Datenbank Spektrum (2015) 15:229–233
DOI 10.1007/s13222-015-0194-1

KURZ ERKLÄRT

Polyglot Persistence

Felix Gessert · Norbert Ritter

Online publiziert: 4. September 2015

1 Motivation

Moderne Anwendungen werden zunehmend datengetriebener und verteilter, um immer mehr Nutzer bei stets wachsenden Datenmengen mit geringstmöglichen Antwortzeiten zu bedienen. Dabei werden nicht nur die verarbeiteten Daten heterogener, sondern auch die Anforderungen: Horizontale Skalierbarkeit, Schemaflexibilität und Ausfallsicherheit sind für viele Anwendungen unabdingbar geworden. Während klassische relationale Datenbanksysteme viele funktionale Anforderungen (z. B. ACID-Transaktionen, ausdrucksstarke Queries) erfolgreich abdecken, fehlt ihnen die Skalierbarkeit, Performance und Fehlertoleranz, die spezialisierte Systeme durch bewusst getroffene Kompromisse erzielen können. Die explosionsartig ansteigende Systemvielfalt der NoSQL- und Big-Data-Bewegung hat dazu geführt, dass für bestimmte Teilprobleme einer Anwendung oft besonders geeignete Datenbanksysteme zur Verfügung stehen.

Das Architekturmuster *Polyglot Persistence* bezeichnet den Einsatz verschiedener Datenbanken für verschiedene Anforderungen. Der Begriff wurde 2011 von Martin Fowler popularisiert und lehnt sich an Polyglot Programming an [6]. Der Kerngedanke von Polyglot Persistence ist, dass eine Abkehr von monolithisch konzipierten Anwendungen mit einer „one-size-fits-all"-Datenbank *Entwicklungsproduktivität* und *Performance* erhöhen kann. Polyglotte (mehrsprachige) Persistenz kann sich dabei sowohl auf einzelne Anwendungen als auch ganze Unternehmen beziehen.

Abb. 1 zeigt eine beispielhafte Polyglot-Persistence-Architektur für eine E-Commerce-Applikation, wie sie in ähnlicher Form für viele große Anwendungen Realität ist [3]. Die Daten werden entsprechend ihrer Anforderungen auf die jeweiligen Datenbanksysteme verteilt. So werden beispielsweise Finanztransaktionen über eine relationale Datenbank abgewickelt, um durch Datenbanktransaktionen Korrektheit sicherzustellen. Da Produktbeschreibungen ein natürliches semistrukturiertes Aggregat bilden, eignen sie sich ideal zur Speicherung in einem verteilten Document Store, der Skalierbarkeit des Datenvolumens und der Leselast garantiert. Wide-Column Stores sind durch ihre Log-strukturierte Speicherung gut geeignet, um hohe Schreibraten der Anwendungsereignisse zu bewältigen. Sie bieten zudem Anbindungen an analytische Plattformen für komplexe Datenanalysen (z. B. Hadoop und Spark). Das Beispiel macht deutlich, dass

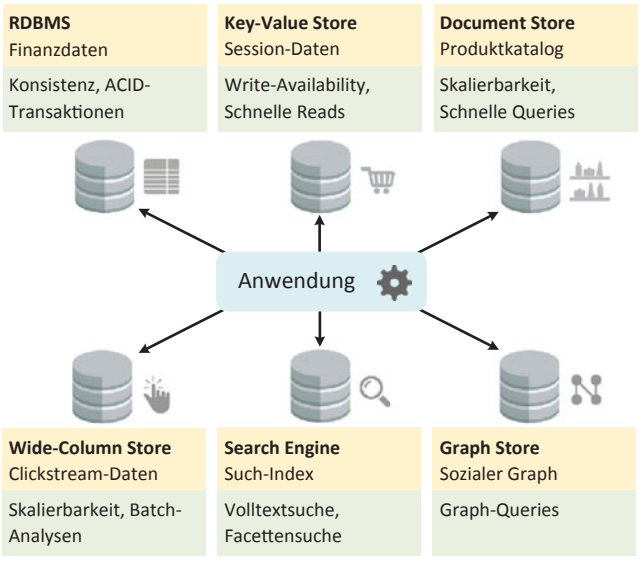

Abb. 1 Beispiel für eine Polyglot-Persistence-Architektur mit verschiedenen Datenbanken, Daten und Anforderungen

