TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

DOKTORARBEIT

# Datentransformationen in NoSQL-Datenbanken

Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
*Doktor der Ingenieurswissenschaften (Dr.-Ing.)*
genehmigte Dissertation

*von*

M.Sc. Johannes Schildgen

*Dekan des Fachbereichs Informatik:*
Prof. Dr.-Ing. Stefan Deßloch

*Prüfungskommison:*

| | |
|---|---|
| Vorsitzender: | Prof. Dr.-Ing. Reinhard Gotzhein |
| Berichterstatter: | Prof. Dr.-Ing. Stefan Deßloch |
| | Prof. Dr.-Ing. Wolfgang Lehner |

*Datum der wissenschaftlichen Aussprache:*
25. September 2017

D 386

# Zusammenfassung

NoSQL-Datenbanken werden als Alternative zu klassischen relationalen Datenbanksystemen eingesetzt, um die Herausforderungen zu meistern, die „Big Data“ mit sich bringt. Big Data wird über die drei V definiert: Es sind große Datenmengen („Volume“), die schnell anwachsen („Velocity“) und heterogene Strukturen haben („Variety“). NoSQL-Datenbanken besitzen zudem meist nur sehr einfache Anfragemethoden. Um auch komplexe Datenanalysen durchzuführen, kommen meist Datenverarbeitungsframeworks wie MapReduce, Spark oder Flink zum Einsatz. Diese sind jedoch schwieriger in der Benutzung als SQL oder andere Anfragesprachen.

In dieser Arbeit wird die Datentransformationssprache *NotaQL* vorgestellt. Die Sprache verfolgt drei Ziele. Erstens ist sie mächtig, einfach zu erlernen und ermöglicht komplexe Transformationen in wenigen Code-Zeilen. Zweitens ist die Sprache unabhängig von einem speziellen Datenbankmanagementsystem oder einem Datenmodell. Daten können von einem System in ein anderes transformiert und Datenmodelle dementsprechend ineinander überführt werden. Drittens ist es möglich, NotaQL-Skripte auf verschiedene Arten auszuführen, sei es mittels eines Datenverarbeitsungsframeworks oder über die Abbildung in eine andere Sprache. Typische Datentransformationen werden periodisch ausgeführt, um bei sich ändernden Basisdaten die Ergebnisse aktuell zu halten. Für solche Transformationen werden in dieser Arbeit verschiedene inkrementellen Ansätze miteinander verglichen, die es möglich machen, dass NotaQL-Transformationen die vorherigen Ergebnisse wiederbenutzen und Änderungen seit der letzten Berechnung darauf anwenden können. Die NotaQL-Plattform unterstützt verschiedene inkrementelle und nicht-inkrementelle Ausführungsarten und beinhaltet eine intelligente Advisor-Komponente, um Transformationen stets auf die bestmögliche Art auszuführen. Die vorgestellte Sprache ist optimiert für die gebräuchlichen NoSQL-Datenbanken, also Key-Value-Stores, Wide-Column-Stores, Dokumenten- und Graph-Datenbanken. Das mächtige und erweiterbare Datenmodell der Sprache erlaubt die Nutzung von Arrays, verschachtelten Objekten und Beziehungen zwischen Objekten. Darüber hinaus kann NotaQL aber nicht nur auf NoSQL-Datenbanken, sondern auch auf relationalen Datenbanken, Dateiformaten, Diensten und Datenströmen eingesetzt werden. Stößt ein Benutzer an das Limit, sind Kopplungen zu Programmiersprachen und existierenden Anwendungen mittels der Entwicklung benutzerdefinierter Funktionen und Engines möglich. Die Anwendungsmöglichkeiten von NotaQL sind Datentransformationen jeglicher Art, von Big-Data-Analysen und Polyglot-Persistence-Anwendungen bis hin zu Datenmigrationen und -integrationen.

# Abtract

NoSQL databases are a new family of databases systems that do not necessarily follow the relational model, they have simple access methods, and they are made to handle Big Data. Characterized with the *3 Vs*, Big Data is typically defined in terms of Volume, Variety, and Velocity. Data-processing frameworks like MapReduce, Spark, or Flink are often used to perform complex data analyses due to the simple access methods of NoSQL databases. These systems are powerfull but more difficult to use than query languages such as SQL.

In this thesis, a data-transformation language called *NotaQL* is presented. This language follows three objectives. Firstly, it is easy to learn, concise and powerful. Secondly, it is independent of the actual underlying database system and its data model. Data from different systems can be read and written to perform a cross-system transformation. Thirdly, NotaQL transformations can be executed in various ways. There are approaches that make use of MapReduce, Spark, or other tools and languages. As typical data transformations are executed periodically to bring the transformation results up to date, this thesis shows and evaluates different approaches for incremental transformations by reusing a previous result and analyzing only the changes since the former computation to produce the new result. Different incremental and non-incremental approaches are supported by the NotaQL transformation platform. An intelligent advisor component can be used to always execute a transformation in the best possible way.

NotaQL supports all typical forms of NoSQL databases: key-value stores, wide-column stores, document stores and graph databases. Its powerful and extensible data model allows for handling arrays, complex objects, and relationships between objects. However, NotaQL is not limited to NoSQL databases; it can also be used to analyze and transform data from relational databases, files, services, and data streams. When a user reaches the limits of NotaQL, user-defined functions and engines can be developed to extend the language and connect transformations to other systems.

The fields of application of the presented language are data transformations of any kind: from Big-Data analytics and polyglot-persistence applications to data migrations and information integration.

# Danksagung

Ich möchte einen besonderen Dank meinem Betreuer und Doktorvater Prof. Stefan Deßloch aussprechen. Er stand von Anfang an voll und ganz hinter meiner Forschung und zeigte mir stets mit wertvollen Ratschlägen den richtigen Weg. Des Weiteren bin ich ihm dankbar dafür, dass ich neben meiner Forschung besonders viel Lehrerfahrung sammeln durfte. Auch den beiden anderen Professoren des Lehrgebiets Informationssysteme, Prof. Theo Härder und Prof. Sebastian Michel danke ich dafür, dass sie mir ihr Wissen an Datenbanken weitergegeben haben und ich mich auch in deren Arbeitsgruppen in die Lehre mit einbringen durfte.

Ein großer Dank geht an zwei weitere Professoren, und zwar zum einen Prof. Reinhard Gotzhein, der sich bereiterklärt hat, den Vorsitz des Promotionskommittees zu übernehmen und an Prof. Wolfgang Lehner, den zweiten Gutachter dieser Doktorarbeit.

Ohne meine Kollegen am Lehrstuhl wäre meine Arbeit nur halb so angenehm gewesen. Daher bedanke ich mich für unterhaltsame Mittagspausen in der Mensa und aufregende Partien am Kickertisch bei Caetano Sauer, Daniel Schall, Evica Milchevski, Heike Neu, Kiril Panev, Koninika Pal, Manuel Hoffmann, Steffen Reithermann, Weiping Qu und Yong Hu. Auch an die wissenschaftlichen Mitarbeiter des Lehrgebiets zu meiner Studienzeit geht ein großer Dank, insbesondere Karsten Schmidt, Sebastian Bächle und Thomas Jörg. Ihr habt mir vorgemacht, wie man erfolgreich promoviert, lehrt, forscht und betreut.

In meinen viereinhalb Jahren in der Arbeitsgruppe Heterogene Informationssysteme hatte ich das große Glück, ständig von hervorragenden Studenten umgeben zu sein. Die Betreuung deren Bachelor- und Masterarbeiten war nicht nur eine große Freunde für mich, sondern jede der Arbeiten leistete auch einen großen Beitrag für diese Doktorarbeit. Danke Fabian Süß, Florian Haubold, Jan Adamczyk, Jonathan Priebe, Manuel Hoffmann, Marc Schäfer, Michael Emde, Nico Schäfer, Peter Brucker, Philipp Thau, Sougata Bhattacharjee, Stefan Braun, Thomas Lottermann, Tobias Hamann, Voichita Droanca und Yannick Krück!

Zu guter Letzt möchte ich meiner Familie und ganz besonders Inna danken – dafür, dass ich euch habe und dass ihr für mich da seid.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# 1 Einleitung

Der Begriff des *Daten-Managements* hat sich im Laufe des letzten Jahrzehnts und mit dem Aufkommen von Web-Diensten wie sozialen Netzwerken und Online-Shops drastisch verändert. Während in klassischen betrieblichen Informationssystemen in der Regel relationale Datenbankmanagementsysteme eingesetzt wurden, um strukturierte Daten zu speichern, finden heutzutage vermehrt *NoSQL-Datenbanksysteme* [Edl+10] Verwendung. Diese Systeme bieten nicht nur Möglichkeiten, um *große Datenmengen* verteilt zu speichern, sondern bringen auch eine gewisse *Schema-Flexibilität* mit sich. Ebendiese flexiblen Schemata erlauben die Speicherung heterogener Daten. Weder muss zu Beginn ein Datenschema definiert werden, noch müssen die zu speichernden Datensätze zueinander homogen sein.

In relationalen Datenbanken bietet die *Structured Query Language (SQL)* Möglichkeiten zur Definition von Tabellen samt ihrer Spalten und deren Datentypen. In diesen Tabellen herrscht *horizontale und vertikale Homogenität* [Wie15, S. 35]. Horizontal bedeutet hier, dass jede Zeile einer Tabelle die gleichen Spalten besitzt. Nullwerte machen es zwar möglich, einzelne Spalten wertfrei zu lassen, belegen jedoch meistens trotzdem Speicher und müssen organisiert werden. Für die einzelnen Spalten einer Tabelle bedeutet die vertikale Homogenität, dass sich in einer Spalte über die komplette Tabelle hinweg Daten ein und des selben Typs befinden.

Die Schema-Flexibilität und die teilweise sehr mächtigen Datenmodelle von NoSQL-Datenbanken machen es möglich, dass Anwendungen ohne vorherige Datendefinitionsphasen beliebig strukturierte Objekte in die Datenbank legen und diese mittels einfachen Zugriffsmethoden lesen können. Da Datensätze genau so gespeichert werden können, wie die Anwendung sie benötigt - also zum Beispiel ein Blogbeitrag-Objekt mit einem Listenattribut, welches alle Kommentare zu diesem Beitrag und deren Verfasser enthält -, sind die in verteilten Datenbanken sehr kostspieligen Verbundoperationen nicht mehr vonnöten. In NoSQL-Datenbanken findet man dementsprechend nicht nur spezielle Datenmodelle, sondern auch neuartige und meist sehr schlichte Anfragemechanismen.

Es zeigt sich, dass die Heterogenität, die in der Datenbankforschung bereits seit langer Zeit eine große Herausforderung bei der Datenintegration

und -migration darstellt [LN07], im Zeitalter von NoSQL auf neue Probleme stößt. Dazu zählt auch der Trend zur *polyglotten Persistenz* [SF12]. Sogenannte Polyglot-Persistence-Anwendungen greifen nicht auf eine einzige Datenbank zu, sondern verbinden sich mit verschiedenen Datenbanken, Diensten und externen Anwendungen, um Daten zu lesen, zu analysieren und miteinander zu verbinden. Der Grund dafür ist, dass es keine Einheitslösung gibt, sondern jedes System gewisse Vor- und Nachteile hat. Daher werden Plattformen, Sprachen und Algorithmen zur Auflösung der in diesen Anwendungen vorliegenden Heterogenität benötigt. In dieser Arbeit stellen wir als Lösungsansatz für diese neuen Herausforderungen bei der Datentransformation die Sprache NotaQL sowie eine dazugehörige Plattform vor.

## 1.1 Big Data

Bei der Diskussion um die Definition des Begriffes *Big Data* startete man mit den sogenannten *drei Vs*: „Volume, Velocity, Variety“ [Lan01]. Diese Begriffe verdeutlichen den weiter oben beschriebenen Trend, dass Datenmengen größer werden, rasant anwachsen und heterogene Strukturen vorweisen. Im Laufe der Zeit kamen zu der Big-Data-Definition weitere Vs hinzu, darunter ist eines die „Veracity“ [Zik+12]. Während die ersten drei Begriffe die Eigenschaften der Daten aus technischer Sicht betonen, bedeutet die Veraciy (*Richtigkeit*), dass wir nicht mehr zwangsweise von einer *wahren Welt* sprechen können. In Systemen, in denen vom Benutzer generierte Daten gespeichert und verbreitet werden, kommen falsche Informationen ebenso vor wie Widersprüche, fehlerhafte Übersetzungen sowie Nutzer-Reaktionen auf fälschlich interpretierte Nachrichten.

Die Herausforderungen bei der *Analyse von Big Data* lassen sich aus den soeben dargestellten Vs ableiten: Da wir es mit großen Datenmengen zu tun haben, müssen Systeme in der Lage sein, diese enormen Mengen abspeichern und sie schnell und effizient analysieren zu können. Während sich reine Platzprobleme leicht mit Aufrüstungen und dem Zuschalten weiterer Festplatten und Solid-State-Drives lösen lassen, ist eine solche Aufrüstung - das sogenannte *Scale-up* - auf Prozessor- und Arbeitsspeicherebene begrenzt und zudem meist sehr kostspielig. Stattdessen ist es mittels *Scale-out* günstiger möglich, sowohl Speicherplatz als auch Rechenleistung zu erhöhen, indem weitere Rechner in ein Rechencluster eingefügt werden. Verteilte Datenbanken und verteilte Dateisysteme setzen dabei meist auf *Hash-Partitionierung* oder *Bereichspartitionierung* [DG92], um Daten basierend auf ihren Identifikatoren oder Attributswerten auf den jeweiligen Rechnern zu verteilen.

## 1.2 Motivation

Die Zeiten, in denen Anwendungen „nur SQL“ mit einer Datenbank sprachen, sind vorbei. Der Begriff *NoSQL* bedeutet so viel wie „Mehr als nur SQL“ (Not only SQL) und findet in immer mehr Unternehmen und Anwendungen Bestätigung. Big Data und agile Softwaremethoden sind nur zwei der Gründe dafür, dass vermehrt nicht-relationale Datenbanksysteme zum Einsatz kommen. Aber auch wegen der Einsetzbarkeit im Cloud-Computing und nicht zuletzt aus Gründen der Kostenersparnis entscheiden sich immer mehr Unternehmen für NoSQL. Anders als bei relationalen Datenbanksystemen gibt es bei NoSQL-Datenbanken keine einheitlichen Datenmodelle und Anfragesprachen. Manche Systeme haben ein extrem schlichtes Datenmodell, sind aber dafür besonders skalierbar und schnell. Andere Systeme erlauben eine komplexe Datenmodellierung als Graph-Struktur und bieten spezielle Sprachen zur Suche und zum Traviersieren über Knoten und Kanten. Weitere Systeme unterstützen und spezialisieren sich auf Textdaten, Benutzerkommentare, Bilder, Videos, Geo-Daten, Datenströme oder soziale Netzwerke. Meist gibt es jedoch für eine Anwendung keine Einheitslösung, sondern es erfordert die Kombination verschiedener Systeme. Die vorliegende Arbeit beschäftigt sich mit ebensolchen Polyglot-Persistence-Anwendungen und beschreibt Ansätze, um Big-Data-Analysen durchzuführen, Daten zwischen den verschiedenen Systemen zu transferieren, Systeme durch andere zu ersetzen, Daten zu migrieren und sie zu modifizieren.

## 1.3 Wissenschaftlicher Beitrag

In dieser Arbeit werden NoSQL-Datenbanksysteme analysiert und neuartige Probleme aufgezeigt, die diese Systeme mit sich bringen. In der Literatur existieren Ansätze, die sich mit der Überwindung von Heterogenität beschäftigen, beispielsweise mittels Wrappern, die einen einheitlichen Zugriff auf unterschiedliche Datenquellen ermöglichen. Es wird untersucht, in wie fern sich die meist aus relationalen Datenbanken stammenden existierenden Techniken auf die NoSQL-Welt übertragen lassen und welche Herausforderungen es noch zu meistern gilt. Aktuell existieren keine Ansätze, die das Ziel verfolgen, Heterogenität in der vollständigen NoSQL-Landschaft zu überwinden. Stattdessen entstehen immer mehr Ansätze, die lediglich auf einer speziellen Klasse von Systemen anwendbar ist.

Der Hauptteil der Arbeit besteht aus der Vorstellung der Sprache *NotaQL*. Anhand dieser Sprache wird exemplarisch gezeigt, wie Datentransformationen auf flexiblen Schemata und unterschiedlichen Datenmodellen erfolgen können. Mittels *Datenverarbeitungsframeworks* können viele der aufgezeigten Probleme zwar bereits gelöst werden, jedoch erfordern diese einen hohen Aufwand bei der Einarbeitung, Entwicklung von Transformationsprogrammen und der Wartung. Aus diesem Grund werden verschiedene Ansätze vorgestellt, welche auf existierenden Frameworks basieren, um NotaQL-Transformationen effizient auszuführen.

Der zweite große Teil der Arbeit beschäftigt sich mit *inkrementellen Datentransformationen*. Solche Art von Berechnungen kommen häufig im *Data-Warehousing* zum Einsatz, da sie deutlich schneller durchzuführen sind als

Komplettberechnungen. Dazu müssen periodisch ausgeführten Transformationsprogrammen die Änderungen seit einer vorherigen Ausführung vorliegen, damit diese basierend auf dem alten Ergebnis und ebendiesen Änderungen das neue Ergebnis berechnen können. In dieser Arbeit werden existierende Techniken zur Änderungserfassung und zur inkrementellen Wartbarkeit untersucht und auf NoSQL-Datenbanken sowie die Sprache NotaQL übertragen. Eine NotaQL-Plattform vereinigt verschiedene inkrementelle und nicht-inkrementelle Ausführungsarten und wählt mit ihrer Advisor-Komponente automatisch die als schnellste eingeschätzte Art.

Zusammengefasst sind die Beiträge der vorliegenden Arbeit wie folgt:

- Eine Datentransformationssprache, die für die Schema-Flexibilität und die Datenmodelle von NoSQL-Datenbanken optimiert ist.
- Eine Ausführungsplattform, welche in der Lage ist, Datentransformationen virtuell und materialisiert auszuführen.
- Inkrementelle Berechnungsmethoden für die effiziente Ausführung sich periodisch wiederholender Transformationen.
- Eine Analyse über die Anwendbarkeit der Sprache bei der Big-Data-Analyse, bei Polyglott-Persistence-Anwendungen, Datenintegrationen und Schema-Evolutionen.

## 1.4 Grenzen

Die Entwicklung der schnellsten Datentransformationsplattform ist nicht Ziel dieser Arbeit. Stattdessen wird die Machbarkeit einer solchen Plattform gezeigt und Lösungen zur Überwindung von Heterogenität und zur Unterstützung flexibler Schemata vorgestellt. Die händische Entwicklung eines für einen bestimmten Anwendungszweck gedachtes Transformationsprogramms wird in der Regel zu einer effizienteren Ausführung führen als mit dem in dieser Arbeit vorgestelltem Ansatz. Auf der anderen Seite sind solche Programme aufwändiger zu entwickeln und können nicht einfach auf andersartige Systeme und Datenmengen übertragen werden.

Des Weiteren soll nicht Ziel der Arbeit sein, NotaQL als die benutzerfreundlichste und mächtigste Transformationssprache anzupreisen. Die beiden genannten Aspekte der Benutzerfreundlichkeit und Mächtigkeit waren zwar einige der Ziele bei der Entwicklung der Sprache, sind jedoch stets ausbaufähig. Das zeigt sich auch daran, dass sich die Sprache im Laufe der vergangenen Jahre ständig verändert und weiterentwickelt hat. Zum Teil ist dies mit gestiegenen Anforderungen wegen komplexeren Transformationen oder zu unterstützenden Datenmodellen zu begründen, teilweise wurde aber auch lediglich erkannt, dass gewisse Sprachkonstrukte eleganter als andere sind. Diese Erfahrungen, Ideen und Lösungen führten zur Sprache NotaQL, wie sie aktuell ist. Zum einen bedeutet das, dass weitere Iterationen die Sprache NotaQL weiter verändern, zum anderen kann bei der Entwicklung neuer Sprachen auf die hier vorgestellten Ansätze zurückgegriffen werden.

Um Missverständnisse zu vermeiden, soll die folgende Aufzählung Themen zeigen, die zwar mit der vorliegenden Arbeit verwandt sind, jedoch als orthogonal oder außerhalb des Fokus angesehen werden:

- *Performanz*: Die Performanz hängt vom verwendeten Datenbanksystem, dem zugrundeliegenden Transformationsframework, der Hardware und mehr ab.
- *Optimierungen*: Die Ausführung von NotaQL-Transformationen und die Implementierung der Operatoren kann stets optimiert werden; diese sind vor allem auch von der Ausführungsart abhängig. Diese Arbeit soll jedoch generisch und nicht auf eine bestimmte Ausführungsart spezialisiert sein.
- *Transaktionsbegriff*: Fragestellungen über die atomare und isolierte Ausführung sowie das Verhalten im Fehlerfall wurden ausgeblendet, da diese Themen in der Literatur bereits detailliert behandelt wurden und ein Übertragen auf die generische NotaQL-Plattform den Rahmen dieser Arbeit sprengen würde.
- *Hardware- und Netzwerk-Charakteristika*: Die vorgestellten Ansätze profitieren von Hardware-Optimierungen und verteilten Speicherungen, setzen diese aber nicht voraus.
- *Anfragesprachen*: NotaQL ist keine Anfragesprache, daher werden zur Interaktion zwischen der Anwendung und der Datenbank, in welcher sich Ein- und Ausgabedaten von Transformationen befinden, weiterhin benötigt.

## 1.5 Überblick

Im zweiten Kapitel dieser Arbeit werden Grundlagen zu NoSQL-Datenbanken und Datenverarbeitungsframeworks präsentiert. Auch Techniken zur inkrementellen Wartbarkeit und Herausforderungen bei der Integration von Daten sind Teil dieses Kapitels. Danach werden in Kapitel 3 existierende Transformationssprachen für NoSQL untersucht und miteinander verglichen.

Der Hauptteil der Arbeit, welcher die Vorstellung der Sprache NotaQL ist, befindet sich in Kapitel 4. Nach einer Demonstration von Sprachkonstrukten für einfache Attribut-Mappings, Selektionen und Aggregationen folgen komplexere Sprachelemente zur Transformation von Graphen und Datenströmen.

Die im zweiten Kapitel vorgestellten Datentransformationsframeworks dienen im fünften Kapitel als Basis für verschiedene Ausführungsarten von NotaQL. Anhand mehrerer solcher Frameworks, aber auch über alternative Wege mittels Übersetzungen in eine andere Sprachen oder über virtuelle Transformationen werden unterschiedliche Ansätze präsentiert und verglichen. Um die genannten Ansätze um eine inkrementelle Wartbarkeit zu erweitern, folgen in Kapitel 6 Implementierungsmöglichkeiten für die ebenfalls im zweiten Kapitel vorgestellten Änderungserfassungsmethoden.

In Kapitel 7 werden einige Anwendungsmöglichkeiten präsentiert, für welche sich die Sprache NotaQL einsetzen lässt. Zum Ende in Kapitel 8 erfolgt eine Zusammenfassung der Arbeit.

# 2 Grundlagen

In diesem Kapitel werden die Grundlagen zu dieser Arbeit präsentiert. Dabei wird zunächst mit der Definition und Vorstellung von NoSQL-Datenbanksystemen begonnen. Da diese Systeme Daten meist auf eine verteilte Art speichern und gleichzeitig aber nur sehr einfache Zugriffsmethoden bieten, kommen für komplexe Analysezwecke meist Datenverarbeitungsframeworks wie MapReduce oder Spark zum Einsatz, welche weiter unten in diesem Kapitel vorgestellt werden. Im Anschluss daran folgt eine Präsentation von Ansätzen zur inkrementellen Berechenbarkeit. Diese Techniken sind vonnöten, um langdauerende und sich regelmäßig wiederholende Datenanalysen zu beschleunigen. Diese Arbeit beschäftigt sich aber nicht nur mit Datenanalysen, sondern mit Transformationen vielfältiger Art. Dazu zählen auch Datenintegrationen, also der Transfer von Daten zwischen verschiedenen Systemen. Mit dem Thema der Datenintegration wird dieses Grundlagen-Kapitel abgeschlossen.

## 2.1 NoSQL-Datenbanken

Eine Definition für NoSQL-Datenbanksysteme ist in [Edl+10, S. 2] zu finden. Dort heißt es, dass einige - nicht zwangsläufig alle - der nachfolgenden Kriterien erfüllt sein müssen: Ein nicht-relationales Datenmodell, verteilte Speicherung, Quelloffenheit, Schemaflexibilität, die Unterstützung für Datenreplikation, eine einfache API und Flexibilität bezüglich der ACID-Einhaltung. Da es mittlerweile weit über einhundert Systeme gibt, die diese Kriterien mehr oder weniger erfüllen, erfolgt eine Klassifizierung meist in die folgenden vier Kategorien:

- Key-Value-Stores,
- Wide-Column-Stores,
- Dokumentendatenbanken,
- Graphdatenbanken.

| | Klasse | Datenquelle | Datensatz | Attribut | ID |
|---|---|---|---|---|---|
| **DB2** | Rel. DBMS | Tabelle | Zeile | Spalte | Primärschl. |
| **Redis** | Key-Value-Store | Datenbank | Schlüssel-Wert-Paar | Wert | Schlüssel |
| **HBase** | Wide-Column-S. | Tabelle | Zeile | Spalte | Row-ID |
| **MongoDB** | Dokumenten-DB | Kollektion | Dokument | Feld | _id |
| **Neo4J** | Graphdatenbank | Graph | Knoten | Property | ID |
| **CSV-Datei** | CSV | Datei | Zeile | Wert | — |

**Tabelle 2.1:** Terminologien verschiedener DBMS

Es existieren zwar auch Systeme, die in keine dieser Kategorien fallen, nichtsdestotrotz möchten wir im Rahmen dieser Arbeit den Fokus auf diese vier Kategorien legen. Die meisten Konzepte sind nämlich auf auch andere Datenbanksysteme übertragbar. Beispielsweise haben Objektdatenbanken aufgrund der Referenzen gewisse Ähnlichkeiten zu Graphdatenbanken, und XML-Datenbanken ähneln aufgrund ihrer hierarchischen Baumstruktur Dokumentendatenbanken. Konzentrieren wir uns zunächst auf die ersten drei genannten Klassen von NoSQL-Datenbanken und gehen dann separat auf Graphdatenbanken ein. Key-Value-Stores, Wide-Column-Stores und Dokumentendatenbanken haben nämlich die Eigenschaft, dass sie Datensätze in einer Form speichern, die eine gewisse *Gesamtheit* (engl. *aggregate*) unterstützt [SF12]. Würde man Blogbeiträge und deren Kommentare in einer relationalen Datenbank speichern, bestände diese aus zwei Tabellen, die zum Anfragezeitpunkt erst mit einem Join verbunden werden müssen. Eine gesamtheitsorientierte NoSQL-Datenbank würde den Blogbeitrag samt Kommentaren als eine Einheit speichern. Zusätzlich gilt für diese Systeme, dass jeder Datensatz einen Identifikator besitzt. Über welchen Geltungsbereich dieser eindeutig definiert sein muss, unterscheidet sich von Klasse zu Klasse.

Da es keine einheitliche klassenübergreifende Terminologie gibt, werden in dieser Arbeit in der Regel die allgemeinen Begriffe *Datenquelle*, *Datensatz*, *Attribut*, *Wert* und *ID* verwendet. Bei relationalen Datenbanken würde man beispielsweise von einer Tabelle als Datenquelle sprechen. Ein Datensatz ist eine Zeile, ein Attribut eine Spalte und die ID einer Zeile steht in denjenigen Spalten, über die man den Primärschlüssel definiert. Tabelle 2.1 zeigt eine Gegenüberstellung der Terminologien verschiedener Datenbankmanagementsysteme [SD16].

### Key-Value-Stores

Ein zunächst sehr einfaches Datenmodell haben die Key-Value-Stores, da sie lediglich Schlüssel-Wert-Paare in der Datenbank ablegen. Man kann sich die Datenbank als eine Tabelle mit nur zwei Spalten (*Schlüssel*, *Wert*) vorstellen, wobei der Wert von einem beliebigen Typ sein kann. Je nach Implementierung gibt es global nur eine einzige Datenbank, in der alle Schlüssel eindeutig sein müssen, oder es gibt mehrere Datenbanken, die als eigenständig angesehen werden können. Im ersten Fall ist das Datenbanksystem eine Art *Map*, die zu jedem Schlüssel einen Wert zuordnet:

$$Schlüssel \rightarrow Wert$$

Im zweiten Fall existiert für jede *Datenbank* eine eigene Map:

$$Datenbank \rightarrow (Schlüssel \rightarrow Wert)$$

Der Key-Value-Store *Riak* [Bas15] ist ein Vertreter für diesen zweiten Typ. Die Datenbanken werden in Riak *Buckets* genannt. Beim Lese- und Schreibzugriff ist stets anzugeben, auf welchem Bucket man arbeiten möchte. Im Datenbanksystem *Redis* [Red] gibt es zwar auch die Unterteilung in verschiedene Datenbanken, aber dort wird üblicherweise im Gegensatz zu Riak Daten verschiedenen Typs in einer einzigen Datenbank zusammengefasst. Während Buckets bei Riak Namen wie „Produkte“ oder „Kunden“ haben können, sind Datenbanken in Redis lediglich durchnummeriert. Sie werden also eher verwendet, um die Daten von unabhängigen Anwendungen voneinander zu separieren. Eine Anwendung arbeitet dann meist nur auf einer einzigen Datenbank, deren Nummer direkt nach dem Verbindungsaufbau angegeben wird. Innerhalb dieser Datenbank werden unterschiedliche Entitätsmengen meist mittels Schlüsselpräfixen voneinander separiert, z. B. `produkt/812`.

Der Zugriff auf Schlüssel-Wert-Paare erfolgt ähnlich wie bei Maps mit den Methoden `GET` und `PUT`. `GET` nimmt einen Schlüssel entgegen und liefert den dazugehörigen Wert (oder `null`). `PUT` übergibt man ein Schlüssel-Wert-Paar, um es zu speichern. Die `PUT`-Operation bietet ein sogenanntes *Upsert*-Verhalten, welches eine Mischung aus *Insert* und *Update* darstellt. Das bedeutet, dass der Wert eines Schlüssels geändert wird, wenn der Schlüssel bereits vorhanden ist. Andernfalls erfolgt ein Einfügen.

Die meisten Key-Value-Stores unterstützen eine Vielzahl von Datentypen für die zu speichernden Werte. Neben primitiven Typen wie Zahlen und Zeichenketten werden auch große binäre Objekte wie Bilder oder Videos unterstützt. Aber auch komplexe Typen wie Listen, geordnete und ungeordnete (Multi-)Mengen und Maps können eingesetzt werden und bieten passende Zugriffsmethoden.

Das folgende Beispiel zeigt, wie mit Redis-Kommandos auf Schlüssel-Wert-Paare lesend und schreibend zugegriffen werden kann:

```
SET counter 9
INCR counter
GET counter                              // liefert 10
RPUSH pers:1:projekte "DB Projekt"
HSET pers:1 vorname "Rita"
HGET pers:1 vorname                    // liefert Rita
```

Im Beispiel wurde ein Counter-Wert gesetzt, um eins erhöht und anschließend wieder gelesen. An das rechte Ende einer `projekte`-Liste wurde ein neues Projekt angefügt und in einer Hash-Map bekommt die Person mit dem Schlüssel `pers:1` an der Stelle `vorname` den Wert `Rita`. Der letzte Befehlt gibt schließlich ebendiesen Vornamen aus.

