

Abbildung des Domänenmodells

Beinahe jede Problemstellung wird heutzutage in das relationale Modell abgebildet. Anhand eines einfachen Domänenmodells (s. Abb. 3) wird die Abbildung in die Modelle der NoSQL-Konzepte vorgestellt.

ColumnFamily Datastores

Die Abbildung des Domänenmodells aus Abbildung 3 auf ColumnFamily Datastores ist, wie bereits erwähnt, mit der ersten Normalform der relationalen Datenbanken vergleichbar. Die erste Normalform ist dann erreicht, wenn die nicht atomaren Attribute, wie zum Beispiel Aufzählungen, aufgelöst sind. Anders als bei relationalen Datenbanken empfiehlt es sich nicht, für die unterschiedlichen Werte neue Zeilen einzufügen. Das ColumnFamily-Modell hat seine Stärke in der Abbildung auf Spalten. Anders als bei relationalen Abbildungen nimmt so die Redundanz nicht zu und der Primärschlüssel behält seine Bedeutung.

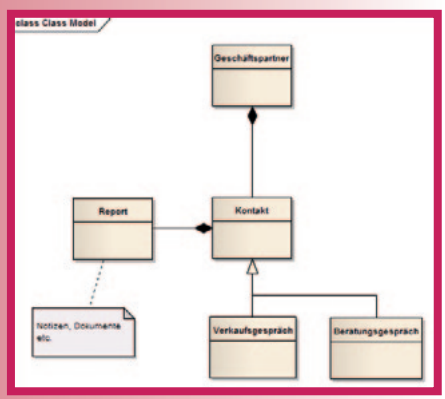

Abb. 3: Beispieldomänenmodell

Natürlich bleibt es auch bei ColumnFamily Datastores oft nicht nur bei einer Tabelle. Entgegen den relationalen Datenbanken dienen diese jedoch nicht der Normalisierung. Es sind in der Regel Tabellen für Indizes (sogenannte Sekundär-Indizes). Diese Indizes helfen, komplexe Eigenschaften auf den einen möglichen RowKey abzubilden. Bei ColumnFamily Datastores bestehen zwei Möglichkeiten: Entweder drückt der RowKey alle Selektionskriterien aus oder es werden sekundäre Indizes verwendet. Zur Umsetzung lohnt sich für HBase ein Blick auf das Framework „HBase indexing library“ [HBaseIndex].

Bei Objektmodellen liegt die Herausforderung in der Abbildung von Vererbung und den Beziehungen zwischen Objekten. Beides ist bei ColumnFamilies einfach durch das dynamische Hinzufügen von Spalten umzusetzen. Um zum Beispiel die Hierarchie in der Vererbung auszudrücken, ist jeder Vererbungsebene eine neue ColumnFamily mit entsprechenden Spalten zuzuweisen oder für jede Beziehung jeweils eine eigene ColumnFamily. Das Referenz-Beispiel aus Abbildung 3 sieht unter Berücksichtigung dieser Regeln wie in Abbildung 4 aus.

Eine wichtige Optimierung sind zudem die Einstellungen (Attribute) der ColumnFamily. Auf diese wird aber in diesem Beitrag nicht eingegangen. Weiterführende Informationen sind in den Blog-Einträgen unter [DeaGhe04] zu finden.