F. Gessert (✉) · Norbert Ritter
Universität Hamburg,
Hamburg, Deutschland
E-Mail: gessert@informatik.uni-hamburg.de

bei Polyglot Persistence stets ein Trade-off zwischen erhöhter Komplexität der Entwicklung und Wartung gegenüber besserer Speicherung zu treffen ist.

Zusammenfassend ist Polyglot Persistence also schlicht der Ansatz, für ein Problem stets die am besten geeignete Persistenztechnologie einzusetzen. Im Folgenden geben wir einen Überblick über typische Problem- und korrespondierende Systemklassen. Anschließend diskutieren und kategorisieren wir Umsetzungsstrategien für Polyglot Persistence.

2 Polyglottes Data Management

Die *Produktivität* wird durch Polyglot Persistence erhöht, wenn der *Impedance Mismatch* zwischen Anwendungsdatenstrukturen und dem persistenten Datenmodell minimiert wird. Wenn also beispielsweise Web-Anwendungen mit JSON-basierten REST-Schnittstellen in einer Dokumentendatenbank native JSON-Dokumente persistieren, vereinfacht das den Entwicklungsprozess signifikant.

Leistungsgewinne können vor allem dann erzielt werden, wenn die Persistenz-Anforderungen es erlauben, dass Daten repliziert und partitioniert gespeichert und mit spezialisierten Index- und Sekundärspeicherstrukturen angefragt werden. Wenn zudem die Anwendung weiterhin korrekt arbeitet, obwohl Garantien relaxiert werden (z. B. Konsistenz und Serialisierbarkeit), können die Datenbanksysteme das ausnutzen, um Durchsatz, Latenz und speicherbares Datenvolumen zu optimieren.

2.1 Anforderungen und Systemklassen

Typische *funktionale* Persistenz-Anforderungen sind [7]:

- ACID-Transaktionen mit verschiedenen Isolationsleveln
- Atomare, bedingte und/oder mengenorientierte Updates
- Query-Typen: Punktzugriffe, Scans, Aggregationen, Selektionen, Projektionen, Joins, Subqueries, Map-Reduce, Graph-Queries, Batch-Analysen, Suche
- Partielle und/oder kommutative Änderungsoperationen
- Datenstrukturen: Graphen, Listen, Mengen, Hash-Tabellen, Bäume, Dokumente etc.
- Strukturierte, semistrukturierte oder implizite Schemata
- Semantische Integritätsbedingungen

Zu den *nicht-funktionalen* Anforderungen zählen:

- Durchsatz für Reads, Writes, Queries
- Lese- und Schreiblatenz
- Verfügbarkeit für Writes und/oder Reads bei Netzwerkpartitionen und/oder Serverausfällen
- Skalierbarkeit für Datenmenge, Reads, Writes, Queries
- Konsistenzgarantien: starke Konsistenz, Read-Your-Writes, kausale Konsistenz, Monotonic Reads, Monotonic Writes, Writes follow Reads, Eventual Consistency
- Dauerhaftigkeit von Änderungen
- Elastischer Scale-out und/oder Scale-in

Polyglot Persistence ist zu einem wichtigen Teil dadurch motiviert, dass Negativresultate wie CAP- und FLP-Theorem beweisen, dass kein System alle obigen Anforderungen zugleich erfüllen kann: Starke Konsistenz, Serialisierbarkeit und verteilter Konsens sind mit hoher Verfügbarkeit unvereinbar [3]. Zur Realisierung muss deshalb aus verschiedenen Systemklassen gewählt werden. Die NoSQL-Bewegung[1] ist ein Wegbereiter von Polyglot Persistence, da die Anzahl und Vielseitigkeit verfügbarer Systeme sich dadurch in den letzten Jahren drastisch vergößert hat.