### Wide-Column-Stores

Das Datenmodell von Wide-Column-Stores gleicht dem der Key-Value-Stores, wenn als Werte die im vorherigen Beispiel zu sehende Hash-Map verwendet wird. Es erlaubt also die Speicherung von Datensätzen, die eine ID besitzen sowie beliebige Attribut-Wert-Paare. Genau wie bei relationalen Datenbanken spricht man bei Wide-Column-Stores auch von *Tabellen*, die aus *Zeilen* und *Spalten* bestehen. Die Unterschiede liegen jedoch in der Schemafreiheit, da jede Zeile beliebige Spalten haben kann, die nicht vorher definiert werden müssen, sowie in der flexiblen Verwendung von Datentypen. Im Wide-Colum-Store *HBase* [Apab] wird der Einfachheit halber nur der Datentyp Byte-Array unterstützt. Jegliche Datenwerte müssen also auf Anwendungsseite vor der Speicherung in einen Byte-Array gewandelt und beim Lesen wieder rückkonvertiert werden.

Das vereinfachte Datenmodell eines Wide-Column-Stores lässt sich wie folgt darstellen:

$$Tabelle \rightarrow (Row\text{-}ID \rightarrow (Spalte \rightarrow Wert))$$

Eine bestimmte Spalte innerhalb einer Zeile einer Tabelle nennen wir auch *Zelle*. Wenn das Datenbankmanagementsystem Versionierung unterstützt, wird zu jeder Zelle eine Historie an Werten gespeichert, üblicherweise zusammen mit den Zeitstempeln des Schreibens:

$$Tabelle \rightarrow (Row\text{-}ID \rightarrow (Spalte \rightarrow (Version \rightarrow Wert)))$$

Der Datenzugriff erfolgt auch hier wieder mit den Kommandos `GET` und `PUT`. Das folgende Beispiel gilt für das Datenbankmanagementsystem HBase:

```
PUT 'pers', 'P1', 'info:vorname', 'Rita'
GET 'pers', 'P1'
SCAN 'pers'
```

Das erste Kommando fügt eine Zeile in die Tabelle `pers` mit der Row-ID `P1` sowie einer Vornamen-Spalte hinzu. HBase kategorisiert Spalten in sogenannte *Spaltenfamilien*. Beim Anlegen einer Tabelle müssen die Spaltenfamilien festgelegt werden. Diese dienen zur Aufteilung unterschiedlicher Anwendungskonzepte innerhalb einer Zeile. Im genannten Beispiel wird lediglich die Spaltenfamilie `info` verwendet, in der die Spalte `info:vorname` Platz findet. Informationen über Profilaufrufstatistiken oder Beziehungen zwischen Personen könnten in anderen Spaltenfamilien abgelegt werden. Das `GET`-Kommando liefert die Zeile mit der gegebenen Row-ID samt aller ihrer Spalten. Über weitere Parameter kann eine Projektion nach Spaltenfamilien oder einzelnen Spalten erfolgen. Das `SCAN`-Kommando liefert einen Cursor, um die Tabelle zeilenweise zu lesen. Auch hier ist der Einsatz von Filtern möglich, oder aber die Angabe eines Bereiches, in dem die Row-IDs zu liegen haben. Letzteres ist aufgrund der in HBase verwendeten Bereichspartitionierung sehr effizient ausführbar.

### Dokumentendatenbanken

Während das Datenmodell in den zwei bisher genannten Klassen relativ einfach war, sind in Dokumentendatenbanken komplexe hierarchische Strukturen möglich. Das Datenmodell erscheint jedoch zunächst sehr simpel:

$$Datenbank \rightarrow (Kollektion \rightarrow Dokument*)$$

Eine *Kollektion*, die sich in einer bestimmten *Datenbank* befindet, beinhaltet eine Menge von *Dokumenten*. Die Komplexität kommt erst durch den möglichen Aufbau dieser Dokumente zustande. Es gilt, dass die ID stets ein Teil des Dokuments ist, also - anders als bei den oben vorgestellten Systemen - nicht separat betrachtet wird. Zugriffe sind nicht nur über die ID möglich, sondern es werden - ähnlich wie bei SQL - beliebige Selektionen auf Sekundärattributen unterstützt. Zur effizienten Suche unterstützen Dokumentendatenbanken die Definition von Indexen auf ebendiesen Attributen. Ein Index auf der Dokumenten-ID ist üblicherweise immer vorhanden.

Dokumente haben objektähnliche Strukturen. Sie beinhalten *Felder*, die Werte haben. Diese Werte können neben Zahlen und Zeichenketten auch Listen und verschachtelte Unterdokumente sein. Listen können ebenfalls wieder Werte beliebigen Typs beinhalten und Unterdokumente wieder beliebige Felder. In *MongoDB* [Mona] und vielen anderen Dokumentendatenbanken wird als Format, in dem die Dokumente dem Benutzer dargestellt werden, *JSON* [JSOa] verwendet. Für die interne Speicherung wird auf effizienter zu durchsuchende Strukturen wie das binäre JSON-Format *BSON* zurückgegriffen. Das folgende Beispiel zeigt einen MongoDB `findOne`-Befehl auf der Kollektion `pers` und das zurückgelieferte JSON-Dokument:

```
db.pers.findOne( { _id: 1 } )
```

```
{
 _id: 1,
  vorname: "Rita",
  telefon: [
    { typ: "privat", nr: "06315555" },
    { typ: "mobil", nr: "017555555" }
  ]
}
```

Ein zweites Beispiel für ein Dokumentendatenbanksystem ist CouchDB. Zwar verwaltet auch CouchDB Dokumente im JSON-Format und unterstützt damit beliebige strukturierte Objekte mit Arrays und Unterdokumenten, allerdings gibt es im Vergleich zu MongoDB einige fundamentale Unterschiede. Der standardmäßige Zugriff auf Dokumente erfolgt über eine REST-API unter der Verwendung von den HTTP-Methoden `PUT`, `GET` und `DELETE`. Für komplexere Suchanfragen über Sekundärattribute wird der Einsatz von *Views* empfohlen. Diese Sichten haben einen eindeutigen Namen und werden über eine JavaScript-Funktion definiert, welche eine MapReduce-artige Transformation ausführt. Anders als Sichten in SQL sind CouchDB-Sichten *materialisiert*. Beim Einfügen, Ändern und Löschen von Dokumenten, werden also alle Sichten gemäß ihrer JavaScript-Implementierung aktualisiert. Transformationen wie Projektionen und Selektionen

bestehend lediglich aus einer Map-Funktion. Komplexere Transformationen, in denen Gruppen aus mehreren Dokumenten gebildet werden und in denen Aggregatfunktionen ausgeführt werden, bestehen aus den Funktionen Map und Reduce. Weitere Details über MapReduce befinden sich in Abschnitt 2.2. Ein weiterer Unterschied zwischen CouchDB und MongoDB ist, dass es in einer CouchDB-Datenbank keine Kollektionen gibt. Alle Dokumente werden also zusammen in eine Datenbank geschrieben, was in folgendem alternativen Datenmodell dargestellt werden soll:

$$Datenbank \rightarrow Dokument*$$

Zur Unterscheidung verschiedenartiger Dokumententypen wird in CouchDB oft das Attribut `doc_type` verwendet. Die erste Operation in der Definition einer Sicht ist somit meistens `if(doc.doc_type == "...")`.

### Graphdatenbanken

Bei den drei zuvor genannten Kategorien von NoSQL-Datenbanksystemen erfolgt der Zugriff auf die Datensätze entweder über eine gegebene ID oder mittels eines sequentiellen oder eines Index-Scans auf einer einzigen Datenquelle. Operationen wie Joins oder Selektionen auf Sekundärattributen werden entweder nicht unterstützt, erfordern gewisse Hilfsstrukturen wie Indexe, oder sie sind schlichtweg einfach nur sehr kostspielig. Graphdatenbanken sind für *Traversierungen*, also *navigierende Zugriffe*, optimiert. Die einzelnen Datensätze werden *Knoten* genannt und sind vergleichbar mit Dokumenten in Dokumentendatenbanken. Jeder Knoten kann eine beliebige Anzahl von Attributen haben (sogenannte *Properties*), in der die zu den Knoten gehörenden Daten gespeichert werden. Das Konzept einer ID wird nicht von allen Graphdatenbanken unterstützt. Manche setzen sie zwar intern zur Unterscheidung von Knoten ein, bieten aber nicht die Möglichkeit, eine ID manuell festzulegen. Viele Systeme bieten Unterstützung für sogenannte Knoten-*Labels*. Diese beschriften einen Knoten mit einem oder mehreren Typen, z. B. „Kunde“ oder „Produkt“. Alle Knoten werden zusammen in einem Graphen gespeichert, sodass zusammen mit den *Kanten*, die die Knoten miteinander verbinden, das Datenmodell als Graph im mathematischen Sinne gesehen werden kann:

$$G = (V, E)$$

Der *Property-Graph* $G$ besteht aus einer Menge von Knoten $V$ und Kanten $E$. Knoten besitzen beliebige Properties sowie optional Labels und eine ID. Auch Kanten können Properties besitzen. Bei ihnen ist das Label Pflicht, um die Art der Beziehung zwischen zwei Knoten anzugeben. Kanten sind in den meisten Graphdatenbanken stets gerichtet, sie besitzen also jeweils einen Start- und einen Ziel-Knoten.

Das folgende Beispiel zeigt, wie ein Graph bearbeitet und angefragt werden kann. Die hier verwendete Sprache ist *Gremlin* [Rod15], die sich in Verbindung mit gängigen Graphdatenbanksystemen wie zum Beispiel Neo4J [Neo] verwenden lässt:

```
k = g.addVertex(1, [label:"Person", vorname:"Kai"])
u = g.addVertex(2, [label:"Person", vorname:"Ute"])
```

```
g.addEdge(null, k, u, "Freund", [seit:2016])
freunde = g.V.has("vorname", "Kai").both("Freund")
```

Im Beispiel werden zwei Personen-Knoten erstellt, die mit einer Kante des Labels „Freund“ verbunden werden. Obwohl eine Freundschaftsbeziehung symmetrisch sein sollte, werden in Graphdatenbanken üblicherweise nur gerichtete Kanten verwendet. Die Richtung ist in unserem Beispielgraphen zwar gegeben, aber irrelevant. Bei der Abfrage in der untersten Zeile wird zunächst die Person mit dem Vornamen Kai gesucht und von diesem Knoten aus über alle Kanten mit dem Label „Freund“ traversiert. Das Ergebnis des `both`-Schrittes liefert alle Knoten, die über solche Kanten, egal ob ein- oder ausgehend, erreichbar sind, also in diesem Fall alle Freunde von Kai.

## 2.2 Datenverarbeitungsframeworks

Wie wir im vorherigen Abschnitt gesehen haben, bieten typische NoSQL-Systeme sehr einfache Abfragemöglichkeiten. Anders als bei relationalen Datenbanksystemen kommt auf Anwendungsseite üblicherweise keine Anfragesprache wie SQL zum Einsatz, stattdessen werden direkte API-Methoden wie `PUT` und `GET` verwendet. Dies erspart zwar den Aufwand, den das Datensystem zum Parsen, zur Interpretation und Optimierung einer Anfrage benötigt, es schränkt aber die Mächtigkeit der möglichen Anfragen stark ein. Der Hauptanwendungszweck der betrachteten Systeme ist es, einen Datensatz zurückzugeben, der den gegebene ID-Wert besitzt. Üblicherweise kann diese typische `GET`-Operation um eine Projektion erweitert werden, um nicht benötigte Attribute auszublenden. Viele Systeme bieten auch Bereichs- oder Mustersuche anhand der ID an. So kann man in HBase alle Zeilen lesen, deren Row-ID in einem bestimmten Bereich liegen und von Redis eine Liste von Schlüsseln ausliefern lassen, die zu einem gegebenen regulären Ausdruck passen. Prädikate auf anderen Attributen als der ID sind in Wide-Column-Stores typischerweise nur mit kostspieligen Filter-Operatoren möglich, die einen vollständigen Tabellenscan erfordern. In Key-Value-Stores werden Wert-basierte Filter meist gar nicht unterstützt. Dokumenten- und Graphdatenbanken bieten hier einen Sonderfall, da diese ähnliche Selektionsmethoden und Zugriffspfade bieten, wie man sie aus relationalen Datenbanken kennt. Durch das flexible Schema kann jedoch die Speicherung nicht so effizient wie in letzteren erfolgen, was schlussendlich auch wieder die Zugriffsgeschwindigkeit reduziert.

Geht man von einfachen Zugriffen und Suchoperationen zu Datenanalysemethoden, bieten die meisten NoSQL-Datenbankmanagementsysteme dafür keine Unterstützung. Eine Ausnahme ist die MongoDB Aggregation Pipeline, mit der eine Folge von Operationen definiert werden kann, um Daten zu filtern, gruppieren, aggregieren und zu sortieren. Über weitere Methoden sind auch das Auseinandernehmen von Listen sowie Verbundoperationen möglich. In Kapitel 3.3 befinden sich weitere Details über die Funktionsweise und die Einschränkungen der MongoDB Aggregation Pipeline. Ein anderer Weg, komplexe Datenanalysen durchzuführen, bieten Plattformen, die es erlauben, SQL-Anfragen an NoSQL-Datenbanken zu stellen. Damit kann sich der Anwender aus den aus SQL bekannten Möglichkeiten zur Gruppierung, Aggregation und mehr bedienen. Auch auf diese wird später, in Kapitel 3.1 detailliert eingegangen.

Wenn große Datenmengen als Gesamtes analysiert oder transformiert werden sollen, ist es geschickt, ein Datenverarbeitungsframework einzusetzen. Der Zugriff erfolgt dann anders als bei der Verwendung der Datenbank-API-Methoden nicht auf einzelne Datensätze, sondern auf den kompletten Datenbestand. Bei Wide-Column-Stores wird also üblicherweise eine komplette Tabelle analysiert, in Dokumentendatenbanken eine Kollektion. Auch Textdateien können auf diese Art vollständig analysiert werden. Die im Folgenden vorgestellten Frameworks nutzen die Verteiltheit der Daten aus und führen so viele Operationen wie möglich direkt auf denjenigen Rechnern aus, auf denen die Daten gespeichert sind. Zur finalen Berechnung der Ergebnisse ist in verteilten Umgebungen meist ein Datentransport über das Netzwerk vonnöten, der jedoch durch die lokale Vorverarbeitung und durch geeignete Partitionierungstechniken optimiert werden kann. Die Ausgabe einer Berechnung wird üblicherweise in eine Datenbank oder eine Datei geschrieben. Die direkte Verwendung des Ergebnisses innerhalb einer Anwendung ist in vielen Frameworks aber auch möglich.

Bei Frameworks handelt es sich um *Programmiergerüste*, also um Schnittstellenbeschreibungen sowie fertige Klassen und Methoden, die einem Anwendungsprogrammierer gewisse Aufgaben abnehmen. Der Anwender entwickelt weiterhin in einer Programmiersprache wie Java oder Python und hat die komplette Mächtigkeit, die diese Sprachen mit sich bringen. Er muss sich dabei allerdings nicht um die Entwicklung generischer Aufgaben kümmern, sondern kann sich auf die für den speziellen Datenanalysezweck spezifischen Aspekte konzentrieren. Die Frameworks bringen folgende, nicht zwangsweise alle, Funktionalitäten mit:

- Ein Programmiermodell, an das sich der Anwender bei der Entwicklung hält,
- die verteilte Ausführung benutzerdefinierter Methoden,
- Schnittstellen zu (verteilten) Datenbanken und Dateisystemen für die Ein- und Ausgabe,
- Verteilung (Partitionierung) von Zwischen- und Endergebnissen auf die Rechnerknoten im Netzwerk,
- Bereitstellung typischer Funktionen (Gruppieren, Aggregieren, Sortieren, Zählen, ...),
- Wiederanlauf im Fehlerfall sowie das Wiederverwenden oder Verwerfen unvollständiger (Zwischen-)ergebnisse,
- Monitoring-Tools, um die Ausführung und Historie zu betrachten,
- Sicherheitsmechanismen wie Authentifizierung und Datenverschlüsselung.

Im Folgenden werden einige prominente Datenverarbeitungsframeworks kurz vorgestellt.

**MapReduce / Apache Hadoop** Das von Google vorgestellte Programmiermodell und Framework *MapReduce* [DG04] ist aufgrund seiner Einfachheit und dennoch sehr großen Mächtigkeit sehr populär. Der berühmteste Klon von Google's MapReduce ist das in Java geschriebene quelloffene Framework *Apache Hadoop* [Apaa]. Die Grundidee hinter MapReduce ist es, eine Menge von Eingabedaten Datensatz für Datensatz mit einer benutzerdefinierten *Map*-Funktion in Schlüssel-Wert-Paare zu transformieren. Da die einzelnen Berechnungen voneinander unabhängig sind, können sie verteilt durchgeführt werden. Die Zwischenergebnisse werden im Anschluss anhand ihres Schlüssels gruppiert und partitioniert, sodass sie zur weiteren Verarbeitung wieder auf den vorhandenen Rechnerknoten verteilt vorliegen. Eine benutzerdefinierte *Reduce*-Funktion nimmt nacheinander einen dieser Schlüssel sowie die Liste der zu diesem Schlüssel gehörenden Werte entgegen und berechnet anhand dieser das Endergebnis. Die Ausführung der Reduce-Funktion erfolgt ebenfalls wieder verteilt.

$$map : Datensatz \rightarrow (Schlüssel \times Wert)*$$
$$reduce : Schlüssel \times Wert* \rightarrow Schlüssel \times Wert$$

Die hier gezeigte Definition der Map- und Reduce-Funktion soll lediglich dem groben Verständnis dienen. Verschiedene MapReduce-Implementierungen weichen teilweise stark von diesem Modell ab. Manche setzen voraus, dass bereits die Eingabedaten als Schlüssel-Wert-Paare vorliegen. Auch ist es möglich, dass die Reduce-Funktion mehrere Ausgaben liefern kann.

In *Hadoop* erfolgt die Ausführung eines MapReduce-*Jobs* als eine Menge von *Tasks*. Auf jedem Rechner wenden ein oder mehrere Map-Tasks die Map-Funktion auf einer Menge von Eingabedatensätzen an. Beim Auftreten eines Fehlers oder beim Ausfall eines Rechners wird der betreffende Task neu gestartet. Eventuell vorhandene Zwischenergebnisse werden verworfen, um ein Exacly-Once-Verhalten und damit die Korrektheit des Endergebnisses sowie eine deterministische Ausführung zu gewährleisten. Nach Abschluss der Map-Phase erfolgt äquivalent dazu die Ausführung der Reduce-Tasks.

Vorteile von MapReduce sind neben dem einfachen Programmiermodell, der Mächtigkeit und der Fehlertoleranz auch die gute Skalierbarkeit. MapReduce wird häufig für Text- und Log-Analysen eingesetzt sowie zur Ausführung von Graph-Algorithmen. Nachteile sind die Einschränkungen, die das starre Programmiermodell mitbringt. Beispielsweise ist für einige Algorithmen, die nicht als ein einziger MapReduce-Job ausgedrückt werden können, die Definition von MapReduce-*Ketten* nötig. Ein erster Job erledigt einen ersten Schritt und schreibt die Ergebnisse auf die Festplatte, die anschließend als Eingabe für einen zweiten Job dienen. Dieses Schreiben und Lesen kann zu einer sehr inperformanten Ausführung führen. Das gleiche Problem tritt bei iterativen Berechnungen auf. Dort wird das Ergebnis jeder Iteration auf der Festplatte gespeichert, bevor es in der nächsten Iteration gelesen werden kann.

Manche Datenbanksysteme - wie zum Beispiel MongoDB - bieten eine eingebaute MapReduce-Funktionalität, um benutzerdefinierte Berechnungen verteilt auszuführen. Der allgemeingültige und elegantere Weg ist es jedoch, Apache Hadoop in Verbindung mit den für ein Datenbanksystem

spezifischen Ein- und Ausgabeformaten zu verwenden, wie zum Beispiel dem „TableInputFormat“ für HBase oder dem „MongoInputFormat“, welches ein Teil des MongoDB Hadoop Connectors ist.

**Apache Spark** Die Idee hinter Spark [Zah+10] ist es - genau wie beim MapReduce-Framework - eine Plattform zu bieten, um benutzerdefinierte Methoden auf großen Datenmengen verteilt auszuführen. Allerdings verzichtet Spark auf ein starres Programmiermodell und bietet stattdessen die Möglichkeit, eine Reihe von Operationen nach Belieben zu definieren. Dazu gehören neben den Funktionen Map und Reduce auch Filter, Joins und Gruppierungen. Die Datenzugriffe und auch die Operationen liefern sogenannte *Resilient Distributed Datasets* (*RDD*s). Für MongoDB bietet beispielsweise der MongoDB Hadoop Connector den Zugriff auf eine Kollektion als RDD. Werden nun auf diesem RDD Operationen wie Filter oder Map ausgeführt, sorgt das Spark Framework dafür, dass diese Berechnungen so weit es geht direkt auf den Rechnern erfolgen, auf denen die Daten gespeichert sind. Genau wie bei MapReduce wird so der Datentransport minimiert. Aufgrund des flexiblen Programmiermodells werden die in MapReduce erwähnten Schreibvorgänge für Zwischenergebnisse vermieden. Stattdessen führt Spark weitestgehend alle Berechnungen im Arbeitsspeicher aus und schreibt erst das Endergebnis in die gewünschte Ausgabedatei oder in eine Datenbank. Auch bei iterativen Berechnungen behält Spark die Zwischenergebnisse im RAM. Genau wie Hadoop ist Spark fehlertolerant und garantiert eine Exactly-Once-Ausführung für alle Berechnungen.

In Spark gibt es zwei Arten von Operationen, nämlich *Transformationen* und *Aktionen*. Transformationen unterscheiden sich wiederum in solche mit *nahen Abhängigkeiten* und *weiten Abhängigkeiten*. Erstere können direkt hintereinandergeschaltet werden, z. B. eine einfache Datensatztransformation mittels einer Map-Funktion und ein Filter. Weite Abhänigkeiten erfordern eine Berechnung von Zwischenergebnissen. Eine Reduce-Funktion benötigt beispielsweise die vollständige Liste an Werten zu einem Schlüssel, bevor die Berechnung starten kann. Beim Start eines Spark-Jobs werden die Operationenfolgen analysiert und optimiert. Dabei erzeugt Spark so viele Ausführungsphasen (Stages), wie es weite Abhängigkeiten gibt, plus eine weitere. Ein typischer MapReduce-Job würde also auch bei Spark aus zwei Phasen bestehen. Unter Aktionen versteht man Operationen welche tatsächlich Berechnungen ausführen. Denn allein durch die Ausführung einer Transformation werden keine Daten gelesen oder verarbeitet. Dies passiert erst *lazy*, sobald eine Aktion erfolgt. Typische Aktionen sind `collect` oder `cache`. Erstere wandelt ein RDD in eine Java-, Scala oder Python-Kollektion, sie hebt also die Verteilung auf und materialisiert die Ergebnisse. Die `cache`-Operation liefert wie gehabt ein RDD, erzwingt allerdings das Ausführen vorangegangener Transformationen. Dies ermöglicht ein mehrfaches Wiederverwenden von Berechnungszwischenergebnissen.

*Spark Streaming* [Zah+12] erweitert Spark um die Möglichkeit, Datenströme in Echtzeit zu analysieren. Im Gegensatz zur klassischen Stapelverarbeitung von gespeicherten Datensätzen sind Datenströme kontinuierlich, sodass die Berechnung theoretisch nie abgeschlossen ist. Stattdessen liegen die Ergebnisse von Datenstromanalysen auch wieder als Datenstrom vor.

Analog zu RDDs werden bei Spark Streaming die gleichen Arten von Operationen (Map, Reduce, Join, ...) auf diskretisierten Strömen (*DStream*s) ausgeführt. Der eingehende Datenstrom wird zunächst in Pakete („Batches“) mit einer Größe von wenigen Sekunden zerlegt, auf denen wie auf RDDs gearbeitet werden kann. Die Ergebnis-Pakete werden als Datenstrom ausgegeben oder in eine Datenbank oder Datei gespeichert. Typische Datenstromanfragen sind Fenster-basiert. Das heißt, auf einer Sequenz von RDDs wird ein Fenster definiert, welches eine bestimmte Größe hat und in bestimmten Intervallen aktualisiert wird. Wird Spark Streaming beispielsweise zur Analyse von Webserver-Logdateien eingesetzt, könnte eine Anfrage alle fünf Minuten die Anzahl der Besuche pro Webseite innerhalb der letzten halben Stunde ausgeben. Hier wäre die Fenstergröße dreißig Minuten und das Aktualisierungsintervall fünf Minuten.

**Apache Flink** Es existieren weitere Datenverarbeitungsframeworks, die eine ähnliche Funktionalität wie die oben vorgestellten Programme bieten. Apache Flink [Apa15b] ist speziell für die Analyse von Datenströmen konzipiert, bietet aber auch Möglichkeiten zur Stapelverarbeitung. Die Verwendung ist ähnlich zu der von Apache Spark, allerdings werden Ströme nicht als Sequenzen von RDDs mit einer gewissen zeitlichen Größe angesehen, sondern als tatsächliche kontinuierliche Datenstrom-Objekte. Dadurch eignet sich Flink besonders für Echtzeit-Analysen.

## 2.3 Inkrementelle Wartbarkeit

Die zu Beginn dieses Kapitels vorgestellten API-Methoden wie `PUT` und `GET` bieten *OLTP*-Anwendungen einen Datenzugriff innerhalb weniger Millisekunden. OLTP steht für *Online-Transaction-Processing*, also für die Echtzeitverarbeitung von Transaktionen. Da Zugriffe in NoSQL-Datenbanken meist ID-basiert sind und Transaktionen in dem Sinne, dass sie eine atomare Folge von Datenbankaktionen darstellen, von den meisten Systemen nicht unterstützt werden, sind Datenbankzugriffe in OLTP-Anwendungen sehr schnell und es können sehr viele davon parallel stattfinden.

Im zweiten Abschnitt dieses Kapitels wurden mehrere Datenverarbeitungsframeworks vorgestellt, die für die Zwecke des *Online-Analytical-Processings* (*OLAP*) eingesetzt werden. OLAP-Anwendungen dienen zur Datenanalyse und bestehen typischerweise aus Prozessen mit langer Laufzeit, die ihre Ergebnisse in eine Zieldatenbank oder in eine Datei schreiben. Da die Ergebnisse allerdings nur auf dem Stand der Daten zum Zeitpunkt der Analyse sind und da sich die Basisdaten regelmäßig ändern, müssen die Berechnungen typischerweise oft wiederholt werden. In diesem Abschnitt geht es darum, OLAP-Anwendungen inkrementell wartbar zu machen. Unter inkrementeller Wartbarkeit versteht man die Fähigkeit, dass eine Anwendung zur erneuten Berechnung eines Ergebnisses das vorherige Ergebnis als Basis nehmen kann und die Änderungen an den Basisdaten darauf anwendet.

Soll eine Berechnung inkrementell wartbar sein, müssen die folgenden Fragen geklärt werden:

1. Was hat sich in den Basisdaten geändert?

2. Welche Werte im vorherigen Ergebnis müssen geändert werden?

3. Wie wirken sich die Änderungen auf das vorherige Ergebnis aus?

Die Komponente zur Erfassung der Änderung (Punkt 1) ist die sogenannte Änderungserfassung (*Change-Data-Capture, CDC*) [KR11]. Zuerst stellen wir verschiedene Change-Data-Capture-Verfahren vor, die es ermöglichen, die Änderungen in den Basisdaten seit der vorherigen Berechnung herauszufinden. Anschließend beschreiben wir Ideen zur Einbringung der Änderungen auf das vorherige Ergebnis (Punkte 2 und 3).

### Verfahren zur Änderungserfassung

Verschiedene Änderungserfassungsverfahren unterscheiden sich zum einen bezüglich ihrer Einsetzbarkeit für die betrachteten Datenbanksysteme. Des Weiteren müssen je nach Verfahren unterschiedliche Mengen an Daten analysiert werden, was zu teilweise sehr deutlichen Geschwindigkeitsunterschieden führen kann. Aber auch hier gibt es keinen klaren Sieger. Manche Verfahren eignen sich gut, wenn die Änderungen seit der letzten Durchführung gering sind, andere, wenn sich viel geändert hat. Wenn wir von Änderungen sprechen, teilen wir diese in zwei Arten auf, und zwar *Einfügungen* und *Löschungen*. Sei $B$ die Datenbasis, auf der die Berechnung erfolgt. $B(t_1)$ bezeichnet den aktuellen Zustand und $B(t_0)$ den Zustand der Basisdaten zum Zeitpunkt $t_0$ der vorherigen Berechnung.

$$\begin{aligned} \text{Einfügungen } \Delta &= B(t_1) - B(t_0) \\ \text{Löschungen } \nabla &= B(t_0) - B(t_1) \end{aligned}$$

Einfügungen sind also diejenigen Datensätze, die seit der vorherigen Berechnung neu hinzugekommen sind, Löschungen diejenigen, die seitdem entfernt wurden. Findet innerhalb eines existierenden Datensatzes eine Veränderung statt, kann diese als Löschung mit anschließender Einfügung gesehen werden. Bei der nun folgenden Analyse der verschiedenen Änderungserfassungsverfahren wird auffallen, dass Löschungen nicht von allen Verfahren unterstützt werden.

**Audit-Columns** Der Begriff stammt aus Data-Warehouse-Systemen, bei denen in den zu überwachenden Tabellen speziell für das CDC vorgesehene Spalten hinzugefügt werden. Im allgemeinen Fall versteht man unter einer Audit-Column ein Attribut pro Datensatz, anhand dessen sich erkennen lässt, ob sich ein Datensatz geändert hat. In der Regel beinhaltet das Attribut den Zeitstempel der letzten Änderung oder eine Versionsnummer. Es ist auch denkbar, dass die Audit-Column lediglich die boolesche Information beinhaltet, ob sich der Datensatz seit der vorherigen Berechnung ändert hat oder nicht. In letzterem Falle muss das Transformationssystem nach einer Berechnung diesen Wert zurücksetzen, während Zeitstempel und Versionsnummern in den Basisdaten erhalten bleiben können.

Das relationale Datenbankmanagementsystem IBM DB2 für z/OS bietet eine native Unterstützung für Audit-Columns. Beim Anlegen der Tabelle wird die betreffende Spalte mit der Option `ON UPDATE AS ROW CHANGE`

`TIMESTAMP` versehen, sodass das Datenbanksystem automatisch den Zeitstempelwert aktuell hält. In MongoDB werden beim Einsatz automatisch erzeugter Objekt-IDs die ersten vier Bytes dieser ID für den Zeitstempel der Einfügung verwendet. Nachträgliche Änderungen am Dokument verändern allerdings nicht die Objekt-ID und damit auch nicht den Zeitstempel. Da auf dem ID-Attribut stets ein B+-Baum-Index existiert, sind mittels Präfixanfragen effiziente Suchen nach neu eingefügten Dokumenten möglich. Im Falle von DB2 lassen sich mit einer Selektion auf der Audit-Column alle Zeilen finden, die sich seit der letzten Berechnung geändert haben. Eine Unterscheidung, ob es sich um einen komplett neuen oder um einen veränderten Datensatz handelt, ist in beiden genannten Ansätzen nicht möglich. In MongoDB werden auf diese Art nur neu eingefügte Dokumente gefunden. In jedem Fall ist das Erkennen von Löschungen ausgeschlossen. Um dieses Problem zu lösen, müssen Löschungen verboten werden und stattdessen eine Audit-Column mit den Zeitpunkt der logischen Löschung eingeführt werden.