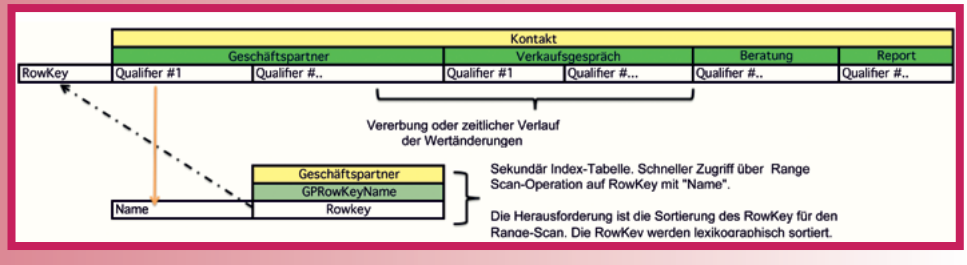

Abb. 4: Beispielabbildung bei ColumnFamily Datastores

Document Datastores

Nimmt man bei Document Datastores nicht das relationale, sondern das Objektmodell als Grundlage, fällt die Abbildung leicht. Der konzeptionelle Unterschied zwischen Objekt- und Datenmodell fällt gegenüber dem relationalen Modell deutlich geringer aus. Dadurch können mehrwertige Attribute direkt auf Listen im Dokument abgebildet werden und erfordern keine Trennung in zwei Tabellen, auch wenn die Werte wiederum komplexe Strukturen sind.

Vererbung lässt sich ebenso leicht mit Dokumenten umsetzen. Neue Attribute einer Subklasse können ohne eine Schemaänderung zu einem Dokument hinzugefügt werden. Um die passende Subklasse wieder laden zu können, kann der Typ im Dokument mitgespeichert werden. Hier die Abbildung des Beispielmodells aus Abbildung 3 in ein Dokument (gemäß CouchDB und MongoDB, im JSON-Format):

```
Geschäftspartner : {
  vorname : “Max”,
  nachname : “Mustermann”,
  kontakte : [
    { classname : “Verkauf”,
      reports : []},
    { classname : “Beratung”,
      reports : []}
  ]
}
```

Sowohl Kontakt als auch Report wurden als abhängige Objekte des Geschäftspartners betrachtet und deshalb als eingebettete Dokumente definiert. Im obigen Beispiel ist eine mögliche Abbildung einer 1:n-Beziehung als Liste von eingebetteten Dokumenten dargestellt.

Das Document-Datastore-Modell wird direkt durch das Objektmodell in der Anwendung defi-

niert. Änderungen am Objektmodell wirken sich also direkt auf das Format der persistierten Dokumente aus.

Umgang mit Schemaänderungen

Im Lebenszyklus einer Anwendung sind Änderungen an den Geschäftsobjekten vorprogrammiert. Die Evolution der Geschäftsobjekte führt zwangsweise zu Anpassungen des Modells in der Datenbank. Bei relationalen Datenbanken führt dies zu Migrationsaufwand.

ColumnFamily Datastores

Auch ColumnFamily Datastores bewahren nicht vor Problemen bei Schemaänderungen. Alle Änderungen haben im Rahmen des BigTable-Schemas Auswirkungen auf die durchgeführte Abbildung des Domänenmodells. Durch die Flexibilität beim Hinzunehmen von Spalten innerhalb einer Column-Family können derartige Schemaänderungen aber deutlich einfacher durchgeführt werden. Das nachträgliche Löschen von Spalten (den Qualifiern) oder das Einbringen oder Löschen von ColumnFamilies ist da schon aufwendiger. Bei HBase ist vor der Änderung der ColumnFamilies die Tabelle (BigTable) zu deaktivieren und bei Cassandra ist sogar der ganze Datastore neu zu starten, um hier nur zwei Beispiele zu nennen.

Sind größere Schemaänderungen nötig, hilft aber der zur Verfügung stehende Map/Reduce-Mechanismus [DeaGhe04] weiter. Durch Map/Reduce können auch rechenintensive Operationen im Bruchteil der eigentlich benötigten Zeit durchgeführt werden (s. [Ana09]). Mittels Map/Reduce können regelmäßige Replikation oder Migrationen bei Schemaänderungen innerhalb kürzester Zeit durchgeführt werden.

Document Datastores

Da ein Dokument schemalos ist, ist auch keine Schemamigration und damit kein Sperren der Datenbank notwendig. Wenn die geänderte Anwendung damit umgehen kann, dass neu hinzugekommene Attribute nicht gesetzt sind, ist nicht einmal eine Migration der vorhandenen Dokumente erforderlich.

Müssen hingegen in den Dokumenten Datenänderungen durchgeführt werden, so bietet zum Beispiel MongoDB die Möglichkeit, Batch-Operationen auf Dokumenten per Map/Reduce durchzuführen.