Key-Value Stores beschränken sich auf ein primitives Datenmodell mit einfachen CRUD-Operationen (Create, Read, Update, Delete). Sie sind dadurch leicht *skalierbar* (z. B. Riak, Voldemort, Oracle NoSQL Database, HyperDex) und erzielen hohe *Lese- und Schreibdurchsätze*. In-Memory Key-Value Stores bieten durch Verzicht auf Dauerhaftigkeit besonders *geringe Latenz* (z. B. Redis, Memcached).

Document Stores speichern geschachtelte, denormalisierte Dokumente, typischerweise im JSON-Format. Der Verzicht auf dokumentenübergreifende Queries und Updates macht sie *skalierbar*. Zur Abfrage dienen systemspezifische Ad-hoc Query-Sprachen (z. B. in MongoDB, RethinkDB) oder materialisierte Map-Reduce Views (z. B. in CouchDB, Couchbase). Die nicht-funktionalen Eigenschaften sind systemabhängig. MongoDB garantiert z. B. durch Master-Slave-Replikation starke Konsistenz, während die Multi-Master-Replikation in CouchDB *eventually consistent* ist.

Wide-Column Stores verwenden ein mehrstufiges Datenmodell, bei dem Werte über Column Families, Columns und Timestamps kategorisiert und einem Row-Key zugeordnet sind. Die an Googles BigTable orientierten Systeme HBase, Accumulo und HyperTable garantieren *starke Konsistenz* und implementieren *atomare und bedingte Updates* sowie sortierte *Scans* über den Row-Key. Das auf Dynamo basierende Cassandra übernimmt nur das Datenmodell von BigTable, so dass es je nach Quorumskonfiguration statt Konsistenz *hohe Verfügbarkeit* bietet. Wide-Column Stores arbeiten mit einer „Append-Only"-Organisation des Sekundärspeichers, um hohen *Schreibdurchsatz* und *Dauerhaftigkeit* zu erzielen, und sind durch Partitionierung *skalierbar*.

Graph Stores basieren auf dem Graph-Property-Modell, bei dem Knoten und Kanten eines Graphen mit beliebigen Key-Value-Paaren annotiert werden können. Aufgrund der NP-Vollständigkeit von optimalen Graph-Partitionierungen

[1] Der Neologismus NoSQL – post-hoc häufig als „Not only SQL" [6] interpretiert – ist ein Sammelbegriff für die neuen Systeme.

sind fast alle Systeme (z. B. Neo4j, AllegroGraph) nur repliziert und nicht partitioniert. Sie garantieren deshalb *Skalierbarkeit* nur für Reads und *Konsistenz* statt Verfügbarkeit.

Neben diesen vier Klassen und RDBMS, können ebenso „NewSQL"-Systeme [5] (z. B. VoltDB, Googles Spanner), Cloud-Datenbanken (z. B. Amazon DynamoDB, Azure Tables), Search Engines (z. B. Solr, ElasticSearch), Triple Stores, Data Grids und objektorientierte Datenbanken für Polyglot Persistence eingesetzt werden.

2.2 Herausforderungen

Die zentrale Herausforderung besteht darin, festzustellen, ob ein gegebenes Datenbanksystem eine Menge an Zugriffsmustern und Anforderungen erfüllt. Zwar können einige Performance-Metriken über Benchmarks wie YCSB und TPC experimentell ermittelt werden, viele nicht-funktionale Eigenschaften wie Konsistenz und Skalierbarkeit sind jedoch durch derzeitige Benchmarks nicht abgedeckt [9] und divergieren zudem häufig von dem in der Dokumentation angegebenen Verhalten [3].