Mit Audit-Columns lässt sich zwar feststellen, dass sich etwas am Datensatz geändert hat, aber nicht was. Das Analyseprogramm hat nur Zugriff auf den neuen Stand der Daten, nicht aber auf den Stand zum Zeitpunkt der vorherigen Berechnung. Das liegt daran, dass beim Einsatz von Audit-Columns keine automatische Historie vorheriger Attributwerte angelegt wird.

Das folgende Beispiel zeigt mehrere Varianten, wie Audit-Columns eingesetzt werden können, um Änderungen an Datensätzen zu erfassen:

```
{ _id: 1, name: "Ute", inserted: 1451606400 }
{ _id: 2, name: "Jörg", inserted: 1451606400,
                        updated: 1451607333 }
{ _id: 3, name: "Lisa", modified: true }
{ _id: 4, deleted: 1451607444 }
{ _id: ObjectId("5650d1906b66d66a5dbfda29"), ...}
```

**Zeitstempel-basiertes CDC mittels Multi-Versionierung** Verwaltet ein Datenbanksystem eine Historie von Attributwerten und bietet darauf Zugriffsmöglichkeiten, kann diese zur Erfassung der Änderungen verwendet werden. Mittels Multi-Versionierung lässt sich leicht herausfinden, welche Datensätze seit der vorherigen Berechnung neu eingefügt wurden, welche sich geändert haben und an welchen Datensätzen sich nichts geändert hat. Des Weiteren hat man Zugriff auf die vorherigen Werte. Damit lässt sich beispielsweise berechnen, um wie viel sich ein nummerischer Wert erhöht oder verringert hat. Lediglich Löschungen können mittels Multi-Versionierung nur schwierig erkannt werden. Sollen diese ebenfalls unterstützt werden, empfiehlt es sich, statt dem endgültigen Löschen von Datensätzen (samt ihrer vollständigen Historie) lediglich deren Attribute zu entfernen oder die Attributwerte auf `null` zu setzen. Einige NoSQL-Datenbanksysteme wie HBase bieten eine eingebaute Multi-Versionierung, die allerdings meist erst explizit aktiviert werden muss. Dazu gibt man die Anzahl $n$ der zu verwaltenden Versionen an. Im Extremfall kann dies bei einer Datenbank der Größe $N$ zu einem Speicheraufwand von $O(N \cdot n)$ führen. Bei HBase wird zu jedem Attribut eine Historie von Datenwerten

zusammen mit den Zeitstempeln des Schreibens gespeichert. Im Allgemeinen könnte man aber auch eine laufende Versionsnummer verwenden. Für eine zuverlässige Änderungserfassung ist allerdings die Variante mit Zeitstempeln klar im Vorteil, da basierend auf diesen leicht festgestellt werden kann, ob eine Version vor oder nach der Durchführung der vorherigen Berechnung erstellt wurde.

Andere Ansätze verfolgen die Idee, dass sich nicht das Datenbankmanagementsystem sondern die Anwendung um die Versionierung kümmert. Da der Zugriff bei NoSQL-Datenbanken meist über die ID erfolgt und viele Systeme zur Indizierung der IDs B+-Bäume mit einer Unterstützung für Präfix-Suchen verwenden, ist ein gängiger Ansatz, eine Versionsnummer oder einen Zeitstempel als Suffix an die ID anzuhängen [Gra13]. Statt der ID `p17` könnte diese dann etwa `p17:1451606400` lauten. Über eine Suche nach IDs mit dem Präfix `p17:` werden alle Versionen gefunden. Diese manuelle Art Versionierung kann bei vielerlei NoSQL-Datenbanksystemen eingesetzt werden. Es bietet sich an, eine Client-Erweiterung zu verwenden, um Datenbankmethoden (`PUT`, `GET`, ...) wie gehabt verwenden zu können, um standardmäßig auf die aktuelle Version zuzugreifen und um beim Einfügen und Ändern die Zeitstempel automatisch zu setzen [Fel+14; CL13].

**Log-basiertes CDC** Eine Log-Datei ist eine vom Datenbankmanagementsystem gepflegte Datei, in welcher von Anwendungen durchgeführte Datenbankänderungen protokolliert werden [HR83]. Sie kommt für verschiedene Zwecke zum Einsatz, in der Regel zur Gewährung der ACID-Eigenschaften. Mittels *Transaktionslogs* kann die Atomarität von Transaktionen gewährleistet werden. Geht von einer Reihe von atomar auszuführenden Operationen eine Operation schief oder beschließt die Anwendung den manuellen Abbruch einer Transaktion, kann die Log-Datei genutzt werden, um bereits ausgeführte Änderungen rückgängig zu machen. Dazu ist es vonnöten, dass die Log-Datei die Bytes der geänderten Blocke vor der Änderung enthält. Alternativ kann die Atomarität auch dadurch gewährleistet werden, dass noch nicht freigegebene Änderungen lediglich in die Log-Datei und noch nicht in die Datenbank geschrieben werden. Beim Abbruch der Transaktion müssen dann nur die Einträge in der Log entfernt werden. Erst beim Commit der Transaktion werden die Änderungen in die Datenbank eingebracht. Ein zusätzlicher Vorteil dieser Variante ist die Gewährleistung der Isolation, da nicht freigegebene Änderungen noch nicht von anderen Transaktionen gelesen werden. Diese Variante Transaktionen zu unterstützen wird beispielsweise vom Graphdatenbanksystem Neo4J auf eine ähnlich zu der hier beschriebenen Art eingesetzt. Der Key-Value-Store Redis hingegen nutzt Log-Dateien, um die Dauerhaftigkeit abgeschlossener Transaktionen zu gewährleisten. Dies ist vonnöten, da Redis eine In-Memory-Datenbank ist. Änderungen erfolgen also nur im Hauptspeicher und werden je nach Konfiguration lediglich im Zeitintervall einiger Sekunden auf die Festplatte übertragen. Um zwischenzeitige Schreiboperationen bei einem Stromausfall nicht zu verlieren, kommt die Log-Datei zum Einsatz, welche persistent gespeichert ist. Aufgrund des sequenziellen und asynchronen Schreibens leidet die Performanz des Systems nicht so stark darunter wie beim tatsächlichen persistenten Einbringen jeder Änderung

beim Abschluss einer Transaktion. Auch in MongoDB wird ein sogenanntes Journal eingesetzt, um Anwendungen die Bestätigung der erfolgreichen Ausführung einer Operation zu geben, noch bevor diese persistent geschrieben, wohl aber geloggt, wurden.

Ein anderer Anwendungsfall für Log-Dateien kommt bei der *Datenreplikation* zum Einsatz. Der Grundgedanke von Replikation ist, dass jede Änderung, die auf einem Rechner (Replika) ausgeführt wird, ebenfalls auch auf die anderen Replikas übertragen wird, um dort den kompletten Datenbestand quasi in Echtzeit zu spiegeln. Aufgrund von Netzwerklatenzen, temporären Ausfällen und anderen Verarbeitungsverzögerungen ist das gespiegelte Abbild jedoch nur „beinahe live“. Ist ein Replika aufgrund schwererer Probleme oder wegen eines Ausfalls für eine längere Zeit nicht erreichbar, müssen beim Wiederanlauf alle verpassten Operationen nachgeholt werden. Hierbei kommt die Log-Datei zum Einsatz. Diese enthält entweder Logsequenznummern oder Zeitstempel, mit denen sich feststellen lässt, welche Änderungen noch auszuführen sind.

Log-Dateien können im Wesentlichen auf zwei Arten aufgebaut sein. Die erste Möglichkeit ist *logisches Logging* oder auch *Statement-basiertes Logging*. Dabei beinhaltet jeder Log-Eintrag eine konkrete Datenbankoperation, beispielsweise ein Einfügen, Löschen oder Ändern. Leseoperationen werden nicht geloggt, da sie keine Auswirkungen auf den Datenbestand haben. Die alternative Methode ist *physisches Logging* oder auch *binäres Logging*. Hierbei wird protokolliert, welche Bytes in welchen Blöcken inwiefern geändert werden müssen. Zwar gibt es noch weitere Log-Typen - wie das physiologische Logging -, auf die aber hier nicht weiter eingegangen wird. Stattdessen betrachten wir die Vorteile und Nachteile sowie die Möglichkeiten, wie Log-Dateien zu Zwecken der Änderungserfassung verwendet werden können. Statement-basiertes Logging kommt beispielsweise bei der Replikation in MongoDB zum Einsatz, da diese auch dann funktioniert, wenn auf unterschiedlichen Replikas verschiedene MongoDB-Versionen oder unterschiedliche Storage-Engines verwendet werden. Der Nachteil ist, dass aus den Log-Einträgen keine Information entnommen werden kann, wie eine Operationen wieder rückgängig gemacht werden kann. Die Log-Datei kann also nur vorwärts und nicht rückwärts abgespielt werden. Bei der Replikation ist ohnehin nur ersteres vonnöten.

Für die Änderungserfassung bringt Statement-basiertes Logging große Nachteile mit sich. Mit der Log-Datei lässt sich zwar feststellen, welche Änderungen an der Datenbank seit einer vorherigen Analyse ausgeführt wurden, allerdings ist der Basisdaten-Zustand zum Zeitpunkt der vorherigen Analyse nicht mehr abrufbar. Einfach gesagt werden also nur Einfügungen und keine Änderungen oder Löschungen unterstützt. Erst durch aufwändige Methoden, wie dem Durchsuchen der Log nach vorherigen Operationen auf einem gegebenen Datensatz oder über eine Kombination mit Snapshot-basiertem Logging (siehe nächster Absatz) lässt sich auch Log-basiertes CDC zuverlässig verwenden.

Das folgende Beispiel zeigt einen Eintrag der in MongoDB zur Replikation geführten Operationen-Log. Er beinhaltet unter anderem einen Zeitstempel (`ts`), die Art der Operation (`i` = Einfügung), die betroffene Datenbank und Kollektion (`ns` = Namespace) sowie das neue Objekt (`o`). Bei Änderungen und Löschungen befinden sich die IDs der zu ändernden bzw. zu löschenden Dokumente in den Log-Einträgen.

```
{ ts : Timestamp(1448269379, 1), ..., op: "i",
   ns: "test.personen", o: { _id: 1, name: "Ute" } }
```

**Snapshot-basiertes CDC** Der Grundgedanke dieser Art der Änderungserfassung ist sehr einfach: Zum Zeitpunkt jeder Analyse wird ein vollständiges Abbild der Basisdaten, ein sogenannter Snapshot oder Schnappschuss, gespeichert. Bei einer späteren Analyse kann der aktuelle Datenbankzustand mit dem vorherigen verglichen werden [Lin+86; LGM96]. Aufwändig daran ist nicht nur das Anlegen dieser Abbilder sondern auch der Vergleich zweier Snapshots. Allerdings bringt Snapshot-basiertes CDC auch Vorteile mit sich. Ändern sich einzelne Datensätze auf einer Datenbasis enorm oft, oder wird eben so viel gelöscht wie eingefügt, würden bei Multi-Versionierung oder Log-Dateien große Datenmengen anfallen, da sie alle Zwischenzustände beinhalten. Bei Snapshot-basiertem CDC bleiben diese Zwischenzustände versteckt und es geht lediglich um den Vorher-Nachher-Vergleich. Während bei den zuvor vorgestellten Alternativen entweder das Datenbankmanagementsystem spezielle Funktionalität mitbringen muss oder Anwendungen angepasst werden müssen, ist diese Variante des CDC prinzipiell immer möglich. Auch unterstützt diese Variante alle Arten von Änderungen: Einfügungen, Löschungen sowie die Information, welche Attributwerte sich wie geändert haben. Herausforderungen liegen jedoch bei der Erstellung eines konsistenten Snapshots. Da der Erstellungsvorgang bei großen Datenmengen mehrere Minuten in Anspruch nehmen kann und in dieser Zeit weiterhin Änderungen durchgeführt werden, dürfen nicht Teile des Snapshots alte und andere Teile neue Zustände haben. Eine Möglichkeit zur Erreichung der Konsistenz ist beispielsweise die Verwendung von Multi-Versionierung.

**Trigger-basiertes CDC** Ein Datenbanktrigger ist eine aktive Komponente, welche benutzerdefinierte Operationen ausführt, sobald Änderungskommandos an die Datenbank gestellt werden. Unterstützt ein System Trigger, können diese zur Änderungserfassung eingesetzt werden. Eine Möglichkeit ist, dass ein Trigger bei jeder Einfügung, Änderung und Löschung „feuert“ und die Änderungen sowie den vorherigen Datenzustand in eine Warteschlange speichert. Diese kann zum Zeitpunkt von Berechnungen verwendet werden, um die Auswirkungen seit der vorherigen Berechnung zu bestimmen. Dieses Verfahren ist analog zum Log-basierten CDC, es ermöglicht allerdings ein individuell anpassbares Loggen von Operationen und Werte-Historien. Wenn das Datenbankmanagementsystem keine Log-Dateien führt oder wenn Log-Dateien keine Redo-Informationen beinhalten, kann dies durch den Einsatz von manuellem Logging mittels Triggern kompensiert werden.

Ein alternativer Ansatz von Triggern ist die unmittelbare Ausführung von Berechnungen. Während bei den vorherigen CDC-Ansätzen sowie bei einer vollständigen Neuberechnung Analysen üblicherweise in gewissen Zeitabständen (z. B. einmal täglich) ausgeführt werden und CDC dazu eingesetzt wird, die Änderungen seit der vorherigen Berechnung zu ermitteln, ist es mit Triggern möglich, Änderungen auf das Ergebnis anzuwenden, sobald sie auf den Basisdaten erfolgen. Abgesehen von der Berechnungsdauer und anderen Verzögerungen, ist das Ergebnis bei dieser Variante immer

„live“ und auf dem neusten Stand. In SQL ist ein äquivalentes Datenbankobjekt eine materialisierte Sicht mit der Option `REFRESH IMMEDIATE`. Sobald sich die Daten der in der `FROM`-Klausel stehenden Tabellen ändern, werden diese Änderungen direkt auf das materialisierte Ergebnis der Sicht angewandt.

### Inkrementelle Aggregationen

Datentransformationen, die lediglich Selektionen oder Projektionen durchführen sind leicht inkrementell ausführbar. Dazu wird bei neu eingefügten Datensätzen geschaut, ob sie das Selektionsprädikat erfüllen und dementsprechend werden sie zur Transformation betrachtet oder nicht. Bei Löschungen erfolgt auf gleiche Weise das Entfernen der Sätze. Lediglich bei Änderungen muss darauf geschaut werden, ob Prädikate vorher erfüllt waren und immer noch erfüllt sind. Sieht man Änderungen jedoch als Löschungen mit anschließendem Neueinfügen, ist auch hier die Lösung einfach.

Komplexe Berechnungen sind jedoch diejenigen, in denen Daten gruppiert und aggregiert werden. Bei der Berechnung einer Summe wird das bei der vorherigen Analyse berechnete Ergebnis als Basis genommen und die betreffenden Werte der eingefügten Datensätze darauf aufaddiert. Gelöschte Datensätze sorgen für Subtraktionen und Änderungen können auch hier wieder als eine Kombination aus beidem angesehen werden.

Betrachten wir das folgende Beispiel, in dem die Gehaltsumme pro Firma berechnet werden soll. Zu Beginn sehe der Datenbestand wie folgt aus:

```
{ _id: 1, name: "Ute", gehalt: 38000, firma: "IBM" }
{ _id: 2, name: "Jörg", gehalt: 37000, firma: "SAP" }
{ _id: 3, name: "Lisa", gehalt: 42000, firma: "IBM" }
```

Das Ergebnis ist also dieses:

```
{ _id: 'IBM', summe: 80000 }
{ _id: 'SAP', summe: 37000 }
```

Betrachten wir nun folgende Einfügung, Löschung und Änderung:

```
db.personen.insert({ _id: 4, name: 'Rita',
                     gehalt: 50000, firma: 'IBM' })
db.personen.remove({ _id: 1 })
db.personen.update({ _id: 3} {$set: { firma: 'SAP' }}
```

Die Gehälter der einzelnen Firmen ändern sich also folgendermaßen:

```
{ _id: 'IBM', summe: 80000+50000-38000-42000=50000}
{ _id: 'SAP', summe: 37000+42000=79000 }
```

Es fällt auf, dass die Zahl 42000, die zur Berechnung der neuen Summen benötigt wird, nicht in der Liste der Änderungsoperationen zu finden ist. Es ist also auch nötig, unveränderte Attribute zu betrachten.

Dieses Unterkapitel soll lediglich einen kleinen Einblick in die Herausforderungen bei inkrementellen Berechnungen liefern. Im späteren Verlauf der Arbeit wird auf diese genauer eingegangen. Bei der Verwendung von anderen Aggregatfunktionenen als der Summenfunktion treten noch weitere Probleme auf:

**AVG** Die Berechnung eines Durchschnittes ist nicht möglich, wenn lediglich ein zuvor berechneter Durchschnittswert vorliegt. Dies liegt daran, dass die Durchschnittsfunktion nicht assoziativ ist:

$$\texttt{AVG}(\texttt{AVG}(a,b),c) \neq \texttt{AVG}(a,\texttt{AVG}(b,c)) \neq \texttt{AVG}(a,b,c)$$

Später in dieser Arbeit werden wir sehen, dass die Assoziativität für die inkrementelle Berechnung erfüllt sein muss. Bei der Durchschnittsfunktion lässt sich diese durch die Speicherung von Hilfsdaten, nämlich der Anzahl der aggregierten Elemente, erreichen.

**MIN / MAX** Die Minimum- und Maximum-Funktionen sind ein Beispiel für irreversible Aggregatfunktionen. Sie unterstützen somit keine Löschungen. Wird der aktuell kleinste (bzw. größte) Wert einer Menge entfernt, ist die Bestimmung des Minimums (Maximums) unmöglich. Hier kann ein MIN/MAX-Cache [Cha00] Abhilfe schaffen, in dem die $n$ kleinsten und größten Werte gespeichert werden, um zumindest die Unterstützung von $n$ Löschungen zu bieten.

**COUNT** Nehmen wir an, es soll eine Zählung von Werten $(1, 7, 9)$ vorgenommen werden. Das Ergebnis ist $3$. Soll nun die Zahl $9$ eingefügt werden, ist das Ergebnis nicht $\texttt{COUNT}(3, 9) = 2$ sondern $3+\texttt{COUNT}(9) = 4$. Beim Zählen darf also nicht der Fehler gemacht werden, dass eine vorherige Zählung lediglich als ein einziges Element interpretiert wird. Als Lösung bietet sich an, die Zählfunktion als Summe von Einsen `SUM(1)` zu interpretieren.

**Sonstige** Bei nicht-kommutativen Aggregatfunktionen wie `IMPLODE` (zur Konkatenation von Zeichenketten) kann bei der inkrementellen Berechnung eine Ordnung zerstört werden. Ein Beispiel: Es soll eine Komma-getrennte Liste aller Namen von Personen pro Firma erstellt werden, die nach Gehältern geordnet ist. Ändert sich das Gehalt einer Person, ist unklar, ob und wie der Name in der Liste verschoben werden muss. Darüber hinaus gibt es Aggregafunktionen, in der eine Duplikateliminierung erfolgt. In SQL wird beispielsweise `COUNT(DISTINCT firma)` eingesetzt, um die Anzahl der Firmen zu ermitteln und dabei jede Firma nur einmal zu zählen. Bei einer Löschung tritt ähnlich wie bei MIN und MAX das Problem auf, dass man nicht mehr weiß, ob die letzte Instanz eines Wertes gelöscht wurde oder ob der Wert noch in anderen Datensätzen präsent ist.

### Änderungseinbringung

Bei inkrementellen Berechnungen muss in der Regel auf zwei Datenquellen lesend zugegriffen werden. Zum einen auf die eigentlichen Basisdaten, auf denen auch die Änderungserfassung erfolgt, zum anderen aber auch auf das Ergebnis der vorherigen Berechnung. Das Ziel der inkrementellen Berechnung soll sein, dass das Resultat äquivalent zu dem einer vollständigen Berechnung ist. Betrachten wir die Basisdatensätze $a_1, \ldots, a_j$, die bereits bei einer vorherigen Berechnung das Ergebnis $f(a_1, \ldots, a_j)$ erzeugt haben. Nach der Löschung von $a_i, \ldots, a_j$ und der Einfügung von $a_{j+1}, \ldots, a_k$ gilt die inkrementelle Berechnung unter der Operation $\circ$ als korrekt, genau dann wenn gilt:

$$f(a_1, \ldots, a_j) \circ f^*(a_i, \ldots, a_j) \circ f(a_{j+1}, \ldots, a_k) = f(a_1, \ldots, a_{i-1}, a_{j+1}, \ldots, a_k)$$

Es fällt auf, dass eine Funktion $f^*$ vonnöten ist, um eine zu $f$ inverse Berechnung auszuführen, also um die Auswirkungen einer Berechnung wieder zu kompensieren. Es muss gelten $f \circ f^* = e$, wobei $e$ das neutrale Element bezüglich $\circ$ ist. Des Weiteren ist Assoziativität und Kommutativität von $\circ$ vonnöten, sodass $\circ$, $f^*$ und $e$ auf dem Wertebereich von $f$ eine abelsche Gruppe bilden. Die genaue Definition der Operationen hängt von der Art der Berechnung ab. Einige Beispiele:

- *Keine Aggregation*: Die Berechnung bestehe lediglich aus einer Projektion. Aus jedem Personendatensatz $a_j$ soll das Geburtsjahr $y_j = f(a_j)$ extrahiert werden. Da keine Aggregation erfolgt, ist das Ergebnis eine Multimenge von Geburtsjahren $\{y_1, \ldots, y_j\} = f(a_1, \ldots, a_j)$. Bei der Operation $\circ$ handelt es sich um die Multimengen-Vereinigung $\uplus$, die Funktion $f^*$ fungiert wie $f$, jedoch mit negativen Häufigkeiten. Das neutrale Element ist die leere Multimenge: $f \circ f^* = \emptyset = e$.
- *Summenbildung*: Ist $f(a_1, \ldots, a_j) = \sum_{i \in a_1, \ldots, a_j} i$, handelt es sich bei $\circ$ um die Addition $+$ und es gilt $f^* = -1 \cdot f$.
- *Zählen*: $f(a_1, \ldots, a_j) = \sum_{i \in a_1, \ldots, a_j} 1$ ist ein Spezialfall der Summe, somit ist auch hier $\circ$ die Addition und $f^* = -1 \cdot f$.

Weitere Funktionen wie die Durchschnittsberechnung oder die Summenbildung im Anschluss an eine Projektion unter der Bildung von Gruppierungen stellen Spezialformen der drei genannten Berechnungen dar und können durch ihre Kombination dargestellt werden. Dabei gelten die im vorherigen Abschnitt genannten Einschränkungen. Hier ist nochmals zu erwähnen, dass es zur Berechnung von Minima und Maxima nicht ohne Weiteres möglich ist, $f^*$ zu definieren.

Bei der eigentlichen Berechnung eines Transformationsergebnisses unter Verwendung der sich seit der letzten Berechnung geänderten Basisdaten sowie des vorherigen Ergebnisses gibt es zwei Alternativen, wann das vorherige Ergebnis gelesen wird. Einmal vor dem Beginn der eigentlichen Berechnung, oder alternativ kurz vor dem Schreiben des Ergebnisses. Bei ersterem handelt es sich um die sogenannte *Overwrite-Installation*, bei zweiterem um die *Increment-Installation* [Jör+11b]. Im Folgenden erläutern wir die Alternativen kurz und weisen auf deren Vor- und Nachteile sowie deren Einschränkungen hin.

**Overwrite-Installation** Bei der Overwrite-Installation wird das Ergebnis der vorherigen Berechnung vollständig eingelesen und in eine geeignete Form gebracht, sodass es direkt mit den eigentlichen Basisdaten aggregiert werden kann. Nehmen wir als Beispiel die Berechnung der Gehaltsumme pro Firma. Aus dem Ergebnisdatensatz $(SAP, 1500000)$ wird ein anonymer Personen-Eingabedatensatz erzeugt mit der Firma SAP und dem Gehalt $1500000$. Zu Beginn der Berechnung - im gegebenen Beispiel also vor der Summenbildung - liegen alle zur Ermittlung des Endergebnisses notwendigen Daten vor. Das Endergebnis ist vollständig und beinhaltet alle Endwerte, auch solche, die sich nicht geändert haben. Die Variante trägt den Namen *Overwrite*, weil üblicherweise das vorherige Ergebnis vollständig überschrieben wird. Alternativ könnte man es aber auch in ein separates Ausgabeziel schreiben lassen. Ein Nachteil der Overwrite-Installation ist, dass das komplette vorherige Ergebnis eingelesen werden muss, selbst wenn daran nur sehr wenige Änderungen vorgenommen werden. Der Vorteil ist die universelle Anwendbarkeit und eine hohe Performanz, wenn es viele Änderungen gibt [Sch+14].

**Increment-Installation** Wie der Name schon sagt, beinhaltet diese Variante eine *Inkrement*-Operation. Sie wird meist für die Berechnung von Summe und Anzahl verwendet. Theoretisch ist das Verfahren aber auch auf andere Analysen übertragbar. Die Berechnung läuft unter dieser Variante so ab, dass sie nur auf den Änderungen in den Basisdaten basiert. Das Ergebnis der Berechnung ist also nicht final, sondern stellt nur eine absolute Änderung dar, etwa $(SAP, -17000)$, weil entweder das Gehalt einer bei SAP arbeitenden Person gekürzt wurde oder weil eine ebensolche Person die Firma gewechselt hat oder ganz gelöscht wurde. Dieses Ergebnis darf natürlich nicht wie bei der Overwrite-Installation einfach die Werte in der Datenbank überschreiben, sondern es muss eine Art „Plus-Gleich“ (+=) durchgeführt werden. Das Schreiben läuft also so ab, dass zunächst der vorherige Ergebniswert gelesen werden muss, dann werden die beiden vorliegenden Zahlen addiert und schließlich erfolgt das Zurückschreiben der Summe. Der Vorteil dieser Variante ist, dass nur diejenigen Ergebnisdatensätze gelesen werden müssen, bei denen eine Änderung zu erfolgen hat. Alle anderen bleiben unberührt. Der Nachteil dieser Variante ist, dass die Inkrement-Operation meist sehr kostspielig ist. Während bei der Overwrite-Installation das vorherige Ergebnis sequenziell auf einen Schlag schnell gelesen und zum Ende wieder auf sequenzielle Art als Gesamtes ebenso schnell geschrieben werden kann, handelt es sich bei der Inkrement-Operation um einen wahlfreien Zugriff auf die Daten. Der zu ändernde Datensatz muss also zunächst gefunden, dann gelesen und letztendlich zurückgeschrieben werden. Bei sehr vielen Änderungen kann dieser Prozess deutlich langsamer sein als bei der Overwrite-Installation oder als bei einer vollständigen Neuberechnung.

Bei allen Varianten der inkrementellen Berechnung ist natürlich zu beachten, dass sie nur durchgeführt werden können, wenn auf vorherige Endergebnisse lesend zugegriffen werden kann. Erfolgt das Schreiben der Daten in ein Ziel ohne Leseberechtigung oder in ein Write-only-Datenziel, sind inkrementelle Berechnungen nicht durchführbar. Ein Beispiel dazu ist eine Analyse, deren Ergebnisse per E-Mail versendet werden (es existiert kein Gesendet-Ordner) oder deren Ergebnisse mittels Text-to-Speech aus einem Lautsprecher ertönen.

## 2.4 Datenintegration

In diesem Abschnitt geht es um Vorgänge, bei denen Daten aus mehreren Quellsystemen in einem gemeinsamen Zielsystem integriert werden sollen. Integration bedeutet in diesem Falle, dass es nicht damit getan ist, die Daten lediglich eins zu eins zu kopieren, sondern dass zur Gewährung der Integrität komplexere Schritte nötig sind. Im Allgemeinen geht es dabei um die *Überwindung von Heterogenität* [LN07]. Werden unterschiedliche Softwaresysteme zur Datenspeicherung im Quell- und Zielsystem eingesetzt, unterscheiden diese sich typischerweise in ihren Zugriffsmethoden und in ihren Datenmodellen. Geeignete Datenbank-Middleware kann dazu eingesetzt werden, diese sogenannte *technische Heterogenität* und die *Datenmodell-Heterogenität* zu überwinden. Anderen Formen von Heterogenität resultieren aus der *Autonomie* beim Schema-Design. Verschiedene Datenbankentwickler modellieren die gleichen Reale-Welt-Konzepte mit anderen Datenmodellkonzepten oder unter der Verwendung unterschiedlicher Terminologien und Datentypen. Im Folgenden gehen wir auf insgesamt sechs Arten von Heterogenität ein [LN07, Kapitel 3]. Weiter unten werden zwei Möglichkeiten zur Datenintegration vorgestellt und diskutiert, nämlich die materialisierte und die virtuelle Integration.

### Formen von Heterogenität

**Technische Heterogenität** Betrachtet man zwei verschiedene Datenbankmanagementsysteme, bieten diese in der Regel zwei unterschiedliche Zugriffsschnittstellen (APIs). Dazu zählen Methoden zum Verbindungsaufbau, zum Lesen und Schreiben von Daten sowie zur Steuerung von Transaktionen. In relationalen Datenbanken kommen zur Überwindung der technischen Heterogenität *Datenbank-Gateways* wie zum Beispiel *JDBC* (Java Database Connectivity) zum Einsatz. Diese machen es Entwicklern möglich, mit gewohnten Methoden auf ein fremdes Datenbankmangementsystem zuzugreifen, ohne erst eine neue API zu verstehen. Des Weiteren kann in einer Anwendung schnell von einem System auf ein anderes gewechselt werden, indem lediglich ein anderer Datenbanktreiber zum Einsatz kommt. Bei NoSQL-Datenbanken kommen DB-Gateways aktuell in der Regel noch nicht zum Einsatz. Das bedeutet, dass jede Interaktion mit der Datenbank

neu formuliert werden muss, wenn von einem System auf ein anderes gewechselt wird. Bei Datenintegrationen ist die Überwindung technischer Heterogenität meist einfach zu lösen, da dieser Prozess generisch für ein Datenbankmanagementsystem erfolgen kann. Typischerweise erfolgt die Überwindung mittels der Entwicklung von *Wrappern*. Ein Wrapper ist eine speziell für ein System entwickelte Komponente, welches zum Lesen und Schreiben von Daten eingesetzt werden kann. Nach dem Lesen bringt der Wrapper die Daten in ein einheitliches internes Format, aus welchem vor dem Schreiben wieder das Datenbank-spezifische Format erzeugt wird.