Sind neue Subklassen hinzugekommen, so können diese sofort ohne Weiteres gespeichert werden. Es wird nur ein Mittel benötigt, um beim Laden wieder die korrekte Klasse instanzieren zu können.

Abfragesprachen

Mindestens so wichtig wie die Datenmigration sind die Möglichkeiten zur Datenabfrage. In relationalen Datenbanken steht mit dem SQL-Standard eine mächtige Abfragesprache zur Verfügung. Vordefinierte Abfragen können aus der Anwendung genauso durchgeführt werden, wie nachträglich entstandene Ad-hoc-Abfragen. Was bieten die NoSQL-Datastores hier?

ColumnFamily Datastores

Die Grundoperationen des BigTable-Konzepts entsprechen dem Ressource-Gedanken (Schlagwort RESTful). Es sind die Operationen Put, Get und Delete. Die Get-Operation dient zum Zugriff mittels RowKeys. Zusätzlich stehen für den sequenziellen Zugriff sogenannte Scans zur Verfügung – der Vergleich mit einem Cursor sei an dieser Stelle gestattet. Neben dem direkten Zugriff mittels bekanntem RowKey oder dem sequenziellen Auslesen von Daten können zusätzlich Filter wie zum Beispiel nach regulären Ausdrücken angegeben werden. Durch Filter werden so die unterschiedlichen Werte einer Spalte eingeschränkt. Da die Filter-Operationen (zumindest bei HBase) serverseitig ausgeführt werden, sind auch komplexe Filterbedingungen mit gutem Antwortzeit-Verhalten möglich. Diese Grundoperationen werden sowohl für HBase als auch für Cassandra über unterschiedliche Schnittstellen wie sprachspezifische API oder eine HTTP-Rest-Schnittstelle bereitgestellt.

Wie bereits aufgezeigt, unterstützen sowohl HBase als auch Cassandra die Verarbeitung durch Map/Reduce. Für HBase werden zwei umfangreiche SQL-Dialekte für analytische Anfragen bereitgestellt. Die Rede ist von den Frameworks HIVE [Hive] und PIG [Pig]. Beide können über die gewohnten JDBC-Mechanismen an Anwendung angebunden werden. Für Ad-hoc-Abfragen sind beide Frameworks aber nicht geeignet. Ursache ist deren meist aufwendige Initialisierung, die zu einer grundsätzlichen Latenzzeit in der Antwort führt. Ad-hoc-Abfragen werden für HBase durch das Projekt HBQL [HBql] angeboten. Auch HBQL ist ohne Probleme über JDBC einzubinden. Für Cassandra lässt eine derartige Unterstützung noch auf sich warten.

Abschließend sei erwähnt, dass auch bei ColumnFamily JOIN-Operationen verwendbar und sogar nicht untypisch sind. Einen guten Einstieg in diese Thematik bietet der Beitrag [Sla09].

Document Datastores

Document Datastores verstehen die gespeicherten Datenstrukturen und können darin beliebig tief in die Dokumentstruktur hinein auf Attributebene filtern. Zudem kann, wie bei einem SQL-Select, die Menge der zurückgelieferten Attribute beschränkt werden – um nicht immer vollständige Dokumente zu laden.

Grundsätzlich bieten MongoDB und CouchDB keine JOIN-Operation an. Wenn referenzierte Dokumente mit geladen werden sollen, müssen diese separat geladen und clientseitig verknüpft werden. Durch das Einbetten von Dokumenten lässt sich dies häufig umgehen.

Die Abfragesprachen von MongoDB und CouchDB unterscheiden sich deutlich. MongoDB entspricht eher dem SQL-Ansatz. Die Abfragesprache erlaubt beliebige Ad-hoc-Abfragen auf dem Datenmodell. Der Datastore-Administrator ist dafür verantwortlich, dass die Performance durch die Indizes sichergestellt ist. Mit einem Datenbank-Cursor kann durch eine Ergebnismenge navigiert werden.