In einer Polyglot-Persistence-Architektur sind die Grenzen einer Datenbank auch stets die Grenzen von Transaktionen, Queries und Änderungsoperationen. Wenn deshalb überlappende Anteile der Daten in verschiedenen Datenbanken persistiert und modifiziert werden, wirft dies zwangsläufig Konsistenzprobleme auf. Die Anwendung muss daher die Synchronisation der Daten explizit steuern (z. B. durch periodische ETL-Batch-Jobs) und die Konsistenz auf Anwendungsebene sicherstellen (z. B. durch kommutative oder konvergente Datenstrukturen). Alternativ können Daten disjunkt auf Datenbanken verteilt werden, wodurch sich das Problem auf datenbankübegreifende Anfragen verlagert. Diese Herausforderung ist konzeptionell eng verwandt mit der virtuellen und physischen Datenintegration [5]. Der Unterschied zu Polyglot Persistence liegt primär darin, dass die Datenintegration die Autonomie der Quellsysteme wahrt, während Datenbanken für Polyglot Persistence der Anwendung unterworfen sind und anforderungsgetrieben kombiniert und modifiziert werden.

3 Umsetzung

Um die erhöhte Komplexität der Anwendungsentwicklung und Wartung beherrschbar zu machen, haben sich verschiedene Ansätze etabliert. Wir unterteilen sie in die drei Architekturmuster *applikationskoordinierte polyglotte Persistenz*, *Microservices* und *polyglotte Datenbankdienste*. Als Beispiel diene der Produktkatalog des einleitenden E-Commerce-Szenarios (siehe Abb. 2). Dieser liefert Produktbeschreibungen sowohl über selektierende Queries (z. B. Stichwort-Suche) als auch Top-k-Queries über die Aufrufzahlen – also beliebteste Artikel – aus. Die Anforderung ist daher, dass Aufrufzähler einen hohen Schreibdurchsatz (hochfrequentes Inkrementieren) und Top-k-Queries unterstützen **(1)**, während auf Produkten selektierende Queries und Skalierbarkeit des Datenvolumens möglich sind **(2)**. Diese Anforderungen können durch den Key-Value Store Redis und den Document Store MongoDB erfüllt werden. Redis unterstützt mit seiner Sorted-Set-Datenstruktur ein Mapping von Zählern auf Primärschlüssel von Artikeln. Inkrementierungen und Top-k-Queries werden mit logarithmischer Zeitkomplexität performant In-Memory ausgeführt. MongoDB unterstützt die Speicherung von Artikeln als geschachtelte Dokumente und entsprechende Queries über Attribute der Dokumente. Die Artikel können durch eine Hash-Partitionierung transparent auf viele Knoten verteilt werden, um Skalierbarkeit zu erzielen.