**Datenmodell-Heterogenität** Zwar gibt es viele verschiedene relationale Datenbankmanagementsysteme, im Großen und Ganzen verwenden sie jedoch das gleiche Datenmodell: Tabellen, Spalten, Zeilen, etc. Trotzdem gibt es aber auch hier oft Unterschiede, zum Beispiel die Unterstützung komplexer Datentypen. Eine weiter ausgeprägte Form von Datenmodell-Heterogenität liegt allerdings bei der Verwendung zweiter verschiedener Datenbank-Klassen vor. Dazu betrachte man nur die Definitionen der verschiedenen Datenmodelle in NoSQL-Datenbanken in Abschnitt 2.1. Bei der Datenintegration kann die Überwindung der Datenmodell-Heterogenität leicht sein, wenn das Datenmodell im Zielsystem ein Über-Modell des Quellsystem-Datenmodells darstellt. Als Beispiel können wir die Migration von Dokumenten in CouchDB in die Graphdatenbank Neo4J betrachten. Da Neo4J ein JSON-ähnliches Format für die Knotenproperties verwendet, können einfach alle JSON-Dokumente aus CouchDB als Knoten übernommen werden. In der Rückrichtung ist die Datenintegration äußerst problematisch, da Kanten - also Beziehungen zwischen Datensätzen - nicht in Dokumentendatenbanken unterstützt werden. Für solche Zwecke muss man sich individuell eine Abbildung im Ziel-Datenmodell überlegen, beispielsweise in diesem Fall über einen Array, der Kanten-Labels, -Properties und die IDs der Zieldatensätze enthält.

**Syntaktische Heterogenität** Wenn es darum geht, Datenwerte zu speichern, gibt es meist viele alternativen Formen. Ein Kalender-Datum kann beispielsweise als eine mit Punkten getrennte Folge von Tag, Monat und Jahr in Form eines Strings gespeichert werden oder aber in Form eines Unix-Zeitstempels als ganze Zahl. Eine weitere Alternative wäre die Verwendung eines speziellen Datentyps, etwa der SQL-Typ `DATE`. Andere Fälle syntaktischer Heterogenität treten bei der Verwendung von verschiedenen Zeichenkodierungen, Umlaut-Repräsentationen, Zahlenformaten oder Maßeinheiten auf. Im Allgemeinen lässt sich diese Form der Heterogenität allerdings leicht mittels einfacher Umrechnungsfunktionen oder regulärer Ausdrücke überwinden.

**Strukturelle Heterogenität** Ein Datenbankdesigner muss sich bei seiner Arbeit stets die Frage stellen, welche Konzepte des Datenmodells für welche Reale-Welt-Konzepte verwendet werden sollen. Verwaltet eine Datenbank die Produkte eines Webshops, könnte man in einer relationalen Datenbank die Produktkategorie mit dem Datenmodell-Konzept „Spaltenwert“ umsetzen. Eine Spalte in der Produktdatenbank enthielte also in diesem Fall den Namen der Kategorie, in der sich das Produkt befindet, z. B. `"buch"`. Eine Alternative wäre die Verwendung des Konzepts „Tabelle“. In diesem Fall würden alle Produkte der Buch-Kategorie in eine Tabelle `buch` gespeichert. Man beachte, dass in diesem Beispiel die strukturelle Heterogenität so sehr ausgeprägt ist, dass ein Daten-Metadaten-Konflikt auftritt. In der ersten Alternative wurde die Kategorie auf Datenebene, in der zweiten auf Metadatenebene umgesetzt. Um ein Produkt mit einer neuartigen Kategorie hinzuzufügen, ist im einen Fall ein einfaches `INSERT`, im anderen Fall ein zusätzliches `CREATE TABLE` vonnöten. In NoSQL-Datenbanken gibt es aufgrund der Schema-Flexibilität und der Unterstützung komplexer Typen weitere Ursachen für strukturelle Heterogenität. In MongoDB könnte beispielsweise jede Kategorie ein Dokument sein, welches einen Array aller in ihr enthaltenen Produkte beinhaltet. Zur Überwindung von struktureller Heterogenität sind Methoden oder Sprachen notwendig, die es erlauben, Metadaten in Daten und Daten in Metadaten umzuwandeln.

**Schematische Heterogenität** Verwendet man zur Umsetzung ein und des selben Reale-Welt-Konzepts in zwei Implementierungen das gleiche Datenmodell-Konzept, können trotzdem zwei heterogene Schemata vorliegen. In diesem Fall sprechen wir von schematischer Heterogenität. Nehmen wir an, ein Datenbankdesigner entscheidet sich dafür, den Verlagsort eines Buches in der Produktdatenbank als Spaltenwert zu speichern. In einem Fall kann dies eine Spalte der Produkttabelle sein, im anderen Fall eine Spalte in einer Verlagstabelle, welche über eine Fremdschlüsselbeziehung mit der Produkttabelle verbunden ist. Letztere Modellierung stellt eine Normalisierung der ersteren dar und vermeidet Redundanzen und Anomalien. In beiden Fällen ist der Verlagsort auf Datenebene gespeichert. Die Abbildung des einen Schemas in das andere ist mittels einer Verbundoperation bzw. umgekehrt mit einer Selektion und Projektion möglich. Im Allgemeinen kann die schematische Heterogenität meist mit einfachen Operatoren überwunden werden.

**Semantische Heterogenität** Während die bisher vorgestellten Formen der Heterogenität damit zu tun hatten, wie die Konzepte der realen Welt in einem speziellen Datenbanksystem umgesetzt werden, ist die semantische Heterogenität eine Folge der Autonomie bei der Verwendung von Namen und Terminologien. Auf Metadatenebene gilt es beispielsweise zu entscheiden, ob Attribute deutsche oder englische Namen erhalten, ob Abkürzungen verwendet werden oder ob sie lediglich durchnummeriert werden. Letztere Alternative ist äußerst unelegant, aber erstere Schemata können mittels Wörterbücher und Thesauri ineinander überführt werden. Homonyme und Synonyme sorgen dabei allerdings oft für Unklarheiten. Im Allgemeinen kann man sagen, dass die semantische Heterogenität die am schwierigsten zu überwindende Form darstellt, da vor allem in automatischen Prozessen schwierig festgestellt werden kann, was sich ein Datenbankdesigner

mit der Vergabe eines Namens gedacht hat. Für die Überwindung werden Schema-Matching-Werkzeuge [MBR01; RB01] eingesetzt, welche Korrespondenzen zweier heterogener Schemata erkennen. Ein solches Werkzeug betrachtet dabei die Namen der Schemaelemente, die verwendeten Datentypen und die Struktur, wie die Schemaelemente untereinander angeordnet sind. Zusammengesetzte und hybride Schema-Matcher schauen dabei nicht nur auf ein Kriterium sondern kombinieren verschiedene Ansätze. Zusätzlich ist eine Untersuchung über das Schema hinweg, also auf Instanzebene möglich. Mit dieser Methode wäre es sogar möglich, eine Korrespondenz zwischen zwei Attributen mit kryptischen Namen herauszufinden, wenn sie ähnliche Werte, z. B. Produktkategorien oder Personennamen, enthalten.

### Materialisierte und virtuelle Integration

Die weiter oben beschriebenen periodisch ausgeführten Datentransformationen und Analysen mit den dazugehörigen Änderungserfassungsmethoden entsprechen der *materialisierten Integration* [LN07, Kapitel 4]. Materialisiert bedeutet, dass das Ergebnis tatsächlich in einer Datenbank gespeichert wird. Die Alternative dazu ist die *virtuelle Integration*, bei der das Ergebnis lediglich eine Sicht auf die Basisdaten darstellt. Handelt es sich bei der Integration um eine Aufgabe, die nur ein einziges mal ausgeführt werden muss, beispielsweise die Übernahme von Kundendaten in ein neues System, verwendet man dazu die materialisierte Integration. Ändern sich jedoch die Basisdaten ständig und es ist vonnöten, dass die Änderungen im Zielsystem sofort sichtbar sind, kann eine virtuelle Integration sinnvoll sein.

In einem *Data-Warehouse* findet man die materialisierte Integrationen in der Form von *ETL*-Jobs wieder. Die drei Phasen des *Extract-Transform-Load* bestehen daraus, Daten aus einem Basissystem zu extrahieren, diese so zu transformieren, dass sie sich für typische Datenanalyseaufgaben eignen, und das Ergebnis schließlich ins Datenwarenhaus zu laden. Temporäre Zwischenergebnisse befinden sich in der *Staging-Area*, auf welche in allen drei ETL-Phasen lesend und schreibend zugegriffen werden kann. Die Eingabedaten kommen aus verschiedenen Quellen, bei einem Online-Shop beispielsweise Produktdaten, Einkaufsstatistiken und Webseiten-Besucher-Logs. Komplexe Analysen, etwa um Einkaufsverhalten vorherzusagen oder Berichte zu erzeugen, dauern oft mehrere Minuten oder Stunden und werden deshalb üblicherweise nicht direkt auf den Basisdaten ausgeführt. Stattdessen erfolgt in periodischen Zeitintervallen die Integration der Daten in das Datenwarenhaus, auf welches anschließend zu Analysezwecken nur lesend zugriffen wird. Da im Transform-Schritt die Daten bereits voraggregiert und mit Join-Operationen denormalisiert werden können, sind die letztendlichen Analysen deutlich effizienter, als wenn man sie direkt auf den Basisdaten ausführen würde. Des Weiteren beeinflussen sie die Verfügbarkeit und Performanz des eigentlichen Systems nicht. Bei direkten Datenanalysen würden lang-gehaltene Sperren zu einer drastischen Verlangsamung parallel laufender Transaktionen führen. Die Ausführung des ETL, also des Vorgangs, der nötig ist, um die Daten ins Datenwarenhaus zu bekommen, dauert allerdings je nach Datenmenge, Wahl des Änderungserfassungsverfahrens und Ausmaß an Änderungen seit einer vorherigen

Ausführung eine gewisse Zeit. Um die Daten im Datenwarenhaus aktuell zu halten sind wiederholte Ausführungen vonnöten.

Bei virtuellen Integrationen liegen die Daten stets im aktuellen Zustand vor. *Stale Data*, also veraltete Datenwerte, können aufgrund der direkten Weiterleitung der Leseaktionen zu den Basisdaten nicht auftreten. Des Weiteren sind keine ETL-Jobs vonnöten, die über den kompletten Datenbestand iterieren, und es wird kein zusätzlicher Speicherbedarf benötigt. Eine virtuelle Integration liegt als eine Art „Kochrezept“ vor. Wenn eine Anwendung einen Datensatz im integrierten Schema anfordert, wird das Rezept gemäß der Anfrage angepasst und schließlich direkt auf den Basisdaten ausgeführt. Bereits durch einfache Optimierungen wie die Ausführung eines Predicate-Pushdows, also der Filterung der Daten direkt an der Quelle, können einfache virtuelle Transformationen sehr effizient ausgeführt werden. Allerdings sind komplexen Analysen, in denen große Datenmengen gelesen und aggregiert werden, bei der virtuellen Integration deutlich langsamer als bei der materialisierten.

# 3 Transformationssprachen für NoSQL

Anders als bei relationalen Datenbanken gibt es in der Welt heterogener NoSQL-Datenbanken keine einheitliche Anfragesprache. Die im vorherigen Kapitel vorgestellten Programme und Frameworks machen es möglich, komplexe Analysen effizient und verteilt auszuführen, jedoch ist die Verwendung meist aufwendig und mit einer gewissen Einarbeitungszeit verbunden. Die Alternative zum Schreiben einer Analyse in einer Programmiersprache wie Java ist die Verwendung einer speziellen Anfrage- und Transformationssprache, wie sie in diesem Kapitel vorgestellt werden.

## 3.1 SQL-basierte Sprachen

Die Structured Query Language SQL ist mächtig und wird von einer großen Benutzergruppe beherrscht. Auch in der Programmierung unerfahrene Anwender sind oft im Formulieren von SQL-Anfragen auf relationalen Datenbanken geübt. Deshalb verfolgen viele Ansätze die Idee, die Sprache SQL als Basis zu nehmen und sie so zu erweitern, dass sie zur Definition von Anfragen auf NoSQL-Datenbanken genutzt werden kann. Dadurch wird es möglich, komplexe Anfragen in kurzen und leicht verständlichen SQL-Statements zu formulieren, ohne dass sich der Anwender um die Optimierung und die verteilte Ausführung mittels eines Datenverarbeitungsframeworks kümmern muss. Auf der anderen Seite ist die Sprache SQL jedoch stark für relationale Datenbanken optimiert. Es gibt also keine einfache Unterstützung für flexible Schemata oder komplexe Datentypen. Die im Folgenden vorgestellten Ansätze versuchen dieses Manko teilweise durch die Erweiterung der Sprache zu kompensieren.

**Hive** *Apache Hive* [Thu+09] ist eine Data-Warehouse-Erweiterung für Hadoop [Apaa]. Ursprünglich wurde sie von Facebook entwickelt und im Jahre 2008 der Open-Source-Gemeinde zur Verfügung gestellt [Hiv]. Die Sprache *HiveQL*, die den SQL'92-Standard zwar nicht vollständig erfüllt, sich aber sehr stark daran orientiert und ihn teilweise auch erweitert, ermöglicht es, relationale Tabellen anzulegen, welche in dem verteilten Dateisystem *HDFS* (*Hadoop Distributed File System*) gespeichert werden. Die SQL-ähnlichen Anfragen werden in der ursprünglichen Hive-Implementierung nach einer Optimierung in eine Folge von MapReduce-Jobs umgewandelt und schließlich mittels des Hadoop MapReduce Frameworks ausgeführt. Einfache Selektionsanfragen resultieren in einem einzigen Map-Job. Anfragen, in denen zwei Tabellen über eine Joinoperation miteinander verbunden wurden, werden zu einem Job mit einer Map- und einer Reduce-Phase. Kommen zusätzliche Gruppierungen, Aggregationen, Sortierungen oder weitere Verbünde hinzu, kann die HiveQL-Anfrage nicht mehr als ein einziger MapReduce-Job durchgeführt werden, sondern als eine Verkettung mehrerer solcher. Durch die Unterstützung sogenannter Engines ist es möglich, *TEZ* [Sah+15] oder *Spark* [Zah+10] anstelle von MapReduce einzusetzen. Zur Speicherung der Tabellendaten unterstützt Hive mehrere Dateiformate, unter anderem CSV-Textdateien sowie das optimierte *ORC*-Format (*Optimized Row Columnar* [Orc]). In letzterem werden beispielsweise Tabellendaten in sogenannte Stripes aufgeteilt und innerhalb jedes Stripes werden die kleinsten und größten Werte zu jeder Spalte vermerkt. Dies ermöglicht ein effizienteres Durchsuchen. Zusätzlich zu den nativen Speicherformaten werden in Hive auch externe Tabellen unterstützt, also Tabellen, deren Daten in einem fremden System liegen. Dies kann zum Beispiele eine HBase-Tabelle sein, was es möglich macht, SQL-Anfragen an existierende HBase-Tabellen zu stellen [Hiv]. Allerdings muss die HBase-Tabelle ein festes Schema besitzen, welches man beim Erstellen der Hive-Tabelle zusammen mit den Datentypen der einzelnen Spalten definieren muss.

**Impala** Da Hive ursprünglich für die Ausführung mittels MapReduce entwickelt wurde und in Verbindung mit HBase sehr inperformant war, entstanden weitere Systeme, die ebenfalls den Gedanken verfolgten, Datenmengen verteilt zu speichern und per SQL abrufbar zu machen. Die von der Firma Cloudera entwickelte Datenbank-Engine *Apache Impala* [Kor+16] baut – genau wie Hive – auf den Hadoop-Technologien auf, also dem verteilten Dateisystem HDFS sowie dem Wide-Column Store HBase. Allerdings werden die an Impala gestellten SQL-Anfragen nicht zuerst in schwergewichtige MapReduce-Anfragen gewandelt, sondern direkt von Impala-Prozessen, die auf jedem Rechnerknoten installiert sind, verarbeitet. Impala unterstützt nahezu alle Eigenschaften des SQL'92-Standards sowie einige analytische Erweiterungen aus SQL'2003 und benutzerdefinierte externe Funktionen. Da Impala seine Daten üblicherweise im HDFS speichert, ist es hauptsächlich dazu gedacht, einmal geladene große Datenmengen zu analysieren, und nicht um Änderungen oder Löschungen auf diesen auszuführen.

**Phoenix** Bei *Apache Phoenix* [Apad] wurde von Anfang an das Ziel verfolgt, HBase zur Speicherung zu verwenden, während dies bei der Entwicklung von Hive und Impala nur zweitrangig war. Bei der Arbeit mit Tabellen pflegt Phoenix im Hintergrund viele Hilfsstrukturen wie Metadaten-Tabellen und Trigger, die ACID-Transaktionen sowie das Anlegen und Nutzen von Indexen ermöglichen. Phoenix-Tabellen können mit einem `CREATE TABLE`-Kommando angelegt werden oder auf eine existierende HBase-Tabelle abgebildet werden. Eine Funktionalität, welche über den SQL-Standard hinaus geht, ist die Unterstützung von *Schema-on-Read*. Dies erlaubt es Anwendungen, in Anfragen Namen von Spalten zu verwenden, welche beim Anlegen einer Tabelle nicht definiert wurden. Flexible Schemata werden also von Phoenix unterstützt, allerdings gibt es keine Möglichkeit auf Spalten zuzugreifen, deren Namen der Anwendung unbekannt ist. Phoenix verwendet die nativen API-Methoden von HBase und setzt nicht auf MapReduce. Durch Server-seitige Berechnungen mittels Koprozessoren werden Berechnungen und Filter so weit es geht auf denjenigen Rechnern bearbeitet, auf denen die Daten liegen, und somit die Netzwerklast minimiert. Das Resultat ist, dass die Geschwindigkeit von Phoenix um ein vielfaches höher ist als die von Hive oder Impala. Eine Test-Anfrage, welche in einer Tabelle mit zehn Millionen Zeilen die zehn höchsten Werte einer bestimmten Spalte findet, dauerte in einem Test mit Hive über HBase 46 Sekunden und mit Phoenix nur sieben Sekunden [Pho].

**MRQL** Obwohl die Abkürzung *MRQL* (ausgesprochen: *miracle*) [FLG12] für „MapReduce Query Language“ steht, ist das Ziel dieser Sprache, unabhängig von einer bestimmten Ausführungsplattform zu sein, und stattdessen dem Benutzer die Ausführung von Anfragen auf vielen verschiedenen Plattformen zu ermöglichen. Neben Apache Hadoop zählen dazu auch Spark, Flink und Apache Hama [Seo+10]. MRQL orientiert sich zwar an SQL, unterscheidet sich jedoch davon sehr stark. Das Datenmodell wurde um verschachtelte Strukturen und Bäume erweitert, um auch Formate wie JSON oder XML zu unterstützen. Als Eingabe dienen Dateien, welche sich im Hadoop Distributed File System (HDFS) befinden. Die Sprache MRQL kann als mächtige Variante von Hive gesehen werden. Sie unterstützt nicht nur komplexere Datenmodelle, sondern bietet auch neue Sprachkonstrukte, beispielsweise für iterative Berechnungen.

**BigQuery** Der auf *Dremel* [Mel+11] basierende Web-Dienst *Google BigQuery* [Sat12; Goo15] dient zur Ad-Hoc-Analyse von großen Datenmengen. Mittels SQL ist es möglich, Lese-Anfragen an im Google Cloud Storage befindliche CSV-, JSON- oder Avro-Dateien [Apa15a] zu stellen, welche auf großen Rechenclustern mit tausenden von Knoten parallel ausgeführt werden. BigQuery erweitert SQL um die Unterstützung für verschachtelte Attributwerte. Wenn beispielsweise die Eingabedaten im JSON-Format vorliegen und dort Attribute Unterattribute besitzen, kann auf diese mittels Dot-Notation zugegriffen werden, z. B. `WHERE person.firma.ort = '...'`. Darüber hinaus bietet BigQuery die Unterstützung für sich wiederholende Attribute, was gewisse Ähnlichkeiten zu Arrays hat. Eine im Schema angegebene `repeated`-Option erlaubt das mehrfache Vorkommen des gegebenen Attributs. Bei der Anfrage wird das Ergebnis „flachgeklopft“.

Ein `SELECT name, hobby FROM person` liefert zu jedem Personen-Datensatz mehrere Zeilen, wenn in diesem das `hobby`-Attribut mehrfach präsent ist. Während die SQL-basierte Anfragesprache sehr mächtig ist und große Datenmengen in wenigen Sekunden analysiert werden können, werden Schreiboperationen auf den Daten von BigQuery nicht unterstützt.

**N1QL** N1QL (ausgesprochen: *Nickel*) [OHR15] ist die zur Dokumentendatenbank Couchbase [Cou] gehörende Anfragesprache. N1QL basiert auf SQL, verwendet jedoch als Datenmodell JSON-Dokumente. Für den Zugriff auf Unterattribute wird wie bei BigQuery die Dot-Notation verwendet. Sprachkonstrukte zum Zugriff und zum Kreieren von Arrays und Subdokumenten sind auf das JSON-Datenmodell abgestimmt. Gleichzeitig lässt sich die Sprache auch für relationale Datenbanken verwenden, da sie auf SQL basiert. Mittels `INSERT`-, `UPDATE` und `DELETE`-Anfragen kann N1QL auch zum Modifizieren von Daten verwendet werden.

**OQL** Die *Object Query Language* [ASL89] ist eine durch die Object Database Management Group standardisierte Abfragesprache für Objektdatenbanken. Da Objekte in solchen Datenbanken mehrwertige Attribute in Form von Arrays und Listen, sowie Referenzen zu anderen Objekten besitzen können, wurden in OQL entsprechende Möglichkeiten eingebaut, die über die Mächtigkeit von SQL hinausgehen. Mittels Dot-Notation und Unteranfragen kann Referenzen gefolgt werden. Ein Beispiel: `FROM Person p, p.freunde f`. Die Sprache hatte Einfluss auf die *Enterprise Java Beans Query Language EJB-QL*, welche ebenfalls ein direktes Traversieren von Referenzen ohne die Verwendung von Joins erlaubt.

**FIRA/FISQL, SchemaSQL** Die meisten der bisher vorgestellten Sprachen boten Spracherweiterungen für SQL, um vom relationalen Datenmodell abweichende Strukturen anfragen und erstellen zu können. Wie jedoch im vorherigen Kapitel beschrieben, gilt als eine Herausforderung beim Umgang mit Big Data auch die Heterogenität von Daten. Da viele NoSQL-Datenbanken flexible Schemata ermöglichen, gewinnen Sprachen an Wichtigkeit, welche mit dieser Schemaflexibilität umgehen können. Beim Stellen einer Anfrage in SQL muss nämlich immer das Schema der angefragten Tabellen bekannt sein. Die einzige Ausnahme ist ein `SELECT *`. Es gibt jedoch keine Möglichkeit, Drop-Projektionen durchzuführen, also alle Spalten zu bekommen, bis auf eine gegebene Liste. Des Weiteren ist es in SQL-Anfragen nicht möglich, Spalten- oder Tabellennamen abzufragen, Daten in Metadaten zu wandeln oder den Wert einer Spalte zu lesen, deren Namen als Wert in einer anderen Spalte steht. Genau für solche Anfragen wurde die relationale Algebra sowie SQL um spezielle Operationen erweitert. Die *Federated Interoperable Relational Algebra* (*FIRA*) [WR05], die *Federated Interoperable Structured Query Language* (*FISQL*) [WR05] sowie *SchemaSQL* [LSS96] zählen zu den sogenannten Schema-Anfragesprachen. Im Laufe dieser Arbeit werden wir bei einzelnen Operatoren auf die korrespondierende Konstrukte aus den genannten Sprachen aufmerksam machen.

## 3.2 Systemübergreifende Sprachen

Mittels systemübergreifender Sprachen ist es möglich, auf verschiedene autarke Systeme auf einmal zuzugreifen. Der *SQL/MED Standard* [Mel+02] legt fest, wie innerhalb einer SQL-Anfrage auf Daten von fremden Datenbanksystemen zugegriffen werden kann. Mittels Join-Operationen können sogar Daten aus verschiedenen Systemen kombiniert werden. Eine Implementierung von SQL/MED ist beispielsweise die Komponente der *Foreign Data Wrappers* von PostgreSQL [For]. Benutzer können Wrapper für verschiedene Systeme – dazu zählen auch NoSQL-Datenbanken, Dateiformate und Cloud-Dienste – erstellen und der Community zur Verfügung stellen. Mittels dieser Wrapper und dem Kommando `CREATE FOREING TABLE` wird eine Verbindung zu einer externen Quelle hergestellt. In SQL-Anfragen ist es dann möglich, auf diese fremden Tabellen, genau wie auf lokale Tabellen und Sichten, zuzugreifen. Da am Ende die Sprache SQL ohne spezielle Erweiterungen zum Einsatz kommt, ist es die Aufgabe der Wrapper, die Datenmodelle der verschiedenen Systeme in das relationale Modell zu überführen. Auch *Garlic* [RS97] und *Apache Calcite* [Cal] (vormals bekannt unter dem Namen *Optiq*) verfolgen den Ansatz, mittels Wrappern bzw. Adaptern und der Sprache SQL auf Daten, die auf fremden Systemen liegen, zuzugreifen. In [RLMW14] wird eine erweiterbare Middleware vorgestellt, die eine SQL-Anfrage entgegennimmt und in die Anfragesprache eines NoSQL-Systems übersetzt, sodass diese dort direkt ausgeführt werden kann. Im Gegensatz zu Garlic unterstützt diese Middleware auch Einfügungen und Änderungen, allerdings sind keine Joins möglich. Des Weiteren können in den SQL-Anfragen nur solche Operationen zum Einsatz kommen, die vom verwendeten System nativ unterstützt werden, also zum Beispiel keine Selektionen auf Werten in Key-Value-Datenbanken. Diese Middleware sowie SQL/MED, Garlic und Calcite stoßen bei komplexen Typen und bei allem, was vom relationalen Modell abweicht, an ihre Grenzen. Die nun folgenden weiteren Ansätze ermöglichen mächtigere systemübergreifende Anfragen und Transformationen auf NoSQL-Datenbanken. Viele davon erweitern wie die im vorherigen Abschnitt vorgestellten Sprachen die Sprache SQL, um komplexe Datenstrukturen zu unterstützen. Andere verwenden eine von SQL abweichende Syntax.

**SQL++** Die Sprache *SQL++* [OPV14] verfolgt gleich mehrere Ansätze. Zum einen dient auch hier SQL als Basis, zum anderen werden verschiedenartige Eingabequellen unterstützt und somit Möglichkeiten zur Überwindung technischer Heterogenität und Datenmodellheterogenität geboten. SQL++ besteht aus der SQL++-Anfragesprache sowie zwei Kernkomponenten, nämlich einer virtuellen Datenbank und einem Anfrageverarbeiter. Durch die Definition von Views in der virtuellen Datenbank können mittels Wrappern verschiedene Systeme mit unterschiedlichen Datenmodellen angebunden werden. Die Komponente zur Anfrageverarbeitung nimmt Anfragen entgegen und führt diese direkt auf den externen Datenquellen aus. SQL++ verwendet als internes Datenmodell ein semi-strukturiertes Format mit Mengen, Listen, Verschachtelungen und mehr, also eine Übermenge vom relationalen Modell und JSON. Die Sprache ist abwärtskompatibel zu SQL und kann auch auf relationalen Datenbanksystemen eingesetzt werden. Der dazu notwendige SQL Wrapper bietet eine Sicht auf die Tabellendaten als eine

Menge von Tupeln. Für Unterattribute bei verschachtelten Strukturen wird in SQL++ die Dot-Notation verwendet, zur Unterstützung von Arrays und Mengen wurden Erweiterungen in die `FROM`-Klausel eingebaut. Mit diesen können in der `FROM`-Klausel nicht nur Datenquellen aufgelistet werden, sondern auch Namen von mengenwertigen Attributen. Mittels `FROM person AS p, p.hobbys AS h` wird ähnlich wie in OQL der Hobbys-Array in der Personentabelle in eine flache Struktur gebracht. Anders als im SQL-Standard kann direkt auf die in der `FROM`-Klausel definierten Aliasse zugegriffen werden: `SELECT p` gibt den vollständigen Personendatensatz zurück und `SELECT h` jeden einzelnen Wert im Hobby-Array. Um Metadaten wie zum Beispiel Spaltennamen in Daten umzuwandeln, gibt es das Sprachkonstrukt `SELECT ATTRIBUTE n:v`. Da SQL++ jedoch primär für die Verwendung als Anfragesprache in Applikationen und nicht als Datentransformationssprache gedacht ist, gibt es keine Möglichkeiten, Daten in Metadaten zu wandeln oder auf die Datenquellen schreibend zuzugreifen.

**CloudMdsQL** Die *Cloud Multidatastore Query Language* [Kol+15] verfolgt die Idee, innerhalb einer SQL-Anfrage die für ein verwendetes Datenbanksystem typische Anfragesprache zu verwenden. Dadurch werden systemübergreifende Anfragen ermöglicht und dank der Einbettung nativer Sprachen die technische Heterogenität und Datenmodell-Heterogenität überwunden. Anders als bei SQL++ ist bei CloudMdsQL das Ergebnis einer Anfrage stets eine Menge von Tupeln. Die Sprache kann nur zum Abfragen, aber nicht zum Erzeugen komplexer Datenstrukturen verwendet werden.

**Toad** Das zur Administration von Oracle Datenbanken entwickelte Programm *Toad* [Toa11] bietet mit einer Erweiterung für Cloud-Datenbanken die Möglichkeit, auf externe Datenbanken zuzugreifen. Im Gegensatz zu SQL++ unterstützt Toad auch das Einfügen, Ändern und Löschen auf externen Quellen. Diese werden genau wie Leseoperationen von einem *Data Hub* in API-Befehle des zugrunde liegenden Systems übersetzt. Da jedoch alle Anfragen auf ANSI SQL basieren, gibt es keine Unterstützung für komplexe Datentypen, nicht-relationale Datenmodelle oder flexible Schemata.

**BQL** Die *Bridge Query Language* [Cur+11] ist eine generische Sprache, die sich auf die speziellen Datenmodelle verschiedener NoSQL-Datenbanken anpassen lässt. Der in [Cur+11] vorgestellte Prototyp nimmt eine SQL-Anfrage entgegen, welche anschließend in eine Menge von BQL-Anfragen übersetzt wird – eine für jede in der Anfrage verwendete NoSQL-Datenbank. Für die unterschiedlichen Datenbanksysteme müssen Übersetzer geschrieben werden, welche BQL-Anfragen in die systemspezifische Sprache umwandeln. Das Ziel von BQL ist eine virtuelle Datenintegration, um Anwendungen eine globale Sicht auf die Daten aus verschiedenen Quellen zu bieten. Die Sprache bietet Möglichkeiten zur Selektion, Projektion und zum Join, sowie Sprachkonstrukte zur Anfrage von Metadaten.