Bei CouchDB können Abfragen nur auf zuvor erstellte Views ausgeführt werden. Views werden durch Map/Reduce-Funktionen in JavaScript beschrieben. Die initiale Berechnung einer View läuft über alle Dokumente und ist daher zeitintensiv. Ad-hoc-Abfragen sind also immer mit einem gewissen Zeitaufwand verbunden, doch grundsätzlich möglich. Spätere Abfragen auf den View können auf den berechneten Ergebnissen arbeiten und sind performant, da CouchDB automatisch für alle Suchschlüssel Indizes erzeugt.

Integration in die Java-Welt

Die Java-Unterstützung für relationale Datenbanken ist umfangreich. Ähnliche oder kompatible Schnittstellen existieren aber auch für die vorgestellten NoSQL Datastores. Eine detaillierte Ausführung von konkreten Beispielen finden Sie auf [Widas].

ColumnFamily Datastores

Sowohl HBase als auch Cassandra verfügen über ein breites Spektrum an Java-Werkzeugen. Da Cassandra ursprünglich

nicht speziell auf Java-Schnittstellen ausgelegt war, erfolgt der Zugriff von Hause aus mit dem Apache-RPC-Projekt Thrift Framework [Thrift]. Rund um HBase hat sich aber eine aktive Java-Community gebildet. Zum Beispiel das bereits erwähnte HBQL-Projekt, das einen vollwertigen JDBC-Treiber und eine SQL-Abfragesprache bereitstellt, oder den auf HBase optimierten Object-Mapper Meetup.Beeno [ghelm]. Meetup.Beno bietet zudem ein auf HBase ausgelegtes Criteria-API an. Auch die gewohnte Verwendung von XA-Transaktionen oder Transaktionen auf BigTable-Ebene wird von Hause aus durch HBase unterstützt. Der Verwendung in Java-Projekten steht somit nichts im Wege.

Bei Cassandra ist die Java-API-Unterstützung nicht so umfangreich. Mittlerweile haben sich aber auch für Cassandra einige brauchbare Java-APIs hervorgetan. So zum Beispiel der an JPA angelehnte Object-Mapper Kundera [Kundera].

Document Datastores

Sowohl für MongoDB als auch für CouchDB gibt es Java-Bibliotheken, die das Abbilden (Mapping) von Dokumenten auf Java-Klassen unterstützen. Allerdings hat sich dafür noch kein Document-Mapping-Standard, wie JPA für relationales Mapping, herausgebildet.

Hervorgehoben werden sollen hier Ektorp [ektorp] für CouchDB und Morphia [morphia] für MongoDB, die beide wie JPA dem Ansatz der Annotierung von POJO-Klassen folgen. Dabei lassen sich die notwendigen Annotationen oft auf die Kennzeichnung des Dokument-Identifikators beschränken oder wie bei Ektorp als Mixin außerhalb des POJOs beschreiben. Da Listen und Maps gültige Strukturen in Dokumenten sind, sind hier keine speziellen Abbildungsdefinitionen notwendig.

Beide bieten außerdem eine API für Abfrage- und Speicher-Operationen auf Klassenebene. Hierfür empfiehlt es sich, Data-Access-Objekte (DAO) zu erstellen, die die Spezifika der Datastore-API kapseln (wie z. B. die in JavaScript beschriebenen Map/Reduce-Funktionen in Ektorp).

Neben dem Zugriff über POJO-Klassen gibt es auch Bibliotheken, die das Dokument in Java als Maps oder JSON-Objekte repräsentieren. Dies ist bei Datenstrukturen sinnvoll, die ein sehr variables Datenmodell haben.

Fazit

Der Werkzeugkasten zur Datenspeicherung ist durch die NoSQL Datastores größer geworden. Relationale Datenbanken sind nicht mehr das einzige Mittel zur Lösung der Anforderungen an Geschäftsanwendungen. Die neuen NoSQL-Optionen erfordern aber eine genauere Betrachtung der Anwendung und eine Abwägung von Eigenschaften wie Skalierbarkeit, Verfügbarkeit und Konsistenz (den CAP-Eigenschaften, siehe [WiKiCAP]). Auch das Mapping des Domänenmodells auf das Datenbankschema oder die benötigte Flexibilität der Abfragesprache ist für die Architektur neu zu bewerten.