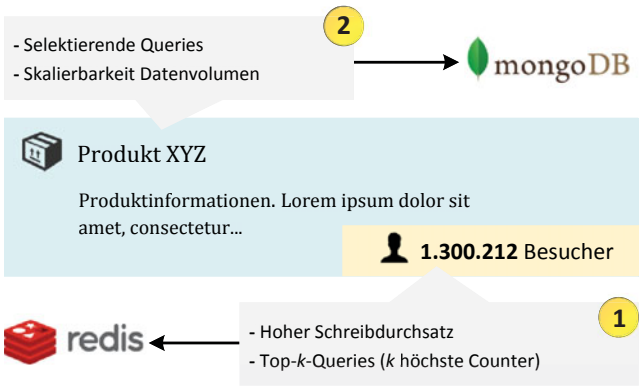

Abb. 2 Polyglot Persistence für einen Produktkatalog

Bei der **applikationskoordinierten polyglotten Persistenz** (siehe Abb. 3) übernimmt der Application Layer der Anwendung (z. B. ein Web-Server) programmatisch die Koordination der polyglotten Persistenz. Dabei folgt die Datenbankzuordnung üblicherweise der Modularisierung der Anwendung. Das erleichtert die Wartung und Entwicklung, da jeweils ein Modul auf die Verwendung einer Datenbank spezialisiert ist und Design-Entscheidungen bei Modellierung und Zugriffsmustern gekapselt sind (lose Kopplung). Diese Separierung kann jedoch aufgeweicht werden: Für den Produktkatalog wäre es sowohl möglich, Zugriffszähler und Artikel getrennt zu modellieren, als auch durch Artikel-Entitäten mit Zählerattribut. Die Abhängigkeit zwischen beiden Datenbanken muss sowohl in Entwicklung als auch operativem Betrieb berücksichtigt werden. Ändert sich beispielsweise das Primärschlüsselformat von Artikeln, muss die Änderung für beide Datenbanken im Code und im Datenbestand übernommen werden. Eine große Chance für die Vereinfachung der Entwicklung sind sogenannte *Object-NoSQL-Mapper* [8], die standardisierte Persistenz-APIs wie JPA auf verschiedene Datenbanken abbilden. Die Mapper sind jedoch derzeit noch dadurch limitiert, dass ihr hohes Abstrak-

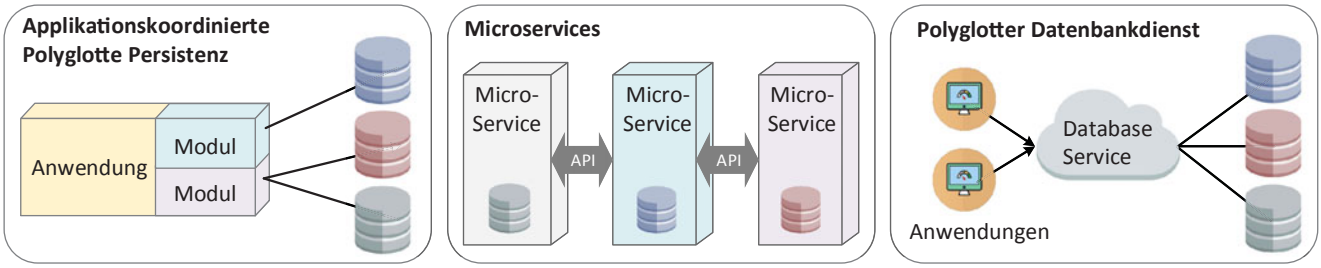

Abb. 3 Architekturstile für polyglotte Persistenz

tionsniveau viele der datenbankspezifischen Vorteile außer Kraft setzt.

Ein praktisches Beispiel für applikationskoordinierte polyglotte Persistenz ist Twitters Persistierung von User-Timelines [1]. Die neusten Tweets der Timeline werden für schnelle Lesezugriffe in einem Redis-Cluster gecacht. Beim Schreiben eines Tweets wird der soziale Graph aus Twitters Graph Store FlockDB abgerufen und der Tweet auf die einzelnen Redis-Timelines verteilt (*Write Fanout*). Als persistenter Fallback für Redis dienen MySQL-Server, die durch die Anwendung partitioniert werden.

Ein Ansatz für eine noch stärke Kapselung von Persistenzentscheidungen sind **Microservices** [4]. Der Microservice-Architekturstil propagiert die Strukturierung von Anwendungen als Sammlung kooperierender, autonomer Dienste, die über leichtgewichtige Schnittstellen wie REST/HTTP miteinander kommunizieren. Der Ansatz wird von Vorreitern wie Netflix erfolgreich in der Praxis eingesetzt. Microservices erlauben es, die Wahl einer Datenbank pro Service zu fällen, und entkoppeln so Entwicklung und Deployment zwischen den Diensten. Technologisch besteht mit IaaS/PaaS-Clouds wie AWS, Container-Plattformen wie Docker und Cluster-Managern wie Kubernetes eine gute Basis für skalierbare, fehlertolerante Microservice-Architekturen. Im Produktkatalog-Beispiel würden zwei separate Microservices jeweils Redis und MongoDB kapseln. Der Redis-Service würde über eine REST-Schnittstelle eine API für das Inkrementieren der Zähler und das Abrufen der beliebtesten Artikel anbieten, während der MongoDB-Service entsprechend eine Abrufschnittstelle für Artikel bereitstellen würde. Der Service, der Endnutzer bedient, müsste lediglich Aufrufe an beide Services delegieren.