## 3.3 Pipeline-basierte Sprachen

SQL und andere deklarative Sprachen verfolgen die Grundidee, das *Was* und nicht das *Wie* zu formulieren. Pipeline-basierte Sprachen dagegen beabsichtigen, dass der Benutzer als eine Folge von *Schritten* angibt, *wie* das Endergebnis berechnet werden soll. Der erste Schritt ist üblicherweise eine Definition woher die Daten stammen, der letzte Schritt eine Ausgabe als Objekt oder als Cursor, oder aber eine Angabe, wohin die Ergebnisse geschrieben werden sollen. Dazwischen legen weitere Schritte fest, wie die Daten gefiltert, transformiert oder gruppiert werden sollen. Pipeline-basierte Sprachen sind sehr ähnlich zu einer Programmiersprache, da das Programm von oben nach unten abgearbeitet werden kann. Meist gibt es jedoch keine Kontrollstrukturen oder Schleifen. Eine Pipeline-basierte Sprache ist im Gegensatz zu einer generischen Programmiersprache für einen ganz bestimmten Zweck und ein bestimmtes Datenmodell konzipiert. Von einem Pipeline-Schritt zum nächsten gibt es üblicherweise keine Seiteneffekte und es herrscht Abgeschlossenheit; das Datenmodell wird also nicht verlassen. Letzteres ermöglicht es, flexibel einzelne Schritte hinzuzufügen oder zu entfernen. Dies macht auch die Arbeit mit der Sprache einfach. Nach jedem Schritt kann sich der Benutzer anschauen, wie die Daten bis dort hin aussehen, bevor man mit der Definition der darauf folgenden Schritte fortfährt.

**Apache Pig** Die beiden Sprachen *Pig* [Ols+08] und *Hive* [Thu+09] werden oft zusammen genannt, obwohl sie sehr verschieden sind. Hive basiert auf SQL und ist deklarativ, Pig ist Pipeline-basiert. Der Grundgedanke bei beiden Sprachen war jedoch der gleiche: Als Alternative dazu, dass Benutzer aufwändige Hadoop-Jobs programmieren, sollen sie ihre Programme in einer kompakten Sprache formulieren, welche im Anschluss in eine Folge von MapReduce-Jobs umgewandelt und auf dem Hadoop-Framework ausgeführt werden. Ein Pig-Programm beginnt meist mit einer `LOAD`-Anweisung, um die Daten aus einer Datei oder einer Datenbank zu lesen. Das Ergebnis jeder Anweisung wird in Variablen geschrieben, auf welche in den folgenden Schritten zugegriffen werden können. Einige mögliche weitere Anweisungen sind Filter, Gruppierungen, Projektionen und Joins. Das Pig-Datenmodell ist eine Kombination aus Multimengen, Tupeln, Maps und atomaren Typen. Beim Lesen einer CSV-Datei entsteht beispielsweise ähnlich wie bei relationalen Datenbanken eine Multimenge von Tupeln. Die Tupel haben jedoch nicht zwangsweise ein vorgegebenes Schema und deren Attribute können auch unbenannt sein. In letzterem Falle wird auf Attributwerte mittels `$0`, `$1`, u. s. w. zugegriffen. Pig-Programme werden vor der Kompilierung in MapReduce optimiert, um nicht benötigte Attribute früh zu entfernen, Filter möglichst früh auszuführen und um eine geschickte Join-Reihenfolge zu wählen. Anders als bei den zuvor vorgestellten Sprachen können in Variablen befindliche Zwischenergebnisse in Pig mehrfach wiederverwendet werden. So kann ein einzelnes Programm in mehrere Ausgabedateien oder -datenbanken auf einmal schreiben. Pig ist gut geeignet für komplexe Datentransformationen und ermöglicht dank unterschiedlicher *Loader* auch systemübergreifende Transformationen. Das Datenmodell der Quellen muss jedoch zum Pig-Datenmodell passen. Des

Weiteren sind die Möglichkeiten zum Umgang mit flexiblen Schemata eingeschränkt.

**Nova** Oft werden einzelne Anfragen oder Skripte zu komplexen Workflows zusammengeschaltet. Auf diese Art können komplexe Algorithmen in einzelne Module unterteilt werden, die sich jeweils um eine spezielle Aufgabe kümmern. Wiederkehrende Aufgaben lassen sich somit leicht wiederverwenden, was nicht nur Programmieraufwand, sondern auch Berechnungszeit einspart. *Nova* [Ols+11] ermöglicht die Definition von *Workflows* in Form von einem Graphen, der aus Pig-Programmen besteht. Es werden inkrementelle Aggregationen und Joins unterstützt, sodass sich Änderungen in den Basisdaten direkt durch den Workflow auf das Ergebnis anwenden lassen, ohne dass eine vollständige Berechnung erfolgen muss.

**MongoDB Aggregation Pipeline** Die Dokumentendatenbank MongoDB [Mona] bietet mit der Aggregation Pipeline die Möglichkeit, eine Abfolge von Schritten zu definieren, welche auf einer Dokumentensammlung zur Datenanalyse durchgeführt werden sollen. Die Pipeline gilt als Nachfolger der in MongoDB fest eingebauten MapReduce-Funktionalität, da sie deutlich einfacher als MapReduce zu benutzen, aber dennoch sehr mächtig ist und da die Berechnungen verteilt durchgeführt werden können. Anders als bei Pig wird die MongoDB Aggregation Pipeline lediglich auf einer einzigen Kollektion aufgerufen, mittels des `$lookup`-Schrittes ist es jedoch möglich, einen linken äußeren Verbund mit einer zweiten Kollektion durchzuführen. Schritte wie `$match` und `$project` werden auf jedem einzelnen Dokument in der Eingabe angewandt und dienen zur Selektion und Projektion. Schritte wie `$sort` und `$group` erfordern einen Datentransport über das Netzwerk, da alle Dokumente auf einmal betrachtet werden müssen. Einfache Graph-Analysen und Transformationen sind mit dem `$graphLookup`-Schritt möglich. Er funktioniert ähnlich dem `$lookup`-Schritt, jedoch werden Referenzen rekursiv traversiert. Beim zugrunde liegenden Datenmodell handelt es sich jedoch stets um JSON-Dokumente und es ist abgeschlossen. Das heißt, jeder Schritt nimmt eine Sequenz von gültigen JSON-Dokumenten entgegen und liefert auch wieder eine ebensolche. Die einzige Ausnahme ist der `$out`-Schritt, welcher das Transformationsergebnis in eine Ausgabe-Kollektion schreibt. Die Aggregation Pipeline ist gut geeignet für Dokumentendatenbanken, hat jedoch noch Schwächen im Umgang mit den dort auftretenden flexiblen Schemata. Es ist beispielsweise nicht möglich, Daten in Metadaten zu wandeln und umgekehrt.

## 3.4 FLWOR-Sprachen

*FLWOR* steht für `FOR`, `LET`, `WHERE`, `ORDER BY`, `RETURN` und ist vor allem aus der XML-Anfragesprache XQuery [Cha+03a] bekannt. In der `FOR`-Klausel kann über eine Sequenz von Eingabedaten iteriert werden. Eine verschachtelte Iteration ist über weitere `FOR`-Klauseln möglich. Nachdem mittels `LET` gegebenenfalls zusätzliche Variablen definiert wurden, die in der Filteroperation `WHERE` und in der `RETURN`-Klausel verwendet werden können, wird letztere Klausel für jeden qualifizierenden Datensatz angewendet, um eine Sequenz von Ausgabedatensätzen zu erzeugen. Mittels `ORDER BY` kann zuvor eine Sortierung vorgenommen werden.

**XQuery** Das von der XML-Anfragesprache XQuery [Cha+03a] verwendete Datenmodell ist das *XQuery Data Model* (*XDM*), welches einer Sequenz aus Knoten und Literalen entspricht. Typischerweise ist die Eingabe ein oder mehrere XML-Dokumente, welche baumartige Strukturen vorweisen. In [SBH13] wurde gezeigt, wie XQuery-Anfragen auf dem MapReduce-Framework ausgeführt werden können, um Daten, die in unterschiedlichen Systemen – nicht zwangsweise in XML-Dokumenten - gespeichert sind, verteilt zu verarbeiten.

**JSONiq** Die Sprache *JSONiq* [Rob+12] ermöglicht die Definition und Ausführung von Anfragen mit FLWOR-Ausdrücken auf JSON-Daten. Es gibt verschiedene Implementierungen, die Verbindungen zu Systemen wie MongoDB, Couchbase, aber auch zu relationalen Datenbanken ermöglichen. Für letztere werden Funktionen bereitgestellt, die eine SQL-Anfrage als Parameter nehmen und in der `FOR`-Klausel verwendet werden können. Ein dazu ähnlicher Ansatz ist SQL/XML [EM01]. Diese Sprache erlaubt es mittels spezieller Funktionen, XQuery-Anfragen auf XML-Daten innerhalb von SQL zu verwenden. JSONiq jedoch kommt meist nur zur Abfrage von Daten in Dokumentendatenbanken oder JSON-Dateien zum Einsatz. Genau wie XQuery unterstützt JSONiq Selektionen, Joins und das Aggregieren von Daten. Zur Erzeugung von Arrays mit Subdokumenten kommen verschachtelte FLWOR-Anfragen zum Einsatz. Analog zum Standard *XQuery Update* [Cha+08a] ermöglicht es auch JSONiq, Dokumente zu modifizieren.

## 3.5 Regelbasierte-Sprachen

Regelbasierte Sprachen kommen oft bei Datentransformationen zum Einsatz und dienen zur Definition von Transformationsregeln. Eine Regel legt fest, wie die Eingabe in die Ausgabe umgewandelt werden soll. Eine Regel kann zum Beispiel wie folgt definiert werden: *Wenn* in der Eingabe ein bestimmtes Muster gefunden wird, *dann* wird für jedes Auftreten des Musters ein Eintrag in der Ausgabe angelegt.

**XSLT** *XSL-Transformationen* [Cla+99] sind Teil der *Extensible Stylesheet Language* und dienen zur Transformation von XML-Dokumenten. Jede in XSLT formulierte Template-Regel beinhaltet ein Muster in Form eines XPath-Ausdrucks [Gro07] und einen Rumpf, welcher beschreibt, was die Ausgabe sein soll. In diesem Rumpf kann auf Unterelemente, Attribute und Textwerte zugegriffen werden, und ein rekursives Aufrufen der Template-Regeln angestoßen werden. Die Eingabe und Ausgabe einer XSL-Transformation sind in der Regel jeweils ein XML-Dokument. Die Sprache wird verwendet, um Dokumente von einem Schema in ein anderes zu überführen, aber auch um Datenwerte aus einem Dokument zu extrahieren oder zu aggregieren.

**Jolt** In XSLT steht eine Transformationsspezifikation selbst im XML-Format. Auch bei der Regel-basierten Transformationssprache für JSON-Dokumente *Jolt* [JSOb] ist die Spezifikation selbst ein JSON-Dokument. Anders als bei der MongoDB Aggregation Pipeline werden jedoch keine durchzuführenden Schritte definiert, sondern eine Liste von Transformationsregeln der Form `"a":{ "b":... }`. Beinhalt das Eingabedokument das Feld `a`, wird im Ausgabedokument das Feld `b` auf den gegebenen Wert gesetzt. Dabei dürfen Felder im Eingabedokument referenziert werden, allerdings gibt es keine Aggregatfunktionen. Mittels Wildcards (`*`, `@`, ...) können Transformationen auf Dokumenten mit heterogenen Schemata ausgeführt werden, sodass unabhängig von den Feldnamen die angegebenen Regeln ausgeführt werden.

## 3.6 Query-By-Example

Der Begriff *Query-By-Example* bedeutet, dass der Benutzer eine Anfrage dadurch formuliert, dass er ein Beispielergebnis angibt. Die Ergebnismenge einer Anfrage liefert also diejenigen Elemente, die zum angegebenen Beispiel passen. In Multimedia-Datenbanken wird diese Methode eingesetzt, um Bilder oder Tonaufnahmen zu finden. Der Benutzer malt eine Skizze oder summt eine Melodie ins Mikrofon und die Suchmaschine findet dazu passende Resultate. Die `find`-Methode der Dokumentendatenbank MongoDB [Mona] kann auch als Query-by-Example angesehen werden, um Selektionen auf Kollektionen durchzuführen. Der Benutzer übergibt der Methode ein Dokument, welches die Felder und Werte enthält, die in den zu findenden Dokumenten präsent sein sollen. Darin können auch Intervalle für Zahlenwerte sowie reguläre Ausdrücke für Zeichenkettenfelder verwendet werden.

**QBE** Die von IBM entwickelte Sprache *QBE* [RG00, Kapitel 6] – was für „Query by Example“ steht – basiert auf dem Domänenkalkül für das relationale Modell. Der Anwender sieht ein Tabellengerüst und füllt Felder, auf denen Selektionen ausgeführt werden sollen, mit Werten oder Bereichsprädikaten aus. Durch die Einführung von in mehreren Tabellen eingetragenen Variablennamen lassen sich damit auch Joins ausführen.

**Transformy** Das Datentransformationsprogramm *Transformy* [Tra] dient zur Vorverarbeitung von Textdateien. Als Eingabe dienen CSV-, JSON- oder anders strukturierte Dateien, deren Einträge homogen sind. Der Benutzer formuliert eine Transformation indem er angibt, wie das Resultat der Transformation für den ersten Eintrag der Datei auszusehen hat. Im ersten Schritt sucht der Transformy-Algorithmus die Korrespondenzen zwischen der gegebenen Ein- und Ausgabezeile. Im zweiten Schritt führt er eine äquivalente Transformation für alle Eingabeelemente durch. Komplexe Transformationen sind mit Transformy nicht möglich. Jede Transformation ist lediglich ein einfaches Mapping. Es können keine Zeilen gefiltert und keine Werte aggregiert werden.

## 3.7 Graph-Anfragesprachen

Die bisher vorgestellten Anfrage- und Transformationssprachen sind für die Verwendung auf relationalen Datenbanken, gesamtheitsorientierten NoSQL-Datenbanken sowie verschiedenen Dateiformaten ausgelegt. Um Beziehungen zwischen verschiedenen Datensätzen, wie sie beispielsweise in Graphdatenbanken vorkommen, zu unterstützen, werden spezielle Graph-Anfragesprachen verwendet.

**SPARQL** Das *Resoruce Description Framework* (*RDF*) [LS99] wird eingesetzt, um Ressourcen und Beziehungen zwischen diesen in Form von Subjekt-Prädikat-Objekt-Tripeln zu beschreiben. Die *SPARQL Protocol And RDF Query Language* [PS+08] hat Ähnlichkeiten zu SQL. Prädikate in der `WHERE`-Klausel werden jedoch über die erwähnten Tripel formuliert. Anders als bei SQL gibt es keine `FROM`-Klausel, da die Eingabe stets der vollständige vorliegende Graph ist. Mit Prädikaten wird ein Muster im Graphen gesucht, z. B. `?x abc:isCapitalOf ?y`. Auf die im Muster verwendeten Variablen kann in den anderen Klauseln zugegriffen werden, um beispielsweise die Ausgabe zu definieren.

**Cypher** Das Graphdatenbanksystem Neo4J [Neo] setzt als Abfragesprache *Cypher* ein. Genau wie SPARQL ist Cypher musterorientiert und hat Ähnlichkeiten zu SQL. Jedoch ist Cypher auf das Datenmodell von Property-Graphen angepasst. Ein in der `MATCH`-Kausel stehendes Muster kann Knoten und Kanten samt ihrer Labels und Properties beinhalten. Zum Beispiel findet das folgende Muster zwei benachbarte Knoten in der Graphdatenbank, wobei einer das Label Ort haben muss: `MATCH (x:Ort)--(y)`. Für jedes gefundene Auftreten wird ein `WHERE`-Prädikat überprüft sowie eine `ORDER BY`- und `RETURN`-Klausel angewandt, um Ergebnisse zu sortieren und sie schließlich auszugeben. Cypher kann auch dazu eingesetzt werden, um Knoten und Kanten in den Graphen einzufügen und um Änderungen durchzuführen.

**Gremlin** Die Idee hinter der Graph-Anfragesprache *Gremlin* [Rod15] ist es, sich schrittweise durch den Graphen zu navigieren. Eine Anfrage beginnt in der Regel mit `g.V()`, womit alle Knoten des Graphen gefunden werden. Im Anschluss daran können mittels `has`-Prädikaten gewisse Knoten aussortiert werden, die gewisse Kriterien – basierend auf den Knotenlabels oder Properties – nicht erfüllen. Über die Schritte `inE`, `outE`, `in` und `out` kann zu den ein- bzw. ausgehenden Kanten oder Nachbarknoten navigiert werden. Gremlin unterscheidet sich also insofern von Cypher, dass nicht der Graph nach Mustern durchsucht wird, sondern dass man den Graphen von Knoten zu Knoten traversiert. Eine Gremlin-Anfrage beschreibt anders als bei Cypher nicht *was* gefunden werden soll, sondern *wie* die Ergebnisse gefunden werden sollen [Neu13]. Die Möglichkeiten zum Erstellen und Bearbeiten von Knoten und Kanten sind zwar gegebenen, allerdings werden keine Batch-Updates oder vollständige Graph-Transformationen unterstützt.

**ArangoDB AQL** Das NoSQL-Datenbanksystem *ArangoDB* [Ara] ist eine Kombination aus verschiedenartigen NoSQL-Systemen. In einer von ArangoDB verwalteten Datenbank können sowohl Schlüssel-Wert-Paare als auch Dokumente und sogar Graphen gespeichert werden. Mit der *Arango Query Language* (*AQL*), die stark von SQL inspiriert wurde, können die Daten verschiedenartiger Datenmodelle abgefragt und modifiziert werden. Der gleiche Ansatz wurde von der *Unstructured Data Query Language* (*UnQL*) [Uns] verfolgt, welche letztendlich jedoch nur Anfragen ermöglichte, die in allen zugrunde liegenden Datenmodellen möglich sind, also beispielsweise ein Schlüssel-basierter Zugriff [Wei12]. AQL dagegen bietet für jedes Datenmodell spezielle Sprachkonstrukte, beispielsweise Joins für Dokumente oder eine Mustersuche und die Bestimmung kürzester Pfade in Graphen.

**Green-Marl** Um typische Graph-Analysen innerhalb weniger Zeilen Code zu formulieren, bietet die Sprache *Green-Marl* [Hon+12] spezielle Konstrukte. Mit diesen können sowohl skalare Eigenschaften für den gesamten Graphen – wie z. B. der Durchmesser – berechnet werden, aber auch Werte für jeden einzelnen Knoten, etwa der PageRank [Pag+99]. Auch kann Green-Marl zum Finden von Subgraphen eingesetzt werden, die gewisse Eigenschaften erfüllen. Die Sprache eignet sich jedoch nicht zur Graph-Konstruktion oder Transformation.

## 3.8 Weitere Sprachen

Es gibt noch viele weitere Sprachen, die nicht in die oben stehende Kategorisierung passen oder eine Spezialform aus einer oder die Kombination mehrerer Kategorien darstellen.

**Sawzall** Google entwickelte die Sprache *Sawzall* [Pik+05], um Datenanalysen auf Textdateien mit wenigen Zeilen Code formulieren zu können. Ein Sawzall-Skript wird als ein MapReduce-Job ausgeführt, welcher aus genau einer Map-und einer Reduce-Phase besteht. In der Map-Phase werden Filter und einfache Transformationen angewandt, in der Reduce-Phase werden Daten aggregiert. Es werden komplexe Typen wie Tupel, Listen und Maps unterstützt.

**CLeager** *CLeager* [SKS13] ist eine Sprache zur Definition von Schema-Evolutionen. Während die anderen bisher vorgestellten Sprachen primär zum Lesen oder Analysieren von Daten verwendet werden, modifiziert CLeager die Daten, oder besser gesagt, das Schema der Daten. Da anders als bei relationalen Datenbanken das Schema in einer NoSQL-Datenbank nicht global definiert wird, sondern in den Daten selbst zu finden ist, müssen Attributsumbenennungen und andere Migrationen auf jedem einzelnen Datensatz durchgeführt werden. In einem CLeager-Skript werden eine Menge von Operationen definiert. Diese können sein: `add`, `delete`, `rename` und einige mehr. Mit Hilfe eines solchen Skripts lässt sich eine Schema-Migration auf kompakte Weise definieren und eine vorliegende Datenmenge auf einen Schlag („eager“) in ein neues Schema überführen.

## 3.9 Vergleich

Tabelle 3.1 zeigt einen Vergleich der vorgestellten Sprachen bezüglich der Kriterien Schema-Flexibilität, systemübergreifende Anwendbarkeit, die Fähigkeit, Daten zu modifizieren sowie die Unterstützung für Graphen. Diese Kriterien sind abgeleitet von den Charakteristika der NoSQL-Datenbanken. Denn anders als bei relationale Datenbanken trifft man dort häufig auf flexible Schemata und heterogene Datenmodelle. Die Bewertung für die Kriterien erfolgen auf einer Skala von -- bis zu ++, wobei ○ für eine mittelmäßige Bewertung steht.
Es fällt auf, dass nur sehr wenige Sprachen für flexible Datenmodelle ausgelegt sind. Diese Unterstützung fehlt vor allem bei SQL-basierten Sprachen. Ansätze, die mit einem + oder ○ bewertet wurden, bieten zumindest

| | Schema-Flex. | Systemübergreifend | DML | Graphen |
|---|---|---|---|---|
| **Hive** | -- | ++ | ++ | -- |
| **Impala** | -- | + | ○ | -- |
| **Phoenix** | ○ | -- | ++ | -- |
| **MRQL** | ○ | - | ++ | - |
| **BigQuery** | - | - | -- | - |
| **N1QL** | - | -- | ++ | -- |
| **OQL** | - | -- | ++ | - |
| **FIRA** | ++ | -- | ++ | - |
| **SQL/MED** | -- | ++ | ++ | -- |
| **SQL++** | ○ | ++ | -- | -- |
| **CloudMdsQL** | - | ++ | -- | -- |
| **Toad** | -- | ++ | ++ | -- |
| **BQL** | ○ | ++ | -- | -- |
| **Pig/Nova** | - | ++ | ++ | - |
| **MongoDB AP** | -- | -- | ++ | ○ |
| **XQuery** | + | + | + | ○ |
| **JSONiq** | + | - | + | ○ |
| **XSLT** | ○ | ○ | + | - |
| **Jolt** | + | - | + | - |
| **QBE** | -- | -- | -- | -- |
| **Transformy** | -- | -- | + | -- |
| **SPARQL** | - | -- | ++ | ++ |
| **Cypher** | - | -- | ++ | ++ |
| **Gremlin** | - | -- | + | ++ |
| **AQL** | ○ | - | ++ | ++ |
| **Green-Marl** | -- | -- | -- | ++ |
| **Sawzall** | -- | - | + | -- |
| **CLeager** | -- | -- | ○ | -- |

**Tabelle 3.1:** Vergleich der NoSQL-Sprachen

die Möglichkeit, über eine Menge von Eingabeattributen zu iterieren, ohne deren Namen explizit angeben zu müssen. Einige der Systeme sind in der Lage, auf die Attributnamen – also auf Metadaten – zuzugreifen, sehr wenige ermöglichen die Wandlung von Daten in Metadaten. Bezüglich der systemübergreifenden Anwendbarkeit gibt es nur bei wenigen Ansätzen eine volle Unterstützung beliebiger Datenbanksysteme (++), andere sind nur für ein spezielles System (--) oder eine bestimmte Klasse von Systemen (-) anwendbar. Einige Ansätze ermöglichen lediglich lesenden Zugriff auf die Daten. In der Tabelle ist dies mit einem -- in der Spalte DML dargestellt. Neben der vollständigen DML-Unterstützung (++) ermöglichen manche Systeme lediglich einen Teil der notwendigen Operationen, beispielsweise nur (oder keine) In-Place-Updates (○/+). Es fällt auf, dass sehr viele NoSQL-Sprachen eine bestimmte Klasse von Datenbanksystemen nicht unterstützen, und zwar die Graphdatenbanken. Die speziell für diesen Zweck existierenden Graph-Anfragesprachen haben hingegen ihre Schwächen in den anderen untersuchten Kategorien.

# 4

# Die Transformationssprache NotaQL

Der Vergleich der im vorherigen Kapitel vorgestellten Sprachen für NoSQL-Datenbanken zeigt, dass es keine Sprache gibt, die für die vollständige NoSQL-Landschaft anwendbar ist. Während es beispielsweise Sprachen speziell für Graphdatenbanken gibt, bieten die übrigen Sprachen meist keine Unterstützung für ebendiese Klasse von Systemen. Das liegt zum einen daran, dass das Datenmodell sich stark von denen der anderen NoSQL-Datenbanken unterscheidet. Zum anderen trägt der Fakt, dass viele Sprachen auf SQL basieren, dazu bei, dass eine Erweiterung der Sprache für Graphtraversionen oft nicht möglich ist, ohne dass die einfache Benutzbarkeit der Sprache darunter leidet. Das starre Datenschema in relationalen Datenbanken ist auch der Grund wieso SQL-basierte sowie auch einige andere Sprachen keine Unterstützung für flexible Schemata bieten. Darüber hinaus sind viele Sprachen lediglich für ein spezielles Datenbanksystem oder eine bestimmte Klasse von Systemen konzipiert. Zwar sind Portierungen auf andere Datenbanken prinzipiell denkbar, jedoch nur, wenn die Datenmodelle kompatibel zueinander sind. Es existieren nur wenige Sprachen, die mehrere verschiedene Datenmodelle unterstützen und Abbildungen von einem Modell ins andere erlauben. Keine der vorgestellten Sprachen erfüllen alle Kriterien an eine universelle NoSQL-Sprache.

In diesem Kapitel stellen wir daher die Sprache *NotaQL* [SD15; SLD16; SKD17] vor. Dabei handelt es sich um eine Datentransformationssprache, die nicht für ein bestimmtes Datenbanksystem oder Datenmodell ausgelegt ist. Vielmehr soll sie für die komplette NoSQL-Welt anwendbar sein. Im ersten Abschnitt in diesem Kapitel werden die Ziele der Sprache präsentiert. Weiter unten erfolgt eine Definition und Erläuterung der Sprachkonstrukte.

## 4.1 Ziele der Sprache

Anhand der Kriterien an NoSQL-Sprachen aus dem vorherigen Kapitel leiten wir eine Liste von Zielen ab, welche von der Sprache NotaQL verfolgt werden, um als universelle Sprache für NoSQL-Datenbanken gelten zu können. Die folgenden Ziele werden im Rahmen dieser Arbeit bei der Konzeption der Sprache NotaQL verfolgt:

- *Schlichtheit:* Datentransformationen sollen innerhalb weniger Code-Zeilen formuliert werden können. Typische Transformationsaufgaben wie das Filtern, Umwandeln, Gruppieren und Aggregieren von Daten sollen ohne großen Aufwand definierbar sein.
- *Einfaches Ausführungsmodell:* Eine typische Transformation soll aus einer Quelle Daten lesen, diese transformieren und gegebenenfalls aggregieren, und schließlich in ein Ziel schreiben. Dieses Verhalten ähnelt dem einer MapReduce-Berechnung.
- *Datenmodellvielfalt:* NotaQL-Transformationen sollen für verschiedenartige Datenmodelle formuliert werden können. Der Anwender verwendet bei der Formulierung einer Transformationsvorschrift eine zum Ein- und Ausgabedatenmodell passende Syntax.
- *Datenmodellüberbrückend:* Die Datenmodelle der Datenquelle und des Datenziels müssen nicht identisch oder kompatibel sein.
- *Schemaflexibel:* Es soll auf eine einfache Art in NotaQL möglich sein, auf Metadaten wie zum Beispiel Spaltennamen zugreifen zu können. Transformationen sollen definierbar sein, ohne dass das genaue Datenschema bekannt ist. Auch die Umwandlung von Daten in Metadaten ist vonnöten.
- *Verteilte und inkrementelle Ausführung:* Die Sprache soll unabhängig von einer bestimmten Ausführungsart sein. Stattdessen soll die Ausführung auf gängigen Verarbeitungsframeworks verteilt erfolgen können; auch inkrementell.
- *Erweiterbarkeit:* Mittels benutzerdefinierter Funktionen, Konstruktoren und anderen Sprachkonstrukten soll NotaQL auf andere Systeme und Datenmodelle erweiterbar sein.

Bei der Betrachtung der Ziele fällt auf, dass es viele Parallelen zu den Definitionen von NoSQL-Datenbanken [Edl+10, S. 2] sowie Big Data [Lan01] gibt (siehe auch Kapitel 2.1 und 1.1):

| **NoSQL-Datenbanken** | **Big Data** | **NotaQL** |
|---|---|---|
| Nicht-relationales Datenm. | Variety | Datenmodellvielfalt |
| Schemaflexibel | Variety | Schemaflexibel |
| Verteilte Speicherung | Volume, Velocity | Verteilte Ausführung |

**Tabelle 4.1:** Korrespondenzen zwischen NoSQL-Datenbanken, Big Data und NotaQL

Im nächsten Abschnitt beginnt die Vorstellung der Sprache NotaQL. Damit ein Eindruck davon vermittelt werden kann, wie sich die Sprache verwenden lässt, ist das Kapitel ähnlich eines Tutorials aufgebaut. Am Ende des Kapitels werden die Sprachelemente bezogen auf die oben definierten Ziele zusammengefasst. Zunächst soll allerdings noch geklärt werden, was NotaQL nicht ist.

**NotaQL ist keine Anfragesprache.** Das sagt bereits der Name „Not a Query Language“. Die Sprache ist nicht dazu gedacht, innerhalb einer Anwendung oder einer Konsole auf Daten ad-hoc zuzugreifen. Stattdessen wird NotaQL dazu verwendet, Daten zu transformieren, entweder in Form von In-Place-Transformationen, um Daten am Ort der Speicherung zu modifizieren, oder in Form von ETL-artigen Transformationen, die ihre Ausgabe in ein separates Ziel schreiben.

**NotaQL ist keine Programmiersprache.** Programmiersprachen sind sehr mächtig und universell einsetzbar. Allerdings erfordern sie oft viel Einarbeitungsaufwand und sind für Anwender ohne Programmiererfahrung meist nicht nutzbar. Beim Programmieren entsteht oft ein langer Programmcode, selbst um simple Aufgaben auszuführen. Vieles davon sind generische Präambeln oder aufwändige Blöcke zur Behandlung von Fehlern. NotaQL ist durch seine Kompaktheit einfacher zu benutzen als eine Programmiersprache. Es entstehen bei der Definition von Datentransformation weniger Fehler und die Transformationsskripte sind besser verständlich.

**NotaQL ist kein Datenverarbeitungsframework.** Stattdessen kann NotaQL Datenverarbeitungsframeworks zur Ausführung nutzen, um von deren Vorteilen zu profitieren. Wie in Kapitel 2.2 beschrieben, kümmern sich die Frameworks um verteilte Berechnungen und um einen Wiederanlauf im Fehlerfall. Die Aufgabe des Anwenders ist es allerdings trotzdem noch, den Berechnungsalgorithmus zu programmieren. NotaQL soll dem Anwender diesen Aufwand abnehmen und stattdessen nur die Formulierung einfacher Transformationsskripte erfordern.

## 4.2 Attribut-Mappings

Die Grundidee von NotaQL ist es, eine Vorschrift zu definieren, wie ein Ausgabedatensatz basierend auf den Eingabedaten erstellt werden soll. Diese Vorschrift wird bei der Ausführung des Skriptes auf jeden Datensatz angewandt. Wir betrachten zunächst *Eins-zu-Eins-Abbildungen* und führen später Filter und Gruppierungen ein. Unter einer Eins-zu-Eins-Abbildung ist zu verstehen, dass jeder Eingabedatensatz genau einen Ausgabedatensatz erzeugt. Filter sind vom Typ *Eins-zu-maximal-Eins* und Gruppierungen vom Typ *Viele-zu-Eins*. Bei letzteren werden mehrere Eingabedatensätze zu einem Ausgabedatensatz kombiniert.