NoSQL-Datenbanken bieten Lösungen für die kostengünstige Daten-Partitionierung und -Replikation (also vereinfachen die Skalierung) an. Für viele Anwendungsfälle ist das Resultat eine hohe Lese- und Schreibperformance. Abschließend ist zu sagen, dass bei moderaten Anforderungen an die Skalierung und bei stark strukturierten, relativ stabilen Domänenmodellen relationale Datenbanken nach wie vor eine sichere Wahl sind. Der Reifegrad von relationalen Datenbanksystemen und die Erfahrungswerte, die bereits damit gesammelt wurden, stellen sicher, dass jedes betriebliche Problem bereits gelöst wurde und ein umfassender Tool-Support bereitsteht. Es ist aber absehbar, dass der Einsatz von NoSQL Datastores zunimmt.

Links

[Ana09] A. Anand, Hadoop Sorts a Petabyte in 16.24 Hours and a Terabyte in 62 Seconds, Yahoo! Developer Network, 11.5.2009, `http://developer.yahoo.com/blogs/hadoop/posts/2009/05/hadoop_so`

[Chang06] F. Chang u. a., Bigtable: A Distributed Storage System for Structured Data, in: Seventh Symposium on Operating System Design and Implementation (OSDI'06), 2006, `http://labs.google.com/papers/bigtable.html`

[DeaGhe04] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, in: Sixth Symposium on Operating System Design and Implementation (OSDI'04), 2004, `http://labs.google.com/papers/mapreduce.html`

[ektorp] Java API for CouchDB, Google Code, `http://code.google.com/p/ektorp/`

[ghelm] ghelmling/meetup.beeno, Simple Java Beans mapping for HBase, GitHub social coding, `http://github.com/ghelmling/meetup.beeno`

[HBaseIndex] Outerthought: HBase Indexing Library, `http://lilycms.org/lily/about/playground/hbaseindexes.html`

[HBql] HBql, Apache Hadoop, `http://www.hbql.com/`

[Hive] Apache Hadoop, `http://hive.apache.org/`

[Kundera] JPA 1.0 ORM library for the Cassandra database, Google Code, `http://code.google.com/p/kundera/`

[morphia] A type-safe java library for MongoDB, Google Code, `http://code.google.com/p/morphia/`

[Pig] Apache Hadoop, `http://pig.apache.org/`

[Sla09] B. Slatkin, Building scalable, complex apps on App Engine, in: Google IO 2009, `http://www.scribd.com/doc/16952419/Building-scalable-complex-apps-on-App-Engine`

[Thrift] `http://incubator.apache.org/thrift/`

[Widas] `https://www.widas.de/nosql`

[WiKiCAP] `http://de.wikipedia.org/wiki/CAP-Theorem`

Dirk Bangel arbeitet als Senior Consultant bei der WidasConcepts GmbH, einer IT-Unternehmensberatung mit den Schwerpunkten IT-Architekturen und Anwendungsentwicklung. Der Fokus seiner Arbeit liegt in der Business-Analyse, der Konzeption, Planung und Realisierung von Softwarearchitekturen mit Java/JEE und dem Projektmanagement.
E-Mai: dirk.bangel@widas.de

Kai Weingärtner arbeitet ebenfalls als Senior Consultant bei der WidasConcepts GmbH. Seine Tätigkeitsschwerpunkte liegen in Web-2.0-Themen sowie der Konzeption und Umsetzung von Web-basierten Java-EE-Anwendungen. Darüber hinaus ist er Autor verschiedener Fachartikel zum Thema Web 2.0 sowie Sprecher auf einschlägigen Fachveranstaltungen.
E-Mai: kai.weingaertner@widas.de