Um polyglotte Persistenz für die Anwendung vollständig transparent zu gestalten, dienen **polyglotte Datenbankdienste**. Diese folgen dem Database-as-a-Service (DBaaS) Ansatz und kapseln hinter (oft Cloud-basierten) Service-Schnittstellen die Allokation von Daten und Anfragen auf Datenbanken. Einige NoSQL-Systeme und DBaaS-Provider verwenden diesen Ansatz, um entweder transparente Volltext-Suche anzubieten (Riak, Cassandra), Metadaten konsistent abzuspeichern (BigTable, HBase) oder Objekte und Queries zu cachen (Facebook Tao, Orestes, Baqend) [2]. Bestehende polyglotte Datenbankdienste nutzen statische Regeln zur Datenverteilung: Wenn der Typ der Daten bekannt ist (z. B. User-Objekte oder Dateien), wird regelbasiert eine Datenbank ausgewählt. Die bisher ungelöste Schwierigkeit ist, die Wahl der Datenbank an die Anforderungen und Workloads der Anwendung zu koppeln. Ein möglicher Ansatz ist es, deklarative, von der Anwendung spezifizierte Annotationen an Schemata oder Objekten als Entscheidungskriterium heranzuziehen [7]. Im Beispiel würde also das Zählerfeld des Produkts mit der Durchsatzanforderung annotiert werden, damit der Datenbankdienst autonom eine passende vertikale Partitionierung vornimmt.

4 Ausblick

Polyglot Persistence ist ein vielversprechender Ansatz, um komplexe Data Management-Probleme zu lösen. Um die erhöhte Komplexität der Entwicklung und Wartung zu entschärfen, sind weitere Arbeiten sinnvoll. Sie sollten den Auswahlprozess adressieren, um zu beantworten, wie für gegebene Anforderungen nach objektiv quantifizierbaren Metriken ein passendes System ausgewählt werden kann. Die Standardisierung und Integration der NoSQL-Datenbanken sollte zudem verbessert werden, um Daten zwischen Systemen übertragen und materialisieren zu können und Queries und Transaktionen über Systemgrenzen hinweg auszuführen. Langfristig ist es sehr attraktiv, Konzepte für automatische und semiautomatische Polyglot Persistence zu erarbeiten.

Literatur

1. URL http://infoq.com/presentations/Twitter-Timeline-Scalability
2. Gessert F, Bücklers F, Ritter N (2014) Orestes: a scalable database-as-a-service architecture for low latency. In: Proc. ICDE Workshops. IEEE, S 215–222
3. Kleppmann M (2016) Designing data-intensive applications, 1 edn. O'Reilly
4. Newman S (2015) Building Microservices, 1 Aufl. O'Reilly
5. Rahm E, Saake G, Sattler KU (2015) Verteiltes und Paralleles Datenmanagement. eXamen.press. Springer

6. Sadalage PJ, Fowler M (2012) NoSQL distilled: a brief guide to the emerging world of polyglot persistence. Pearson Education
7. Schaarschmidt M, Gessert F, Ritter N (2015) Towards automated polyglot persistence. In: Proc. BTW 2015, LNI, Bd 241. GI, S 73–82
8. Störl U, Hauf T, Klettke M, Scherzinger S (2015) Schemaless NoSQL data stores - object-NoSQL mappers to the rescue? In: Proc. BTW 2015, LNI, Bd 241. GI, S 579–599
9. Wingerath W, Friedrich S, Gessert F, Ritter N (2015) Who watches the watchmen? On the lack of validation in NoSQL benchmarking. In: Proc. BTW 2015. S 351–360