Das Herzstück eines jeden NotaQL-Skritps sind die sogenannten *Attribut-Mappings*. Jede Transformation besteht aus mindestens einem solchem Mapping:
Die einzelnen Attribut-Mappings sind Zuweisungen, die mit Kommata voneinander getrennt werden. Auf der linken Seite eines Zuweisungspfeils <–

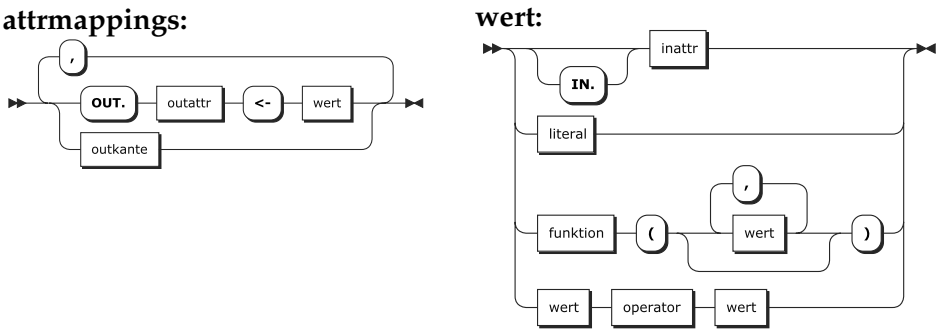



**Abbildung 4.1:** Attribut-Mappings

steht der Name eines Attributs im Ausgabedatensatz, auf der rechten Seite wird festgelegt, wie dessen Wert bestimmt wird. Dieser Wert kann über Funktionsaufrufe, Eingabeattribute oder Literale (also konstante Zahlen-, String- oder anders getypte Werte) definiert werden.[1] Im Laufe dieses Kapitels präsentieren wir die einzelnen Sprachkonstrukte von NotaQL in Form von Syntaxdiagrammen. Zur besseren Lesbarkeit lassen wir einige Definitionen von trivialen Nichtterminalsymbolen (z. B. `literal` oder `operator`) weg. Diese befinden sich jedoch im vollständigen Syntaxdiagramm in Anhang A. Dort ist auch eine Darstellung in erweiterter Backus-Naur-Form (EBNF) zu finden.

Wird im System, in welches die Ausgabe einer Transformation geschrieben werden soll, das Konzept einer ID verwendet – und das ist bei den meisten Datenbanksystemen der Fall –, sollte über das erste Attribut-Mapping innerhalb einer Transformation der ID-Wert eines Ausgabedatensatzes festgelegt werden. Wird dies weggelassen, wird die ID automatisch von System generiert. Weitere Attribut-Mappings definieren die zusätzlichen Attribute der Ausgabedatensätze.

Das folgende Beispiel zeigt eine MongoDB-Kollektion, welche wir als Eingabe für einige in dieser Arbeit präsentierten NotaQL-Transformationsskripte verwenden.

```
{ _id: 1, name: "Anita", gehalt: 50000, bonus: 2000 }
{ _id: 2, name: "Bruno", gehalt: 50000, bonus: 3000 }
{ _id: 3, name: "Clara", gehalt: 60000, bonus: 500 }
```

Das nun folgende erste NotaQL-Beispiel besteht aus zwei Attribut-Mappings. Es erzeugt für jedes Dokument der Eingabekollektion ein Dokument, welches die Felder `_id` und `person` besitzt. Da die Felder `gehalt` und `bonus` in der Transformation nicht auftauchen, sind sie in der Ausgabe nicht präsent.

```
OUT._id <- IN._id,
OUT.person <- IN.name
```

```
{ _id: 1, person: "Anita" }
{ _id: 2, person: "Bruno" }
{ _id: 3, person: "Clara" }
```

[1] Auf das im Diagramm referenzierte Symbol `outkante` wird in Kapitel 4.5 eingegangen.

Auf der linken Seite des Pfeils steht der Name eines Attributs, welches im Ausgabedatensatz enthalten sein soll. Auf der rechten Seite darf auf Felder der Eingabe verwiesen werden, und es dürfen Literale, Funktionen und Berechnungen vorkommen:

```
OUT._id <- IN._id,
OUT.vorname <- uppercase(IN.name),
OUT.nachname <- 'Unbekannt',
OUT.gehalt <- (IN.gehalt + IN.bonus)*1.05,
OUT.bonus <- 0
```

```
{ _id: 1, vorname: "ANITA", nachname: "Unbekannt",
  gehalt: 54600, bonus: 0 }
{ _id: 2, vorname: "BRUNO", nachname: "Unbekannt",
  gehalt: 55650, bonus: 0 }
{ _id: 3, vorname: "CLARA", nachname: "Unbekannt",
  gehalt: 63525, bonus: 0 }
```

Das `_id`-Feld dient in MongoDB zur Identifizierung eines Dokuments und muss innerhalb der Ziel-Kollektion eindeutig sein. In den beiden präsentierten NotaQL-Skripten ist dies immer der Fall, da die `_id`-Werte aus der Eingabe-Kollektion stammen und dort eindeutig sind.

NotaQL verwendet für den Zugriff und für die Erzeugung von Unterdokumenten bzw. Sub-Attributen die Dot-Notation. Mit Punkt-Symbolen voneinander getrennte Attributnamen dienen dabei der Definition der Verschachtelung bis hin zu einer beliebigen Tiefe. Ein Beispiel:

```
OUT._id <- IN.name,
OUT.job.gehalt <- IN.gehalt
```

```
{ _id: "Anita", job: { gehalt: 50000 } }
{ _id: "Bruno", job: { gehalt: 50000 } }
{ _id: "Clara", job: { gehalt: 60000 } }
```

Beinhaltet ein Eingabedokument solche Unterdokumente, kann analog mittels `IN.job.gehalt` auf die Sub-Attribute zugriffen werden. Es ist zu beachten, dass das soeben vorgestellte Skript nur dann erfolgreich ausgeführt werden kann, wenn es keine zwei Personen mit dem gleichen Namen gibt. Andernfalls würden mehrere Ausgabedokumente mit der gleichen ID erzeugt werden und es träten Kollisionen bei den übrigen Attributwerten auf. Pro Ausgabe-ID-Wert wird in NotaQL jedoch immer nur ein einziger Datensatz erzeugt. Im gegebenen Falle würde diese Transformation wegen der Wertekollisionen fehlschlagen. In Abschnitt 4.4 wird erläutert, wie sich dieses Problem mit Hilfe von Aggregatfunktionen lösen lässt.

### Arrays

Im oben stehenden Syntaxdiagramm (Abbildung 4.1) wird zwischen den Symbolen `outattr` und `inattr` unterschieden (siehe auch Abbildung 4.3). Dies hat den Grund, dass sich die Methoden für den lesenden Attributszugriff von denen eines schreibenden unterscheiden können. Dies ist zum Beispiel bei listenwertigen Attributen und Arrays der Fall. Während das Lesen einzelner Werte über die Angabe einer Position erfolgt, werden Arrays mittels einer Aggregatfunktion `LIST` konstruiert. Das Syntaxdiagramm in Abbildung 4.2 zeigt die drei verschiedenen Zugriffsarten auf Arrays und listenwertige Attribute. Der Zugriff auf ein bestimmtes Element ist über die Angabe der Position möglich, das erste Element hat dabei die Position `[0]`. Mit dem Stern-Operator `[*]` wird ein Array „entschachtelt“. Die aus anderen Sprachen unter dem Namen *Unnest* bekannte Operation iteriert über die einzelnen Array-Elemente und behandelt das Eingabedokument für jedes einzelne Element separat. Innerhalb einer Iteration kann in NotaQL mittels `[@]` auf das aktuelle Arrayelement zugegriffen werden.

**inarray:**

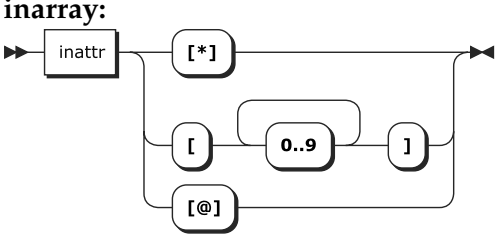



**Abbildung 4.2:** Zugriff auf Arrays in der Eingabe

Der `position()`-Schritt ermöglicht nach einem Arrayzugriff die Ermittlung der Position eines Elements. Das folgende Beispiel beinhaltet verschiedene Zugriffe auf Arrays.

```
{ _id: 1, vornamen: ["Anita", "Rebecca"],
  hobbys: ["Tennis", "Yoga"] }
```

```
OUT._id <- IN.hobbys[*],
OUT.person <- IN.vornamen[0],
OUT.position <- IN.hobbies[@].position()
```

Nach dem Entschachteln des Hobbys-Arrays wird für jedes darin enthaltene Element ein Ausgabedokument erzeugt. Die erzeugten Dokumente haben als `_id` das Hobby, als Person den ersten im Vornamen-Array vorkommenden Wert und als Position den Index im ursprünglichen Hobbys-Array. Da mit dem Stern `[*]` über die Elemente des Array iteriert wird, kann mit `[@]` auf das Element der aktuellen Iteration zugegriffen werden. Die Ausgabe sieht wie folgt aus:

```
{ _id: "Tennis", person: "Anita", position: 0 }
{ _id: "Yoga", person: "Anita", position: 1 }
```

Auch hier ist zu beachten, dass kein Hobby in einem oder in unterschiedlichen Dokumenten mehrfach vorkommen darf, da ansonsten kein eindeutiges Ausgabedokument für die gegebene `_id` mehr erzeugt werden kann. Mehr zu Arrays, insbesondere zur Erzeugung solcher, folgt in Kapitel 4.4.

### Indirektion

Sowohl beim Zugriff auf Eingabeattribute als auch bei der Definiton von Ausgabeattributen kann der *Indirektionsoperator*[2] `$` zum Einsatz kommen [Ind]. Dieser macht es möglich, den Namen eines Attributs nicht fest bestimmen zu müssen, sondern beispielsweise aus Namen oder Werten von Eingabeattributen zu ermitteln. Die Indirektion bietet viele Möglichkeiten beim Umgang mit flexiblen Schemata.

Gegeben sei eine Menge von Eingabedatensätzen, die das Attribut `a` sowie weitere unbekannte Attribute besitzen. `IN.$(IN.a)` liefert den Wert des Attributs, welches den Namen trägt, der sich im Attribut `a` findet. Diese Operation wird in FIRA [WR05] *Attribut-Dereferenz* $\Delta$ genannt. Nehmen wir an, eine Person erhält ihr Gehalt beispielsweise aufgrund von ausländischen Einkünften in verschiedenen Währungen und es soll die folgende Transformation durchgeführt werden, um die gegebene Eingabe in eine Ausgabe zu überführen, in welcher nur ein Gehaltsattribut mit dem Gehalt in der Standard-Währung präsent ist:

```
{ _id: 1, vorname: "Anita", eur: 50000, usd: 2000,
  chf: 7500, standard_waehrung: "eur" }
```

$$\pi_{\_id,vorname,gehalt,standard\_waehrung} \Delta^{gehalt}_{standard\_waehrung}(Person)$$

```
{ _id: 1, vorname: "Anita", gehalt: 50000,
  standard_waehrung: "eur" }
```

Ein äquivalentes NotaQL-Skript sieht wie folgt aus:

```
OUT._id <- IN._id, OUT.vorname <- IN.vorname,
OUT.gehalt <- IN.$(IN.standard_waehrung),
OUT.standard_waehrung <- IN.standard_waehrung
```

Der Indirektionsoperator kann nicht nur auf der Eingabe- sondern auch auf der Ausgabeseite verwendet werden. Dies ermöglicht das Erzeugen von Attributen, deren Namen beispielsweise aus Eingabeattributen entnommen werden. Mittels `OUT.$(IN.a)` kann ein Ausgabeattribut erzeugt werden, welches den Namen trägt, der sich im Eingabeattribut `a` findet.

Analog zu den `[*]`- und `[@]`-Operatoren bei Arrays kann auch bei ganzen Datensätzen über die einzelnen Attribute oder deren Sub-Attribute iteriert werden. Dazu kommen die Pfadschritte `IN.*` bzw. `IN.@` zum Einsatz. Zusätzlich zum Stern-Operator ist es in NotaQL mit dem `?`-Operator möglich, auf alle Attribute zuzugreifen, die ein gegebenes Prädikat erfüllen. Dieser Operator wird auch *Attributfilter* genannt. Innerhalb des Prädikats können logische Operatoren sowie Vergleiche zum Einsatz kommen.

Der `name()`-Schritt kann bei der Iteration über Datensatzattribute eingesetzt werden, um den Namen eines Attributs abzufragen. Er dient also zur Überführung von Metadaten in Daten. Der Indirektionsoperator hingegen wird zur Wandlung von Daten in Metadaten eingesetzt. Verwendet man die beiden Sprachkonstrukte gemeinsam, können Datensätze mit allen ihren Attributen kopiert werden, ohne die Namen der einzelnen Attribute zu kennen. Das folgende Beispiel kopiert also eine vollständige Kollektion:

[2] Eine weitere gebräuchliche Bezeichnung ist *Dereferenzierungsoperator*.

```
OUT._id <- IN._id,
OUT.$(IN.*.name()) <- IN.@
```

Neben der oben präsentierten Attribut-Dereferenz können weitere Operationen aus der Federated Interoperable Relational Algebra (FIRA) [WR05] mittels NotaQL formuliert werden. Einer davon ist der *Transpositions*-Operator $\tau_b^a$ . Dieser kreiert ein neues Attribut, welches den Namen trägt, der im Attribut $a$ zu finden ist, und den Wert aus Attribut $b$ hat. Mit dem Transpositions-Operator kann man die oben präsentierte Standardwährungs-Transformation in die umgekehrte Richtung definieren. Zu beachten ist allerdings, dass bei der ursprünglichen Transformation alle Gehaltswerte, die nicht in der Standard-Währung stehen, verloren gegangen sind. Hier zu sehen ist ein Eingabedatensatz, der dazugehörige FIRA-Ausdruck sowie das Ergebnis:

```
{ _id: 1, vorname: "Anita", gehalt: 50000,
  standard_waehrung: "eur" }
```

$$\text{ш}_{gehalt}\tau_{gehalt}^{standard\_waehrung}(Person)$$

```
{ _id: 1, vorname: "Anita", eur: 50000,
  standard_waehrung: "eur" }
```

Die im FIRA-Ausdruck verwendete *Drop-Projektion* ш ist nötig, um das Gehalts-Attribut wieder zu entfernen. Ansonsten wäre es im dargestellten Beispielergebnis doppelt vorhanden, einmal unter dem Namen `gehalt` und einmal `eur`. Mit einer normalen Projektion ist dies nicht möglich, da die Namen der beizubehaltenden Attribute – in diesem Fall speziell das Attribut `eur` – nicht vollständig bekannt sind. In der folgenden NotaQL-Transformation wird die oben stehende Transposition mit Hilfe des Indirektionsoperators durchgeführt. Eine zweite Indirektion wird verwendet, um alle Attribute mit Ausnahme des Gehalts zu kopieren.

```
OUT._id <- IN._id,
OUT.$(IN.standard_waehrung) <- IN.gehalt,
OUT.$(IN.?(IN.name()!='gehalt').name()) <- IN.@
```

Das im Beispiel verwendete Prädikat `?(IN.name()!='gehalt')` ist für alle Attribute wahr, die einen von `gehalt` verschiedenen Namen haben. Mit der folgenden Zeile würden alle Attribute übernommen; das Gehalt allerdings nur, wenn es größer als 60000 ist:

```
OUT.$(IN.?(IN.name()!='gehalt'||IN.@>60000).name())
                                          <- IN.@
```

*Attribut-Mappings* werden in NotaQL zur Definition, wie Ausgabedatensätze basierend auf der Eingabe kreiert werden sollen, eingesetzt. Die Operatoren `*` und `?` eignen sich für Transformationen auf flexiblen Schemata, also wenn Attributnamen nicht vollständig bekannt sind. Beim `*`-Operator wird über alle Attribute iteriert, beim `?` über diejenigen, die ein gegebenes Prädikat erfüllen. Sollen in der Ausgabe Attribute präsent sein, deren Namen nicht direkt angegeben werden können, kommt der Indirektionsoperator `$` zum Einsatz. Ein dem Operator übergebener Ausdruck legt dabei den Namen des Ausgabeattributs fest.

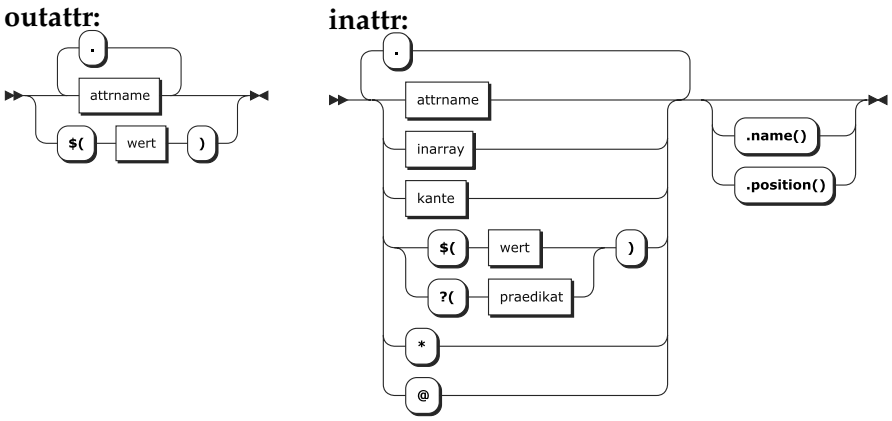



**Abbildung 4.3:** Eingabe- und Ausgabeattribute

## 4.3 Selektionen

Bei einer Selektion handelt es sich um einen über ein logisches Prädikat definierten Filter, welcher auf einer Basisdatenmenge angewendet wird. Nur diejenigen Datensätze, die das Prädikat erfüllen, werden weitergeleitet, alle anderen werden aussortiert. In NotaQL kann zur Selektion der Eingabefilter **`IN-FILTER`** verwendet werden. Mit diesem werden die in der Transformationvorschrift definierten Attribut-Mappings nur auf dem Teil der Basisdaten ausgeführt, der das Filterkriterium erfüllt. Die **`IN-FILTER`**-Klausel wird im NotaQL-Skript vor den Attribut-Mappings platziert. Die Syntax ist in Abbildung 4.4 dargestellt. Das Syntaxdiagramm zeigt, dass Prädikate geklammert, negiert und mit logischen Operatoren verknüpft werden können. Innerhalb der einzelnen Teilprädikate können Vergleiche und Attributsexistenzprüfungen vorgenommen werden.

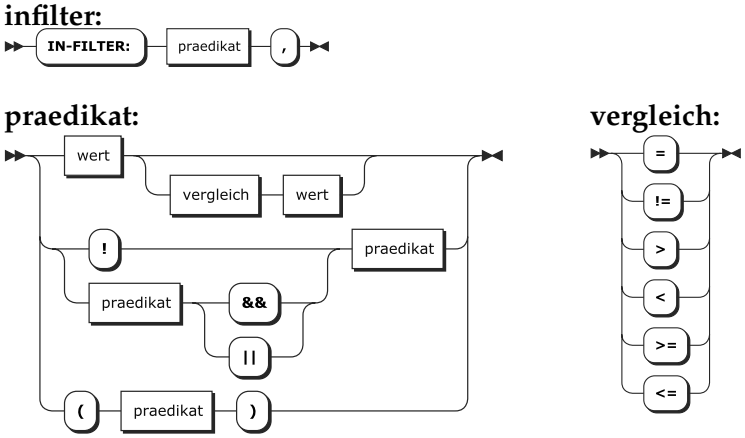



**Abbildung 4.4:** Eingabefilter

Die gängigste Art der Selektion ist diejenige, die in jedem Datensatz überprüft, ob ein Attributwert ein gewisses Kriterium erfüllt. Das folgende NotaQL-Skript kopiert alle Personendatensätze, bei denen ein Gehalt von über 55000 angegeben wurde:

```
IN-FILTER: IN.gehalt > 55000,
OUT._id <- IN._id,
OUT.$(IN.name()) <- IN.@
```

Wenn ein Datensatz das Gehaltsattribut nicht besitzt, wird das Vergleichsprädikat als falsch ausgewertet und der Datensatz somit ausgefiltert. Mit dem Filter `IN-FILTER: IN.gehalt` kommen diejenigen Datensätze in Betracht, bei denen das Gehaltsattribut präsent ist, er überprüft also auf die Existenz eines Attributs.

Im Eingabefilter können genau wie bei Attribut-Mappings eingebaute oder benutzerdefinierte Funktionen aufgerufen werden. Somit findet beispielsweise `uppercase(name) = 'UTE'` alle Ute heißenden Personen, unabhängig von der Groß- und Kleinschreibung. Das folgende Prädikat wird von allen Datensätzen, die mehr als fünf Attribute besitzen, erfüllt: `COUNT(IN.*) > 5`. Zu beachten ist, dass `COUNT` eine Aggregatfunktion ist, die üblicherweise in Verbindung mit Gruppierungen zum Einsatz kommt. In NotaQL ist es aber auch möglich, diesen direkt Listenwerte zu übergeben. Mehr zu Aggregatfunktionen folgt im nun folgenden Abschnitt.

## 4.4 Aggregationen

In SQL kommen Aggregatfunktionen zum Einsatz, um Werte, die bei einer Gruppenbildung innerhalb ein und derselben Gruppe gesammelt werden, zu einem atomaren Wert zusammenzufassen. Ohne die Verwendung einer **GROUP BY**-Klausel landen alle Werte zusammen in einer Gruppe und es wird ein einziges Gesamtresultat berechnet. In NotaQL erfolgt die Gruppierung implizit anhand der Bildung von *Zellen*. Unter einer Zelle versteht man ein Attribut innerhalb eines bestimmten Datensatzes. Der Datensatz wird über seinen ID-Wert identifiziert, das Attribut über seinen Namen. Zerlegt man die Eingabedatensätze einer NotaQL-Transformation jeweils in ihre Zellen, kann eine NotaQL-Transformation als Umwandlung auf Zell-Ebene angesehen werden. Diejenigen Ausgabezeilen, die den gleichen ID-Wert erhalten, bilden einen Ausgabedatensatz. Wie bereits weiter oben erwähnt, schlägt eine Transformation fehl, wenn es innerhalb ein und derselben Ausgabezelle eine Mehrdeutigkeit für einen Attributwert gibt. Anders ausgedrückt: Eine Transformation ist ungültig, wenn ein und derselben Ausgabezelle zwei oder mehr Werte zugewiesen werden.

Betrachten wir die folgende Personenkollektion, bei der jede Person eine Firma und ein Gehalt besitzt.

```
{ _id: 1, name: "Anita", firma: "IBM", gehalt: 50000 }
{ _id: 2, name: "Bruno", firma: "SAP", gehalt: 50000 }
{ _id: 3, name: "Clara", firma: "IBM", gehalt: 60000 }
```

Die folgende NotaQL-Transformation schlägt fehl, da der Wert der Ausgabezelle `(IBM, gehalt)` nicht eindeutig bestimmt werden kann.

```
OUT._id <- IN.firma,
OUT.gehalt <- IN.gehalt                    #schlägt fehl
```

Die aus neun Zellen (drei Zeilen mit je drei Attributen; die `_id` nicht mitgezählt) bestehende Eingabe produziert als Ausgabe die drei Zellen `(IBM,`

gehalt), (SAP, gehalt) und nochmals (IBM, gehalt). Jeweils mit den Werten 50000, 50000 und 60000. Beim Zusammenbau des Ausgabedatensatzes mit der ID *"SAP"* tritt kein Fehler auf; der Wert des Gehaltsattributs lautet 50000. Bei der Zeile mit der ID *"IBM"* liegen jedoch zwei Werte vor, sodass der Wert des Gehaltsattributs nicht bestimmt werden kann. Für solche Fälle sind Aggregatfunktionen zu benutzen. Diese legen fest, wie aus einer Menge von Werten ein atomarer Endwert berechnet wird. Der Endwert kann eine Zahl, eine Zeichenkette, aber auch eine Liste oder ein Objekt eines beliebigen anderen Typs sein. Mit der Atomarität ist hier nur gemeint, dass es lediglich einen einzigen solchen Wert hat. Für das vorliegende Beispiel können die aus SQL bekannten Aggregatfunktionen SUM, MIN, MAX, AVG oder COUNT verwendet werden, um die Gehaltsumme oder den größten bzw. kleinsten Wert pro Firma, das Durchschnittsgehalt oder lediglich die Anzahl der Personen je Firma zu berechnen:

```
OUT._id <- IN.firma,
OUT.gehaltssumme <- SUM(IN.gehalt),
OUT.mingehalt <- MIN(IN.gehalt),
OUT.maxgehalt <- MAX(IN.gehalt),
OUT.durchschnittsgehalt <- AVG(IN.gehalt),
OUT.anzahlpersonen <- COUNT()
```

COUNT() zählt die Anzahl der Zellen, die für eine bestimmte Ausgabezelle aggregiert werden sollen. Äquivalente Ausdrücke dazu sind SUM(1) oder COUNT(_id), wenn die _id einen Datensatz eindeutig identifiziert und sie in jedem Eingabedatensatz präsent ist. Analog zur Zählfunktion in SQL können mit COUNT(IN.a) die Datensätze gezählt werden, in denen das Attribut a präsent ist. Um von der in Abbildung 4.1 gezeigten Funktionsaufrufsyntax nicht abzuweichen, kann man die Anzahl der unterschiedlichen a-Werte mit der Aggregatfunktion COUNT_DISTINCT(IN.a) bestimmen – analog zum SQL-Ausdruck COUNT(DISTINCT a).

Die Summenfunktion funktioniert nur auf nummerischen Attributen. Da in NoSQL-Datenbanken keine vertikale Homogenität herrscht, kann es vorkommen, dass es in verschiedenen Zeilen innerhalb ein und demselben Attribut Werte von verschieden Typen gibt. Um in solchen Fällen Transformationen trotzdem erfolgreich durchführen zu können, interpretiert NotaQL String-Werte als die Zahl 0. Sie beeinflussen damit also nicht die Summe. Da jedoch die Funktion AVG(IN.a) zur Durchschnittsberechnung definiert ist als SUM(IN.a)/COUNT(IN.a), beeinflussen String-Werte sehr wohl den Durchschnitt. Ein im Eingabedatensatz fehlendes Attribut hat auf keinerlei Aggregatfunktion einen Einfluss. Es wird nicht als 0 interpretiert, sondern übersprungen.

Die Funktionen MIN und MAX setzen im Gegensatz zu den oben beschriebenen numerischen Funktionen lediglich eine totale Ordnung auf dem zugrunde liegenden Typ voraus. Sie sind also auch auf Zeichenketten anwendbar, da sie sich lexikographisch ordnen lassen. Das Minimum oder Maximum aus zwei Werten unterschiedlichen Typs ist nicht klar definiert. Einige Implementierungen (z. B. SQLite [Owe03]) interpretieren in solchen Fällen Zahlen als Zeichenketten. Somit ist beispielsweise $5 < \text{"}6\text{"} < 7$. Andere Implementierungen haben eine interne Ordnung auf verschiedenen Typen. In MongoDB [Mona] sind beispielsweise Zahlen stets kleiner als

Strings. Eine dritte Alternative wäre ein Fehlschlagen der Transformation, wenn unterschiedliche Typen als Eingabe in die Aggregatfunktion dienen. In der aktuellen Implementierung von NotaQL wurde die Interpretation von Zahlen als Strings wie in SQLite verfolgt. Mittels benutzerdefinierter Aggregatfunktion lassen sich aber auch die anderen Varianten implementieren.

Mit Ausnahme der `COUNT`-Funktion, welche ohne Parameter auskommt und lediglich die Kardinalität einer Liste berechnet, ohne die einzelnen Elemente zu betrachten, haben die meisten gängigen Aggregatfunktionen genau einen Parameter. In NotaQL können über weitere Parameter zusätzliche Konfigurationen vorgenommen werden. Ein Beispiel ist die Aggregatfunktion `IMPLODE`, die zur Konkatenation von Zeichenketten dient. Während die Zeichenkettenfragmente selbst im ersten Parameter übergeben werden, wird über einen zweiten Parameter das Begrenzungszeichen gesetzt. Das folgende Beispiel liefert zu jeder Firma eine Komma-getrennte Zeichenkette mit den Personen, die in dieser Firma arbeiten:

```
OUT._id <- IN.firma,
OUT.mitarbeiter <- IMPLODE(IN.name, ",")
```

```
{ _id: "IBM", mitarbeiter: "Anita,Clara" }
{ _id: "SAP", mitarbeiter: "Bruno" }
```

Zur Konstruktion von Listen wird ebenfalls eine Aggregatfunktion eingesetzt. Das Resultat der Funktion `LIST` ist ein Array, der die zu aggregierenden Elemente enthält. Das soeben präsentierte Beispiel lässt sich wie folgt ändern, um einen Mitarbeiter-Array pro Firma zu erzeugen:

```
OUT._id <- IN.firma,
OUT.mitarbeiter <- LIST(IN.name)
```

```
{ _id: "IBM", mitarbeiter: ["Anita","Clara"] }
{ _id: "SAP", mitarbeiter: ["Bruno"] }
```

Die Ausgabedaten, die durch die beiden vorgestellten Transformationen erzeugt wurden, kann man als eine Art Index sehen. Würde man basierend auf den Ursprungsdaten beispielsweise die Frage beantworten wollen, welche Mitarbeiter bei IBM arbeiten, müsste in einer NoSQL-Datenbank, welche keine sekundären Indexe bietet, der komplette Datenbestand durchsucht werden. Daher ist es in NoSQL-Datenbanken üblich, dass sich die Anwendung selbst um das Pflegen der Indexe kümmert [Fox16]. Für den Aufbau und das Warten dieser Indexe eignet sich NotaQL.

Die vorgestellten Aggregatfunktionen kombinieren Werte, welche aus verschiedenen Eingabedatensätzen stammen zu einem atomaren Ausgabewert. Innerhalb einer NotaQL-Vorschrift erhalten sie als Eingabe einen Wert des Typs $T$, in der Implementierung jedoch sind sie Funktionen vom Typ $List\langle T\rangle \rightarrow T'$. Die Liste enthält alle nach dem impliziten Gruppieren in einer Zelle gesammelten Werte. Der Ausgabetyp der Aggregatfunktion kann ein anderer sein als der Eingabetyp, was zum Beispiel bei der Funktion `LIST` der Fall ist. Mehr zu benutzerdefinierten Funktionen folgt in Abschnitt 4.7.

Wie bereits weiter oben erwähnt, können Aggregatfunktionen auch direkt listenwertige Attribute übergeben werden. Dies ist sinnvoll, wenn bereits auf der Eingabeseite eine Liste von Werten vorliegt, nicht erst nach dem Gruppieren. Beim Aufruf von `COUNT(IN.hobbys[*])` wird die Anzahl der Elemente in einem Hobbys-Array und mit `COUNT(IN.*)` die Anzahl der Attribute eines Datensatzes gezählt. Diese Berechnungen können direkt auf einem vorliegenden Eingabedatensatz ausgeführt werden. Dies ermöglicht auch sogenannte *horizontale Aggregationen* [LSS96]. Sie berechnen für jeden einzelnen Datensatz Aggregate über eine Menge von Attributen. In folgendem Beispiel sind Gehälter in einzelne Bestandteile aufgeteilt und es soll das Gesamtgehalt jeder Person berechnet werden:

```
{ _id: 1, name: "Anita",
  gehalt: { grundgehalt: 50000, bonus: 2000 } }
```

```
OUT._id <- IN._id, OUT.name <- IN.name,
OUT.gehalt <- SUM(IN.gehalt.*)
```

Die gezeigte Transformation summiert alle Unterattributswerte des Gehalts-Attributs auf, ohne dass deren Namen bekannt sein müssen. In anderen Datensätzen könnten andere Attribute, z. B. `weihnachtsgeld`, vorkommen, dennoch gehen sie mit in die Summe ein.

## 4.5 Graph-Transformationen

Graphen gewinnen in Informationssystemen deutlich an Wichtigkeit, vor allem, wenn es darum geht, verschiedene Datensätze miteinander in Beziehung zu setzen. Typische Anwendungen sind soziale Netzwerke oder Empfehlungssysteme in Webshops. Ein Graph im mathematischen Sinne ist definiert über eine Menge an Knoten sowie eine Menge an Kanten:

$$G = (V, E)$$

In einem vereinfachten Modell geben wir den Knoten lediglich einen Namen: $V = \{v_1, v_2, \dots\}$. Eine Kante verbindet jeweils einen Startknoten mit einem Zielknoten, sie hat also eine Richtung: $E \subseteq V \times V$. Ungerichtete Kanten betrachten wir im Rahmen dieser Arbeit nicht, da sie in Graph-basierten Datenbanken nur selten zum Einsatz kommen.

Ein weit verbreitetes Datenmodell, welches auf Graphen aufbaut, sind die *Property-Graphen*. In diesem Modell besitzen Knoten und Kanten sogenannte *Properties*, das sind Attribut-Wert-Paare. Des Weiteren hat jede Kante genau ein *Label*, also eine Beschriftung. Dieses Label gibt an, welche Art von Beziehung durch eine gegebene Kante dargestellt wird, z. B. eine Freundschaftsbeziehung zwischen zwei Personen. Knoten können ebenfalls Labels haben, welches den Typ des Knoten angibt, beispielsweise Person, Student und Hiwi, oder aber Produkt, Bewertung oder Blogbeitrag. Abbildung 4.5 zeigt einen Graphen mit zwei Personen-Knoten, die mit einer Kante verbunden sind. Die Kantenrichtung gibt an, wer initial die Freundschaftsanfrage gestellt hat, sie hat auf die Freundschaftsbeziehung aufgrund ihrer Symmetrie jedoch keine Relevanz.

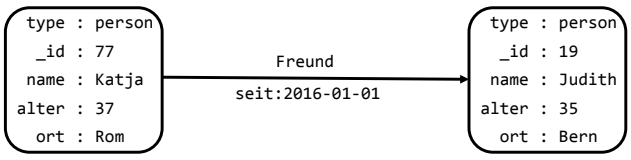



**Abbildung 4.5:** Ein Freundschafts-Graph mit zwei Knoten

In diesem Abschnitt betrachten wir vier gebräuchliche Schema-Varianten, die zur Modellierung von Graphen in relationalen sowie NoSQL-Datenbanken zum Einsatz kommen. Zu jeder Modellierung werden NotaQL-Skripte präsentiert, um Datentransformationen auf und zwischen den jeweiligen Schemata ausführen zu können. Die ersten drei Varianten basieren auf Datenmodellkonzepten, die bereits in den oben stehenden Transformationen genutzt wurden, beispielsweise Tabellenspalten, Subdokumente oder Arrays. Die letzte Variante basiert auf dem Datenmodell von Graphdatenbanken, in welchem Knoten und Kanten als native Konzepte vorkommen. Zur Traversierung über Kanten sowie zur Erstellung solcher werden spezielle NotaQL-Syntaxerweiterungen vorgestellt.

### Knotentabelle und Kantentabelle

Das Schema, welches wir als erstes betrachten, wird häufig zur Modellierung von Graphstrukturen in relationalen Datenbanken eingesetzt. Da diese ein festes Schema besitzen, eignet sich diese Variante gut, wenn die Properties von Knoten und Kanten von vornherein bekannt sind. Das Schema besteht im Wesentlichen aus zwei Tabellen. Die *Knotentabelle* hat eine Spalte für eine Knoten-ID, sowie jeweils eine Spalte für jede Property. Die *Kantentabelle* besitzt jeweils eine Fremdschlüsselspalte mit der Knoten-ID des Start- und Zielknotens, sowie eine Label-Spalte und ebenfalls eine für jede Property. Während der Primärschlüssel der ersten Tabelle über die Knoten-ID definiert wird, muss bei der Kantentabelle der Schlüssel zusammengesetzt werden aus Start- und Zielknoten sowie dem Kantenlabel. Wenn in der Modellierung mehrere Kanten mit gleichem Label zwischen identischen Knoten erlaubt sein sollen, muss eine Kanten-ID eingeführt werden, welche als Primärschlüssel dient. Wir verwenden jedoch das vereinfachte Schema ohne Kanten-ID. Des Weiteren werden in diesem vereinfachten Schema keine Knotenlabels unterstützt und die Properties können nicht nach Belieben erweitert werden, ohne das Schema zu ändern. Über separate Typen- und Property-Tabellen lassen sich diese Probleme lösen [Sun+15]. In unseren Beispielen speichern wir das Label in einer Property `type`.

Die beiden folgende Tabellen zeigen eine dem in Abbildung 4.5 gezeigten Property-Graphen entsprechende Knoten- und Kantentabelle:

| **_id** | **type** | **name** | **alter** | **ort** |
|---|---|---|---|---|
| 77 | person | Katja | 37 | Rom |
| 19 | person | Judith | 35 | Bern |

**Tabelle 4.2:** Knotentabelle

In SQL sind für die Anfragen auf einem solchen Schema meist sehr viele Verbundoperationen nötig. Das liegt daran, dass zur Beantwortung einer

| start | ziel | label | seit |
|---|---|---|---|
| 77 | 19 | Freund | 2016-01-01 |

**Tabelle 4.3:** Kantentabelle

Frage meist sowohl die Knoten- als auch die Kantentabelle benötigt werden, häufig sogar mehrfach. In SQL ist dies kein Problem, da durch Indexe auf den Primär- und Fremdschlüsselspalten ein Verbund effizient durchgeführt werden kann. In NoSQL-Datenbanken ist diese Modellierung jedoch unüblich, da es zum einen selten Indexstrukturen gibt und zum anderen Verbundoperationen aufgrund der verteilten Speicherung kostspielig sind oder von vielen Systemen gar nicht erst unterstützt werden. In Abschnitt 4.8 stellen wir vor, wie Joins in NotaQL definiert und ausgeführt werden können, betonen aber bereits an dieser Stelle, dass die Sprache und die Ausführungsplattform darauf nicht optimiert sind. Aus diesem Grund möchten wir diese Variante mit nur einem einzigen simplen NotaQL-Beispiel abschließen. Die Anfrage nimmt die Kantentabelle als Eingabe und speichert das Ergebnis – die Anzahl der Freunde pro Person – in der Knotentabelle:

```
IN-FILTER: IN.label = 'Freund',
OUT._id <- IN.?(name()='start' || name()='ziel'),
OUT.anzahl_freunde <- COUNT()
```

Nach Ausführung der Transformation beinhaltet in der Knotentabelle die Spalte `anzahl_freunde` die Anzahl der Kanten mit dem Label `Freund`, die von dem jeweiligen Knoten ein- und ausgehen. Das Attributsprädikat in der zweiten Zeile sorgt dafür, dass eine Kante sowohl für die Person mit der ID des Start-Knotens als auch für die mit der ID des Ziel-Knotens zählt. Dies ist im gegebenen Skript nötig, da die eigentlich symmetrische Freundschaftsbeziehung asymmetrisch modelliert wurde. Dies ist ein gängiger Ansatz, um Redundanzen und mögliche Fehler zu vermeiden. Statt der symmetrischen Speicherung wird bei Abfragen eine Kante in zwei Richtungen traversiert.

### Adjazenzliste

Die Adjazenzliste zählt zusammen mit der Adjazenzmatrix zu den beiden gängigen Repräsentationen von Graphen in der Mathematik. Die *Adjazenzmatrix* wird jedoch zur Speicherung von Graphen in Datenbanksystemen seltener verwendet. Jede Zeile und jede Spalte der Matrix repräsentiert jeweils einen Knoten. Ist ein Eintrag in der $i$-ten Zeile und in der $j$-ten Spalte eine $1$, bedeutet dies, dass die beiden jeweiligen Knoten adjazent sind, also dass eine Kante von Knoten $v_i$ nach Knoten $v_j$ existiert. Im anderen Falle wäre der Zellenwert eine $0$.

Die *Adjazenzliste* beinhaltet für jeden einzelnen Knoten Referenzen auf alle seine Nachbarknoten. Bei einem gerichteten Graphen wird also für jeden Knoten gespeichert, zu welchen Knoten eine ausgehende Kante existiert. Diese Darstellung von Graphen wird vor allem für Graphanalysen eingesetzt. Dabei liegt der Graph meist in Form einer Textdatei auf einem lokalen oder verteilten Dateisystem vor und ist wie folgt ausgebaut:

```
77,19,28,39,21
19,40,28
```

Der erste Wert einer Spalte beinhaltet die Knoten-ID, die weiteren Werte die IDs der Nachbarn. Da sich im Gegensatz zur im vorherigen Abschnitt vorgestellten Kantentabelle alle Nachbar-IDs eines Knoten beieinander befinden, eignet sich die Adjazenzliste gut für verteile Berechnungen. Auf jedem Rechner in einem Cluster kann ein Teil der Adjazenzlisten lokal verarbeitet werden.

Neben Textdateien im CSV-Format sind auch Wide-Column-Stores aufgrund ihres flexiblen Schemas zur Speicherung von Adjazenzlisten gut geeignet. Die Knoten-ID wird dabei als Row-ID gespeichert und eine Spaltenfamilie beinhaltet die IDs der Nachbarknoten in Form von Spaltennamen. Die oben stehende Adjazenzliste sieht in einem Wide-Column Store wie folgt aus:

| **row-id** | **Freund** | | | | **Info** | | |
|---|---|---|---|---|---|---|---|
| 77 | 19:- | 28:- | 39:- | 21:- | name:Katja | alter:37 | ort:Rom |
| 19 | 40:- | 28:- | | | name:Judith | alter:35 | ort:Bern |

**Tabelle 4.4:** Adjazenzliste in einem Wide-Column-Store

Der Spaltenwert wurde hier leer gelassen. Er könnte aber auch die Kantenproperties beinhalten. Als Name der Spaltenfamilie wurde hier das Kantenlabel `Freund` verwendet, sodass mehrere verschieden beschriftete Kanten zwischen zwei Knoten möglich sind. In einer zusätzlichen Spaltenfamilie (hier: `Info`) können die Knotenproperties gespeichert werden. Somit sind alle Informationen aus Knotentabelle und Kantentabelle in einer einzigen Tabelle vereint. In Suchmaschinen ist eine solche Modellierung üblich, um Informationen zu einer Webseite in der einen Spaltenfamilie und Links zu anderen Webseiten in einer separaten Familie zu speichern [Cha+08b].

Das folgende NotaQL-Beispiel zeigt, wie die Kantenrichtung im gegebenen Graphen umgekehrt werden kann. Der Zugriff auf die Row-ID erfolgt mit `_r`, auf die Spaltennamen mit `_c` und auf die Spaltenwerte mit `_v`. Die Spaltenfamilien wird ähnlich zu Sub-Attributen in Dokumentendatenbanken mittels der Dot-Notation angegeben.

```
OUT._r <- IN.Freunde._c,
OUT.Freunde.$(IN._r) <- IN._v
```

Die Eingabezelle $(\_r, \_c, \_v) = (77, 19, -)$ erzeugt die Ausgabezelle $(\_r, \_c, \_v) = (19, 77, -)$. Jeder gefundene Spaltenname fügt also in der Zeile mit der Row-ID, die diesem Namen entspricht, eine Spalte mit der eigenen Row-ID als Namen hinzu. Die Werte bleiben unverändert. Im genannten Beispiel enthält der Ergebnisgraph eine ausgehende Kante von Knoten 19 zu Knoten 77, während ebendiese Kante im Eingabegraphen eine eingehende war.

In Dokumentendatenbanken findet man Adjazenzlisten oft in Form von Arrays. Auch hier können Knotenproperties zusammen mit den Adjazenzen gespeichert werden:

```
{ _id: 77, name: "Katja", alter: 37, ort: "Rom",
  freunde: [ 19, 28, 39, 21 ] }
```

Die folgende Beispiel-Transformation fügt in einer gegebenen Personen-Kollektion ein Attribut `freundesFreunde` hinzu, welches die IDs von indirekten Freunden enthält. Dazu wird zweimal über die Freundesliste iteriert. Beim ersten Iterieren wird bestimmt, welche Personendokumente geändert werden, beim zweiten Iterieren, welche IDs in deren Freundesfreunde-Liste hinzugefügt werden.

```
OUT._id <- IN.freunde[*],
OUT.freundesFreunde <- LIST(IN.freunde[*])
```

Beim Iterieren mit dem Stern-Operator werden Arrays in ihre Elemente aufgelöst. Logisch vorstellen kann man sich diese doppelte Iteration als ein Kreuzprodukt aus der Freundesliste mit sich selbst. Das oben stehende Beispieldokument, in welchem vier Freunde vorhanden sind, wird also in 16 ID-Paare entschachtelt: $(19, 19), (19, 28), \ldots, (21, 21)$. Jedes dieser Paare $(a, b)$ fügt $b$ in die Liste der Freundesfreunde von $a$ ein. Die Transformation hat zwei Schwachpunkte bezüglich der Reflexivität und Symmetrie. Zum einen enthält jede Person sich selbst in der Liste der Freundesfreunde. Dies kann durch anschließendes Herausfiltern oder über eine Syntaxerweiterung in NotaQL behoben werden. Letztere könnte wie folgt aus sehen: `IN.freunde[?(@!=IN.freunde[@])]`. Das erste `@` in dem *Listeniterationsfilter* steht für das aktuelle Listenelement, `IN.freunde[@]` bezieht sich auf das Listenelement des vorhergehenden Iterationsvorgangs. Durch den Filter wird die eigene ID übersprungen. Das zweite Problem bezüglich der Symmetrie tritt auf, wenn die Freundesliste wie hier nicht symmetrisch ist. In der Liste der Freundesfreunde würden viele IDs fehlen. Die vorgestellte Transformation ist erst dann anwendbar, wenn die Listen in den Eingabedaten zunächst in eine symmetrische Form gebracht werden. Dieser Vorgang ist ähnlich zum oben stehenden Beispiel, um Kanten in einem Graphen umzukehren.

### Baumstrukturen mittels Subdokumenten

Liegen die Daten in einer Baum-Struktur vor, können die Beziehungen zwischen Knoten als Subdokumente bzw. als Listen von Subdokumenten modelliert werden. Das bisher verwendete Freundschaftsbeispiel eignet sich hier nicht, da in Freundschaftsbeziehungen Zyklen auftreten. Nehmen wir nun den in Abbildung 4.6 gezeigten Graphen, welcher aus zwei Typen von Knoten besteht, zum einen Personen und zum anderen Status-Updates, welche von den Personen geschrieben werden. Eine Kante mit dem Label „schreibt“ verbindet die beiden Knoten.
In einer Dokumentendatenbank lässt sich der Graph wie folgt modellieren:

```
{ _id: 77, name: "Katja", alter: 37,
  status: [
   { sid:714, ts:1448269379,
     text:"Ich bin in Berlin angekommen!" },
   { sid:788, ts:1448372647,
     text:"Das Currywurst-Museum ist der Hammer!" }] }
```

Statt im Array lediglich die IDs der Nachbarknoten zu speichern, werden hier deren vollständige Properties abgelegt. Es ist auch denkbar, dass die

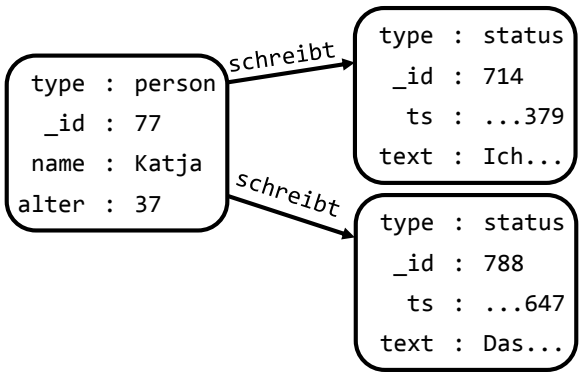



**Abbildung 4.6:** Ein Graph mit Personen- und Statusknoten

Subdokumente wieder eine oder mehrere Listen mit Subdokumenten enthalten können, sodass das ganze Dokument – ähnlich eines XML-Dokuments – eine Baumstruktur darstellt. In NoSQL-Datenbanken ist eine solche Speicherung üblich und hat viele Vorteile. Beim Verteilen der Datensätze auf mehrere Rechnerknoten liegt aufgrund der verschachtelten Speicherung alles zusammen, was zusammen gehört. Dieses Prinzip wird *gesamtheitsorientierte Speicherung* (engl. *aggregate orientation*) [SF12] genannt.

Das folgende NotaQL-Skript überführt das gegebene Schema in die zuvor gezeigte Modellierung mittels eines Arrays von Fremdschlüsseln:

```
OUT._id <- _id, OUT.$(IN.?(name()!='status')) <- IN.@,
OUT.status <- LIST(IN.status[*].sid)
```

In der ersten Zeile werden alle Attribute bis auf das Status-Attribut kopiert. Die zweite Zeile legt einen neuen Status-Array mit den Fremdschlüsselwerten an. Es ist zu beachten, dass die eigentlichen Statuseinträge bei dieser Transformation verloren gehen. Daher sollte man diese zuvor in eine separate Kollektion kopieren.

### Graphdatenbanken

Graphdatenbanken unterscheiden sich von allen anderen NoSQL-Datenbanken dadurch, dass es eine native Unterstützung für Verbindungen zwischen Datensätzen gibt. Dadurch ist es nicht mehr nötig, Beziehungen – wie in den vorherigen Absätzen beschrieben – mittels Fremdschlüsseln zu modellieren. Diese aus den relationalen Datenbanken stammende Modellierung findet man nämlich oft auch auch in Wide-Column-Stores und Dokumentendatenbanken wieder, obwohl diese meist keine – oder zumindest keine so effiziente – Möglichkeit bieten, Datensätze mittels Join-Operationen miteinander zu verbinden. Im Programmcode vieler NoSQL-Anwendungen sieht man allerdings häufig, dass Fremdschlüsselwerte in einem Datensatz gelesen werden, um anschließend mit den nativen API-Methoden die referenzierten Datensätze abzurufen. Diese meist als *Lookup* bezeichnete Operation ist ähnlich zu einem Join. Gute Datenmodellierungen vermeiden jedoch den Einsatz dieser Operation, indem Daten, die zusammengehören, zusammen gespeichert und gar nicht erst in verschiedenen Tabellen oder Kollektionen aufgeteilt werden. Bei Baum-Strukturen funktioniert dies mittels verschachtelter Speicherung gut, allerdings stoßen

die Datenbanken bei der Modellierung komplexer und vor allem zyklischer Graphen an ihre Grenzen.

Die Beziehungen zwischen Datensätzen erfolgt in Graphdatenbanken nicht mittels Fremdschlüsseln oder durch verschachteltes Speichern, wie in den drei oben vorgestellten Varianten. Stattdessen kommen Verbindungen zwischen Datensätzen als native Datenmodellkonzepte zum Einsatz, die intern meist als Zeiger auf die zu referenzierenden Datensätze gespeichert werden. Dies ermöglicht ein schnelles Traviersieren und erfordert keine weiteren Indexstrukturen wie Hashes oder B+-Bäume. Statt von Datensätzen reden wir in Graphdatenbanken von *Knoten* und *Kanten*, welche zusammen den *Graphen* bilden. Knoten repräsentieren Objekte, deren Attribute in *Properties* gespeichert werden, sie haben ein *Label* und eine Menge von ausgehenden und eingehenden Kanten zu anderen Knoten. All diese Informationen werden auf dem Speichermedium so abgelegt, dass ein schnelles Durchsuchen der Properties und der Kanten möglich ist. Über die Kanten kommt man schließlich zu den Kanten-Properties sowie zu den Nachbarknoten. Meist bieten Graphdatenbanken ähnlich zu relationalen Datenbanken Unterstützungen für *sekundäre Indexe*. In Neo4J [Neo] können diese unter Angabe eines Labels und einer Property definiert werden. Alle Knoten, die das betreffende Label besitzen, werden in den Index eingetragen, sodass ein schneller Zugriff über die Angabe eines bestimmten Wertes oder Wertebereiches in der gegebenen Property ermöglicht wird. Durch ebendiesen Index wird ein sequenzieller Scan über alle Knoten vermieten, welcher in Anwendungen oft vonnöten wäre, um den Einstiegspunkt einer Graphanfrage zu finden.

In diesem Abschnitt soll gezeigt werden, wie *Graph-Transformationen* mittels NotaQL durchgeführt werden können. Analog zu den bisher in dieser Arbeit gezeigten Transformationen nimmt auch eine Graph-Transformation eine Datenmenge als Eingabe – in diesem Fall einen Graphen – und transformiert diese in eine Ausgabe. Die Ausgabe kann entweder ein separat gespeicherter neuer Graph sein, oder aber einen existierenden Graphen erweitern oder ihn in-Place – also an Ort und Stelle – modifizieren. Eine NotaQL-Graphtransformationen verarbeitet einen Eingabegraphen Knoten für Knoten. Dieser knotenorientierte Ansatz [Tia+13] erlaubt es uns, konsistent und kompatibel mit dem NotaQL-Verarbeitungsmodell zu bleiben.

Betrachten wir nur die Knoten eines Graphen mit ihren Labels und Properties, kann die Sprache NotaQL wie in diesem Kapitel zuvor beschrieben unverändert zum Einsatz kommen. Mittels `OUT.propertyname` werden die Properties gesetzt, die die Ausgabeknoten besitzen sollen und mittels `IN.propertyname` auf Properties in der Eingabe zugegriffen. Auch das Filtern und Aggregieren ist wie in den weiter oben präsentierten Beispielen möglich. Das liegt daran, dass Graphdatenbanken nahezu äquivalent zu Dokumentendatenbanken sind, wenn man die Kanten weglässt. Wollen wir jedoch auf Kanten im Graphen zugreifen und deren Kanten-Properties lesen oder auf Kanten oder Properties von Nachbarknoten zugreifen, muss die NotaQL-Sprache erweitert werden. Auch zum Erzeugen von Kanten werden spezielle Sprachkonstrukte und Konstruktoren benötigt, welche wir in diesem Abschnitt vorstellen möchten.

**Kantenzugriff** Der Zugriff auf die Kanten ist ähnlich zum Zugriff auf die Elemente eines Listenattributs, da ein Knoten beliebig viele Kanten besitzen kann. Anders als bei Listen ist es jedoch bei Kanten nicht möglich, auf die `n`-te Kante zuzugreifen, weil auf ihnen keine Ordnung definiert ist. Jedoch ist es vonnöten, nur bestimmte Kanten auszuwählen, etwa nur diejenigen mit einem bestimmen Label, nur die ein- bzw. ausgehenden oder nur solche, welche bestimmte Propertywerte haben. Das Sytaxdiagramm in Abbildung 4.7 zeigt die verschiedenen Zugriffsmöglichkeiten auf Kanten. Mittels `IN._e` wird auf alle Kanten zugegriffen, `IN._>e` findet die ausgehenden und `IN._<e` die eingehenden Kanten [SKD17]. Im Anschluss kann mit dem `?`-Operator ähnlich zum weiter oben beschriebenen Attributsfilter ein Prädikat definiert werden, welches die zu findenden Kanten erfüllen müssen. Unter Angabe eines in Hochkommata geschriebenen Labels können die Kanten mit dem entsprechenden Label gefunden werden. Beispielsweise findet `IN._e?('Freund')` alle Kanten – egal ob ein- oder ausgehend – mit dem Label Freund. Dieses Prädikat ist eine von NotaQL unterstützte Kurzschreibweise für `IN._e?(_l = 'Freund')`, wobei `_l` für das Kantenlabel steht. Wird an mehreren Stellen in einem NotaQL-Skript auf Kanten zugegriffen, kann mittels `[@]` auf die Kante der aktuellen Iteration zugegriffen werden. Analog zum Zugriff auf Listenelemente mittels der gleichen Schreibweise wird so verhindert, dass ein erneutes Iterieren über die Menge aller Kanten erfolgt. Folgt nach der Auswahl der Kanten ein Punkt-Symbol, kann auf die Kanten-Properties zugegriffen werden. Ein Unterstrich stattdessen gibt an, dass man sich nun zu den Nachbarknoten bewegt, auf welchen ebenfalls wieder Prädikate definiert werden können, und auf welchen ebenfalls auf Properties sowie auf deren Kanten und Nachbarn zugegriffen werden kann.

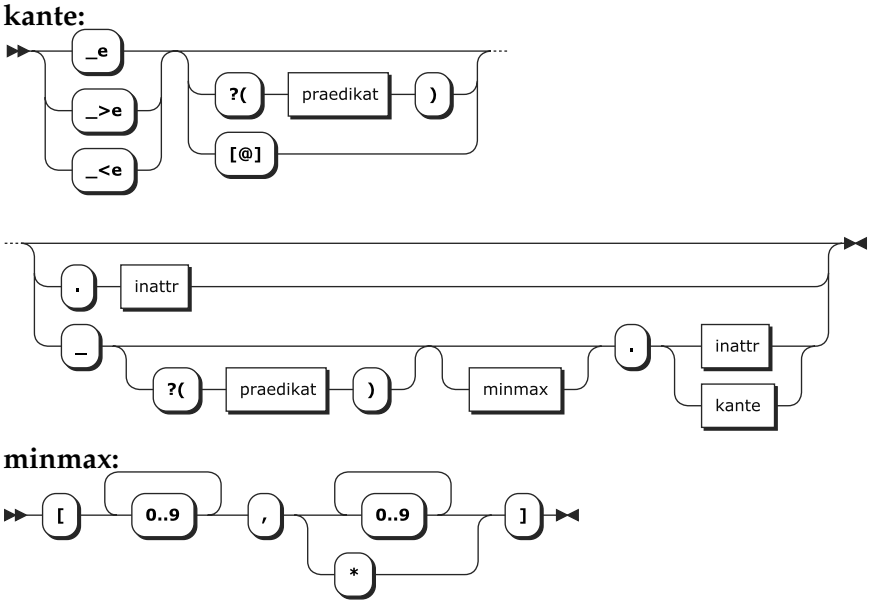



**Abbildung 4.7:** Zugriff auf Kanten in der Eingabe

Statt einer Kante nur einen einzelnen *Hop* zu folgen, kann Kanten, die das angegebene Prädikat erfüllen, auch mehrere Schritte weit gefolgt werden. Über eine Minimum-Maximum-Angabe kann die Länge einer zu traversierenden Kanten-Kette definiert werden. Der Standardfall ist `[1,1]`, also

genau ein Hop. Mittels `[0,1]` wird neben dem Nachbarknoten auch der aktuelle Knoten selbst gefunden. Der Ausdruck `IN._e_[3,3]` findet alle Knoten, die drei Hops entfernt sind. Er ist äquivalent zu `IN._e_._e_._e_`. Die Minimum-Maximum-Option spart aber nicht nur Schreibarbeit bei Definition einer Graph-Transformation, sondern ermöglicht es auch, Pfade unterschiedlicher Länge auf einmal zu verfolgen. Auch ein Finden der transitiven Hülle ist möglich. Die Option `[1,*]` findet die direkten und indirekten Nachbarknoten bis zu einer beliebigen Tiefe. Besitzt der Graph Zyklen, ist darauf zu achten, dass mittels eines Prädikats verhindert wird, dass keine Zyklen traversiert werden, damit ein Terminieren garantiert ist.
Das folgende NotaQL-Skript fügt jedem Personen-Knoten in einer Graphdatenbank einige Properties hinzu:

```
IN-FILTER: IN.type='Person',
OUT._id <- IN._id,
OUT.anzFreunde <- COUNT(IN._e?('Freund')_.id),
OUT.anzAlteFreunde <- COUNT(
                    IN._e?('Freund')_?(alter>80)._id),
OUT.aeltesteFreundschaft <-MIN(IN._e?('Freund').seit),
OUT.nameDerMutter <- IN._>e?('Mutter')_.name
```

Die hier vorgestellten Attribut-Mappings verwenden Aggregatfunktionen, die direkt auf den Eingabeknoten ausgeführt werden können. Hier ist keine Knoten-übergreifende Aggregation vonnöten, da beim Lesen von Propertywerten von Kanten oder Nachbarknoten listenwertige Resultate zurückgeliefert werden. Diese Listen dienen als direkte Eingabe für Aggregatfunktionen. Beim Zählen der Freunde werden die Kanten mit dem Label `Freund` in beliebiger Richtung traversiert und die Anzahl der Nachbarknoten bestimmt. In der vierten Zeile des Skriptes kommt ein Prädikat hinzu, welches nur auf diejenigen Nachbarknoten zutrifft, die einen größeren Wert als 80 in der Property `alter` besitzen. Die Ermittlung des Datums der ältesten Freundschaftsbeziehung erfolgt mittels Anwendung der Minimum-Funktion auf den Werten der Kantenproperty `seit`. Die letzte Zeile verwendet keine Aggregatfunktion, da davon ausgegangen wird, dass es pro Personen-Knoten nur maximal eine ausgehende Mutter-Kante gibt. In diesem Fall kann direkt auf die `name`-Property des Nachbarknotens zugegriffen werden. Besitzt ein Knoten keine ausgehende Mutter-Kante, wird die `nameDerMutter`-Property nicht gesetzt. Besäße er mehrere solcher Kanten, schlüge die Transformation fehl.

Das folgende Beispiel zeigt, dass zwei Aggregationsfunktionen ineinander geschachtelt werden können, wenn aus Eingabeknoten Listenwerte extrahiert werden und zudem eine Gruppierung erfolgt. Die innere Funktion zählt die Anzahl der Freunde pro Person – genauer: die Anzahl der Elemente in der Liste der ID-Werte der Nachbarknoten über die Freundeskanten –, anschließend werden diese Anzahlen nach dem Ort, in dem die Person wohnt, gruppiert und die durchschnittliche Anzahl an Freunden pro Ort berechnet:

```
IN-FILTER: IN.type='Person',
OUT._id <- IN.ort,
OUT.avgAnzFreunde <- AVG(COUNT(IN._e?('Freund')_.id))
```

Der Ausdruck `IN._e?('Freund')_.id` liefert genau wie die Ausdrücke `IN.hobbys[*]` oder `IN.*` ein Objekt vom internen NotaQL-Typ *List*. Dieser dient lediglich als Eingabe für Aggregatfunktionen und kann nicht in die Ausgabe geschrieben werden. Vor dem Gruppieren von Werten dürfen ebenfalls keine listenwertigen Daten vorliegen. Liegen listenwertige Daten bereits in der Eingabe vor, müssen diese vor der Gruppierung aggregiert werden. Die einzige Ausnahme für all diese Bedingungen sind Listen, die aus keinem oder lediglich aus einem Element bestehen. Diese dürfen auch ohne eine Aggregatfunktion in einem Attribut-Mapping vorkommen. Leere Listen werden in diesem Falle als Datenwert `missing` interpretiert. Das Schreiben von `missing` in ein Ausgabeattribut sorgt für das Nichtvorhandensein dieses Attributs im Ausgabedatensatz. Einelementige Listen vom Typ *List*<*T*> werden beim Gruppieren und Schreiben direkt als ein Objekt vom Typ $T$ interpretiert. Zu Veranschaulichung folgen zwei ungültige Beispiele:

```
OUT._id <- IN.ort,                          #schlaegt fehl
OUT.anzFreunde <- COUNT(IN._e?('Freund')_.id)
```

Da der verwendete Kantentraversierungsausdruck einen Listenwert liefert, wird die Funktion `COUNT` bereits vor dem Gruppieren aufgerufen, um die Anzahl der Freunde einer Person zu zählen. Da eine Gruppierung nach dem Ort erfolgt, können Kollisionen im Attribut `anzFreunde` auftreten, wenn zwei Personen, die im gleichen Ort wohnen, eine unterschiedliche Anzahl von Freunden haben.

```
OUT._id <- IN.ort,                          #schlaegt fehl
OUT.freunde <- IN._e?('Freund')_.id
```

Diese Transformation schlägt bereits vor der Gruppierung der Daten fehl, da keine Aggregation auf einem vorliegenden Listenwert durchgeführt wurde.

**Erzeugung von Kanten** Bei Graph-Transformationen mit NotaQL erfolgt das Erzeugen von Kanten im Ausgabegraphen, nachdem alle Knoten samt ihrer Properties erstellt wurden. Diese Ausführungssemantik verhindert es, dass Kanten zu noch nicht existierenden Knoten erstellt werden. Das Syntaxdiagramm in Abbildung 4.8 zeigt zwei Möglichkeiten, Kanten zu erzeugen. Es kann entweder mit dem Kantenkonstruktor geschehen oder durch das Übernehmen von Kanten aus dem Eingabegraphen. In beiden Fällen ist die Richtung der Kante anzugeben. Mit `OUT._>e` werden ausgehende und mit `OUT._<e` eingehende Kanten erzeugt. Bei Verwendung des Kantenkonstruktors muss auf der linken Seite des Zuweisungspfeils ein Prädikat angegeben werden, womit der Zielknoten der Kante festgelegt wird. Ist das Prädikat für keinen Knoten erfüllt, wird keine Kante erzeugt. Trifft es auf mehrere Knoten zu, werden mehrere Kanten erzeugt. Auf der rechten Seite des Zuweisungspfeils werden innerhalb des Kantenkonstruktors das Label und die optionale Liste an Kanten-Properties angegeben.
Im folgenden Beispiel werden zweierlei Kanten erzeugt. Zum einen soll jeder die Person Simon als Freund erhalten, zum anderen wird eine Kante zu der Großmutter jeder Person erzeugt:

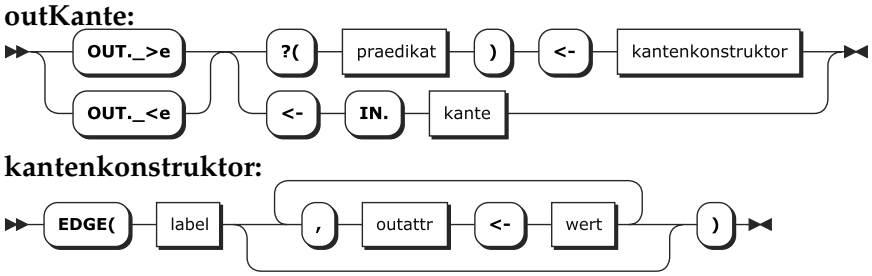



**Abbildung 4.8:** Erzeugung von Kanten

```
IN-FILTER: IN.type='Person',
OUT._id <- IN._id,
OUT._>e?(name = 'Simon')
        <- EDGE('Freund', seit <- 2017),
OUT._>e?(_id = IN._>e?('Mutter' || 'Vater')_
                  ._>e?('Mutter')_._id)
        <- EDGE('Großmutter', via <- IN._e[@]._l).
```

Gibt es im Graphen $n$ Personenknoten und $s$ Personen mit dem Namen Simon, werden $n \cdot s$ neue Freundschaftskanten erzeugt. Setzen wir voraus, dass jede Person nicht mehr als einen Vater und eine Mutter hat, entstehen durch die letzte Zeile im Transformationsskript bis zu $2n$ neue Kanten mit dem Label Großmutter. Die Ziele der Kanten sind diejenigen Knoten, die eine der IDs haben, welche sich über das Traversieren zur Mutter des Vaters oder der Mutter finden lassen. Ähnlich wie bei XPath oder MongoDB ist auch bei NotaQL eine Überprüfung auf Gleichheit auf Listen genau dann wahr, wenn es eine Übereinstimmung bei mindestens einem der Listenelemente gibt.

Beim Kopieren von Kanten ist kein Ziel anzugeben. Kanten werden samt ihrem Ziel, ihrem Label und ihren Properties übernommen. Um einen Graphen vollständig mit allen Knoten und Kanten zu kopieren, kann die folgende Transformation zum Einsatz kommen:

```
OUT._id <- IN._id,
OUT.$(IN.*.name()) <- IN.@,
OUT._>e <- IN._>e
```

Die erste Zeile übernimmt die Knoten-ID, die zweite Zeile alle Properties, die dritte Zeile alle ausgehenden Kanten. Zu beachten ist, dass man entweder nur die ein- oder nur die ausgehenden Kanten kopiert, da die jeweils anderen bereits beim Verarbeitern des Nachbarknoten übernommen werden. Würde man aus- und eingehende Kanten übernehmen, wären im Ausgabegraphen alle Kanten doppelt vorhanden. Durch das Ändern der Pfeilrichtungen in `OUT._<e` könnte man die Kantenrichtung umkehren. Bei komplexeren Kantenkopieraktionen kommt wie im obigen Beispiel der Kantenkonstruktor in Kombination mit einem Zugriff auf die Kanten des Eingabegraphen zum Einsatz. So können beispielsweise existierende Kanten auf andere Zielknoten umgelenkt werden oder Properties oder Label beim Kopieren neue Werte erhalten.

**Iterative Berechnungen** Viele Graph-Algorithmen sind *iterativ*. Das bedeutet, sie müssen mehrmals hintereinander ausgeführt werden, um ein Endergebnis zu berechnen oder um sich einem Ergebniswert anzunähern. Iterative Algorithmen modifizieren den Graphen immer an Ort und Stelle. Es wird kein neuer Ausgabegraph erzeugt, er ist gleich dem Eingabegraphen. Berechnungen, die auf einer Breitensuche basieren und bei denen in jeder Iteration zu jedem Knoten Propertywerte von direkten Nachbarknoten aggregiert werden, erfordern meist bis zu $d + 1$ Durchläufe, wobei $d$ für den Durchmesser des Graphen steht, also die maximale Distanz zwischen zwei Knoten. Die Distanz zwischen zwei Knoten ist als die Anzahl der Kanten definiert, die traversiert werden müssen, um von einem Knoten zum anderen zu gelangen.

Für iterative Berechnungen kommt in NotaQL die `REPEAT`-Klausel zum Einsatz. Durch die Eingabe einer maximalen Anzahl von Iterationen $n$ wird die gegebene Transformationsvorschrift $n$ mal ausgeführt. Wird zwischen zwei Iterationen festgestellt, dass sich der Graph seit der vorherigen Iteration nicht geändert hat, wird vorzeitig abgebrochen. Bei $n = -1$ ist die Anzahl der Iterationen unbegrenzt. Es erfolgen so viele Wiederholungen, bis keine Änderung mehr erfolgt. Soll nicht die Gesamtheit der Properties aller Knoten im Graphen einen fixen Endzustand erreichen, sondern lediglich eine gewisse Knotenproperty konvergieren, kann diese Property zusammen mit einem maximalen Änderungsprozentsatz angegebenen werden. Gibt es nach der Ausführung einer Iteration einen Knoten, dessen Wert sich in der angegebenen Property seit der vorherigen Iteration um mehr als diesen Prozentsatz verändert hat, wird eine weitere Iteration durchgeführt. Andernfalls ist die Berechnung abgeschlossen.

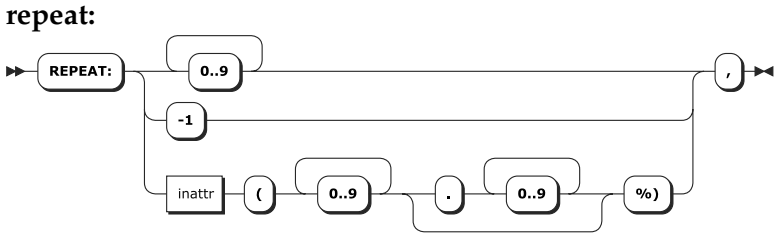



**Abbildung 4.9:** Iterative Berechnungen mit der `REPEAT`-Klausel

Gegeben sei als Beispiel der Graph in Abbildung 4.10 mit dem Durchmesser $d = 2$. Zu Beginn seien die `dist`-Propertywerte nicht gesetzt. Das folgende NotaQL-Skript – bestehend aus zwei Transformationen – weist zunächst dem Knoten $K_1$ den Distanzwert $0$ zu und berechnet anschließend auf iterative Art und Weise die Distanz zwischen $K_1$ und jedem anderen Knoten. Wegen des bekannten Graphdurchmessers sind nicht mehr als zwei Iterationen nötig. In der ersten Iteration erhalten alle Nachbarn von $K_1$ die Distanz $1$, in der zweiten deren Nachbarn die Distanz $2$. Knoten, zu denen es keinen Pfad zu $K_1$ gibt, werden die `dist`-Property nicht erhalten.

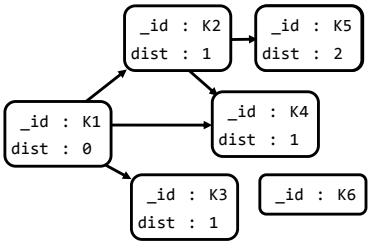



**Abbildung 4.10:** Distanz-Graph

```
IN-FILTER: IN._id = 'K1'
OUT._id <-IN._id,
OUT.dist = 0;

REPEAT: 2,
IN-FILTER: IN.dist,
OUT._id <- IN._e_._id,
OUT.dist <- min(IN.dist)+1
```

Würde beim gegebenen Graphen eine größere Anzahl an Wiederholungen als 2 oder sogar `REPEAT: -1` angegeben werden, wäre das Ergebnis zwar das gleiche, allerdings würden in diesen Fällen exakt drei Iterationen ausgeführt werden. Zusätzlich zu den beiden oben beschriebenen Iterationen erfolgt eine weitere, da erst nach dieser erkannt werden kann, dass sich der Graph nicht mehr verändert hat. Die gezeigte Transformation (ab Zeile 5) wird auf allen Knoten ausgeführt, die die `dist`-Property besitzen, also diejenigen, die bereits eine bekannte Distanz zu $K_1$ haben. In der ersten Iteration ist dies nur $K_1$ selbst. Die Distanz der Nachbarn dieser Knoten werden auf die eigene Distanz plus eins gesetzt. Gibt es zu einem Knoten mehrere Pfade, sorgt die Minimum-Funktion dafür, dass der kürzeste davon gewählt wird.

Bei dem *PageRank*-Algorithmus [Pag+99] handelt es sich um einen iterativen Graph-Algorithmus, der auf einer Fixpunkt-Berechnung basiert. Wendet man ihn auf einem Web-Graphen an, erhalten Webseiten einen um so höheren PageRank-Wert, je öfter sie von Webseiten mit ebenfalls einem hohen PageRank-Wert referenziert werden. Der PageRank eines Knoten $q$ ist wie folgt definiert:

$$PR(q) = \sum_{p \in in(q)} \frac{PR(p)}{|out(p)|}$$

Bevor die Berechnung startet, werden die PageRank-Werte aller Knoten initialisiert, typischerweise mit dem Wert $1/N$, wobei $N$ die Anzahl der Knoten im Graph ist. Dieses Initialisieren sowie die anschließende iterative Berechnung, die so lange erfolgt, bis sich die PageRank-Werte aller Knoten nur noch um maximal 0.0005% seit der vorherigen Iteration verändern, zeigt das folgende NotaQL-Skript:

```
OUT._id <- IN._id,                  # Initialisierung
OUT.pagerank <- 1/NumVertices();

REPEAT: pagerank(0.0005%),          # Iteration
OUT._id <- IN._>e_._id,
OUT.pagerank <- SUM(IN.pagerank/COUNT(IN._>e._id))
```

Ein Knoten $p$, welcher selbst den PageRank-Wert $PR(p)$ hat und $|out(p)|$ ausgehende Kanten besitzt, erhöht den PageRank-Wert seines jeden Nachbarn $q$ um $PR(p)/|out(p)|$. Die gezeigte Transformation berechnet für jeden Nachbarknoten, der über eine ausgehende Kante erreicht werden kann, deren PageRank-Wert. Diese Vorgehensweise orientiert sich an der PageRank-Implementierung mittels MapReduce [LD10], welche als Eingabe Adjazenzlisten erhält.

Ein Problem beim gegebenen Skript tritt bei Knoten auf, die keine eingehenden Kanten haben. Bei solchen Knoten würde die PageRank-Property fehlen. Das Problem lässt sich durch eine Einführung eines Random-Jump-Faktors [Pag+99] im gegebenen Skript leicht beheben.

## 4.6 Datentypen in NotaQL

Jedes Datenbanksystem hat sein eigenes Datenmodell und seine eigenen Datentypen. NotaQL ist eine Sprache, die auf einer Vierzahl von Systemen anwendbar sein soll. Nichtsdestoweniger ist ein internes Datenmodell vonnöten, um Filter, Attribut-Mappings, Gruppierungen, Aggregationen und den Aufruf von Funktionen adäquat definieren zu können. Bei der Entwicklung des NotaQL-Datenmodells haben wir uns an JSON [JSOa] orientiert, welches die folgenden sechs Typen unterstützt:

- *Number*: Eine positive oder negative ganze Zahl oder Fließkommazahl; z. B. `17`, `0`, `12.8`, `-3` oder `6,022e23`.
- *String*: Eine Zeichenkette; z. B. `"Hallo"`, `"Mike Müller"` oder `""`.
- *Object*: Eine ungeordnete Menge von Attribut-Wert-Paaren; Attribute sind Zeichenketten und Werte von einem der sechs Typen, z. B. `{a:5,b:"abc"}` oder `{}`.
- *Array*: Eine geordnete Sequenz von Werten; die Werte können wiederum von einem der sechs Typen sein; z. B. `[1,2,3,4,5]`, `[]` oder `[1,"Hallo",[1,2,3,4,5],{a:5,b:"abc"}]`.
- *Boolean*: `true` oder `false`.
- *Null*: `null`.

Zudem wurde NotaQL um die folgenden internen Typen erweitert:

- *Missing*: Ein Attribut, welches gelesen werden soll, aber nicht vorhanden ist, erhält diesen Wert. Wird `missing` als Wert in ein Ausgabeattribut geschrieben, sorgt dies für ein Nichtvorhandensein dieses Attributs im Ausgabedatensatz. Über die `DROP()`-Operation kann manuell das Löschen eines Attributs erzwungen werden. In [OPV14] ist eine umfangreiche Analyse zu finden, wie verschiedene NoSQL-Systeme `missing` beim Vergleich mit Werten, `missing` oder `null` interpretieren.
- *List*: Ähnlich zum Array, allerdings nur für interne Zwecke. Listenwertige Daten befinden sich nicht direkt in der Eingabe und können nicht in eine Ausgabe geschrieben werden. Sie entstehen beim Iterieren über Arrays, Kanten und andere mehrwertigen Attribute sowie beim Gruppieren von Zellen. Aggregatfunktionen nehmen Listenwerte entgegen und liefern einen der oben genannten sechs JSON-Typen zurück.

Es fällt auf, dass es keine NotaQL-Typen für Knoten oder Kanten gibt. Auch unterstützt NotaQL bewusst keine Hash-Maps oder Sorted Sets [Red], um

das Datenmodell so schlicht wie möglich zu halten. Letztere Typen können beim Auslesen in „Arrays von Objects“ gewandelt werden und beim Schreiben wieder zurück in die nativen Datenbanktypen. Alternativ könnte ein Entwickler aber auch benutzerdefinierte Typen für diese Zwecke erstellen (siehe Kapitel 4.7). Beim Zugriff und beim Erzeugen von Kanten liegt eine Kante intern als *Object* vor, welches aus einer Zielknoten-ID, einer booleschen Richtungsangabe, sowie Label und Properties besteht. Beim Schreiben eines solchen Objekts wird ein natives Kantenobjekt im jeweiligen Graphdatenbanksystem erzeugt.

Die Beschränkung auf die genannten acht internen Typen sorgt für ein einfaches und schlankes NotaQL-Datenmodell. Da diese acht Typen oft sogar eine Obermenge von denjenigen Typen darstellen, die von NoSQL-Datenbanken unterstützt werden, kann eine direkte Abbildung der Typen erfolgen. Natürlich decken die acht Typen nicht alles ab, was derzeit in NoSQL-Datenbanken unterstützt wird, geschweige denn, was zukünftige Systeme bieten. Sicherlich ist es sinnvoll, noch einen Typen für Binärdaten anzubieten. Viele Probleme können jedoch meist schon durch eine Typ-Abbildung oder die Erstellung eines benutzerdefinierten Typs (siehe unten) gelöst werden.

Die Sprache NotaQL unterstützt die zu den oben beschriebenen Typen bekannten grundlegenden zweistelligen Operationen, diese sind wie folgt:

- *Number*: Addition (+), Subtraktion (−), Multiplikation (⋆), Division (/)
- *String*: Konkatenation (+)
- *Boolean*: Konjunktion (&&), Disjunktion (||)

Des Weiteren werden Klammerausdrücke bis zu einer beliebigen Tiefe unterstützt sowie die Negation von booleschen Ausdrücken. Die sechs aus JSON entnommenen Basistypen können als Parameter- und Rückgabetypen von Funktionen verwendet werden. Eine Besonderheit stellen Aggregatfunktionen dar, welche als ersten Parameter einen Listenwert und zudem optional weitere Parameter besitzen. Weitere Details zu Funktionen befinden sich im folgenden Unterkapitel. Die soeben aufgezählten Operationen werden auch vom internen Typ List unterstützt, um komponentenweise Berechnungen durchführen zu können:

Gegeben sind zwei Listen $l_1, l_2$ vom Typ $List$<$T$> mit $|l_1| = |l_2| = L$ und eine Operation $\circ : T \times T \to T \in \{+, -, *, /, \&\&, ||\}$. Unter der Voraussetzung, dass $l_1[i] \circ l_2[i]$ definiert ist für alle $i \in [0..L-1]$, gilt:

$$l_1 \circ l_2 = map(\circ, zip(l_1, l_2))$$

Dabei sind die beiden aus der funktionalen Programmierung stammenden Funktionen $map$ und $zip$ wie folgt definiert:

$$\begin{aligned}
&map(\circ, [\,]) = [\,] \\
&map(\circ, (a, b) : l) = a \circ b : map(\circ, l) \\
&zip(al, [\,]) = zip([\,], bl) = [\,] \\
&zip((a : al), (b : bl)) = (a, b) : zip(al, bl)
\end{aligned}$$

Die Zip-Funktion kombiniert also zwei Listen komponentenweise zu einer Liste von 2-Tupeln. Die Map-Funktion wendet die gegebene Operation auf jedem Element der Liste an – hier also auf jedem 2-Tupel – und liefert eine Liste mit den Ergebnissen der Operation zurück.

Diese Ausführungssemantik macht es möglich, in NotaQL mit listenwertigen Daten zu rechnen, bevor sie als Eingabe in eine Aggregatfunktion dienen. Analog können wir die oben stehende Definition auf einstellige Operationen wie die Negation auf booleschen Werten und auf beliebigstellige benutzerdefinierte Funktionen erweitern. Bevor benutzerdefinierte Funktionen im nächsten Abschnitt detaillierter behandelt werden, schließen wir diesen Abschnitt mit einem Beispiel ab:

Eingabe: `{ _id:1, hobbys: ["Tennis", "Gitarre"] }`

```
OUT._id <- IN._id,
OUT.HOBBYS <- LIST(UPPERCASE(IN.hobbys[*]))
```

Ausgabe: `{ _id:1, HOBBYS: ["TENNIS", "GITARRE"] }`

`IN.hobbys` liefert einen *Array*. Mittels `[*]` wird daraus eine *List<String>* $l$ erzeugt. Der Aufruf der einstelligen Funktion `UPPERCASE` wird interpretiert als $map(\texttt{UPPERCASE}, l)$ und liefert als Ausgabe eine *List<String>* $l'$, die die nun groß geschriebenen Hobbys enthält. Die Aggregatfunktion `LIST` erzeugt daraus wieder einen Array, welcher beispielsweise in ein MongoDB-Dokument geschrieben werden kann.

## 4.7 Benutzerdefinierte Funktionen und Typen

Ein Ziel der Sprache NotaQL ist ihre Erweiterbarkeit. Während im nächsten Kapitel über die Erweiterbarkeit auf andere Datenbanksysteme gesprochen wird, beschäftigt sich dieser Abschnitt mit der Erweiterbarkeit der Sprache um benutzerdefinierte Funktionen. Diese ermöglichen Datentransformationen, welche über einfache Attribut-Mappings, Aggregationen und arithmetischen Berechnungen hinausgehen. Dem Benutzer wird die Möglichkeit gegeben, beliebige Berechnungen auf den von der NotaQL-Plattform gelesenen Eingabedaten durchzuführen. Diese können mathematische Berechnungen oder String-Manipulationen sein, aber auch eine Kopplung zu einem externen System oder zu existierenden Programmbibliotheken darstellen. Letztere ermöglichen komplexe Analysen von Multimediadaten wie Text, Bild und Ton, sowie Klassifizierungen, Clustering und andere Machine-Learning-Algorihmen.

Eine Funktion wird definiert über einen Namen, sie hat eine Liste von Eingabeparametern sowie einen Ausgabeparametertypen. Da NotaQL ein *Werte-basiertes Update-Modell* hat, also alle Änderungen mittels Setzen von Werten erfolgen muss, gibt es keine Prozeduren oder Funktionen vom Typ `void`. Jene kommen in Programmiersprachen oder Datenbanken vor, um mittels Seiteneffekten Objekt- oder Datenbankzustände zu ändern. In NotaQL sind Funktionsaufrufe stets *seiteneffektfrei*.

Die Ausgabetypen von benutzerdefinierten Funktionen können die im vorherigen Abschnitt vorgestellten sechs Basistypen *Number*, *String*, *Object*, *Array*, *Boolean* oder *Null* sein. Zudem darf die Ausgabe auch von einem

benutzerdefinierten Typen sein – mehr dazu weiter unten in diesem Abschnitt. Das gleiche gilt für die Eingabetypen. Hier darf jedoch zusätzlich der Typ *List* verwendet werden. Ist dies der Fall, handelt es sich bei der implementierten Funktion um eine *benutzerdefinierte Aggregatfunktion*. Der Typ *Missing* wird weder als Eingabe- noch als Ausgabetyp verwendet. Ist ein Attribut, welches als Eingabe für eine Funktion dienen soll, `missing` – also nicht präsent –, so ist auch die Ausgabe der Funktion stets `missing` ohne die Funktion selbst aufzurufen.

Durch das *Überladen von Funktionen* ist es möglich, mehrere Funktionen mit dem gleichen Namen zu definieren, z. B. die Funktionen $toNumber : Number \rightarrow Number$, $toNumber : String \rightarrow Number$ und $toNumber : Null \rightarrow Null$. Dies ist in NoSQL-Datenbanken aufgrund der dort fehlenden vertikalen Homogenität besonders hilfreich. In verschiedenen Datensätzen kann ein und das selbe Attribut verschieden getypt sein. Ein Aufruf der jeweils passenden Funktion macht es zur Laufzeit möglich, die Kontrolle über die vorliegende Heterogenität zu behalten.

Da NotaQL lediglich eine Transformationssprache sein soll, gibt es keine DDL[3]-Sprachkonstrukte zur Erzeugung benutzerdefinierter Funktionen und benutzerdefinierter Typen. Sie müssen extern – also in einer Programmiersprache – erstellt und anschließend in der NotaQL-Plattform registriert werden. Ein benutzerdefinierter Typ kann beispielsweise in Java wie folgt definiert werden:

```
1 public class Image implements UDT {
2  int width;
3  int height;
4  Byte[] data;
5 }
```

Etwaige in der Klasse definierten Methoden bleiben beim Registrieren des Typs unbeachtet. Dies ermöglicht ein Wiederverwenden von Klassen aus existierenden Anwendungen. Der neu erstellte Typ kann in Engines (siehe Kapitel 4.8) zum Einsatz kommen, um Daten dieses Typs aus einer Datenbank zu lesen oder sie zu schreiben. Des Weiteren kann der benutzerdefinierte Typ als Ein- und Ausgabetyp von benutzerdefinierten Funktionen verwendet werden. Das folgende Beispiel zeigt, wie in Java eine Funktion zur Änderung von Bildgrößen definiert werden kann:

```
1 @AutoService(FunctionProvider.class)
2 public class ImageUDFs implements FunctionProvider {
3  @SimpleFunction(name="RESIZE")
4  public static Image resize(Image img,
5    NumberValue width, NumberValue height) {
6   return MyResizer.resize(img, width, height);
7  }
8 }
```

In der gezeigten Funktion kommen als Parametertypen zum einen der soeben erstellte benutzerdefinierte Typ `Image` und zum anderen der NotaQL-Typ `NumberValue` zum Einsatz. Innerhalb einer *FunctionProvider*-Klasse können beliebig viele Funktionen definiert werden. Nach der Registrierung

[3] *Data Defintion Language*: z. B. `CREATE TYPE` oder `CREATE FUNCTION` in SQL

stehen diese unter dem angegebenen Namen zur Verwendung in NotaQL-Skripten zur Verfügung. Das folgende Beispiel verwendet die soeben definierte Funktion zur Erstellung von Miniaturansichten von Profilfotos:

```
OUT._id <- IN._id,
OUT.miniFoto <- RESIZE(IN.profilfoto, 320, 240)
```

Benutzerdefinierte Aggregatfunktionen verwenden den internen NotaQL-Typen $List$. Um unser Bilder-Beispiel zu erweitern, könnte man eine Aggregatfunktion zur Erstellung eine Kollage aus mehreren Bildern definieren. Der Typ für Listenwerte darf nicht verwechselt werden mit dem Typen für Arrays. Ersterer beinhaltet Werte aus einer Gruppierung oder solche Werte, die aus einer Iteration beispielsweise über Attribute, Kanten oder Arrayelemente stammen. Der Array-Typ steht für einen Array als Ganzes, welchen man bereits in den Eingabedaten finden kann. Zur Umwandlung eines Listenwerts in einen Array kommt die Aggregatfunktion `LIST` zum Einsatz. Für die umgekehrte Richtung ist der Iterationsschritt `[*]` vonnöten.

## 4.8 Systemübergreifende Transformationen

Die NoSQL-Landschaft ist heterogen und verändert sich ständig. Jedes System hat eigene Datenmodelle, eigene Typen und eigene Zugriffsmethoden. Trotz fehlender Standards ist es ein Ziel der Sprache NotaQL, die verschiedenen Arten von Systemen zu unterstützen. Da die NoSQL-Welt auch ständig in Bewegung ist, ständig neue Systeme entstehen und bestehende Systeme erweitert werden, muss die Sprache auch auf zukünftige Neuerungen und Änderungen vorbereitet sein.

Die Transformationssprache NotaQL ist in dreierlei Hinsicht auf Heterogenität optimiert: Heterogenität in den Daten, Metadaten sowie Meta-Metadaten. Heterogene Daten sind gemäß des *Variety*-Begriffs in der Big-Data-Definition vor allem in NoSQL-Datenbanken zu finden. Dort herrscht meist weder horizontale Homogenität (unterschiedliche Datensätze haben unterschiedliche Attribute) noch vertikale Homogenität (in verschiedenen Datensätzen ist ein und das selbe Attribut unterschiedlich getypt). Ersteres kann mittels namenloser Zugriffspfade wie `IN.*` oder Attributsprädikaten wie `IN.?(@>0)` überwunden werden, letzteres durch Überladen von Funktionen. Auch bei heterogenen Metadaten sind die soeben genannten namenlosen Zugriffspfade nützlich, da genaue Attributnamen nicht bekannt sind und in verschiedenen Datensätzen unterschiedlich lauten können. In diesem Unterkapitel geht es vor allem aber um die Überwindung von Heterogenität auf Meta-Metadatenebene. Diese Ebene beschreibt, welche Datenstrukturen sich innerhalb eines Systems modellieren lassen. In einer relationalen Datenbank sind dies Tabellen, in Dokumentendatenbanken JSON-Dokumente mit Unterdokumenten und Arrays. Genau wie auf der Datenebene und Metadatenebene kann es auch auf Metametadaten-Ebene zu Konflikten kommen, für welche es eine Lösung mit NotaQL geben soll. Ein solcher Konflikt kann immer dann auftreten, wenn für die Transformationseingabe und -ausgabe unterschiedliche Systemen verwendet werden und *Datenmodell-Heterogenität* überwunden werden muss. Abbildung 4.11 zeigt Beispiele für Konflikte auf den drei genannten Ebenen. In (a) wird für das Gehalt links der Datentyp $Number$ und rechts $Object$ verwendet. (b)

zeigt zwei Dokumente, in denen verschiedene Attributnamen für das Gehalt verwendet wurden. Die in (c) vorliegenden Datensätze liegen möglicherweise auf verschiedenen Systemen mit verschiedenen Datenmodellen, links im JSON-Format, rechts in einer Tabelle.

(a)
```
{ _id: 5, name: "Uwe",
  gehalt: 50000 }
```
```
{ _id: 6, name: "Pia",
  gehalt: {
    betrag: 70000,
    waehrung: "EUR" } }
```

(b)
```
{ _id: 5, name: "Uwe",
  gehalt: 50000 }
```
```
{ _id: 6, name: "Pia",
  salary: 70000 }
```

(c)
```
{ _id: 5, name: "Uwe",
  gehalt: 50000 }
```

| id | name | gehalt |
|---|---|---|
| 77 | Pia | 70000 |

**Abbildung 4.11:** Konflikte auf (a) Datenebene, (b) Metadatenebene und (c) Meta-Metadatenebenene

Beim Umgang mit semistrukturierten Daten werden Metadaten eins mit den Daten und somit verschmelzen die beiden Fälle (a) und (b) in vielen NoSQL-Datenbanken. Wie sich ebendiese Heterogenität lösen lässt, wurde weiter oben in diesem Kapitel diskutiert. In diesem Abschnitt geht es um Fälle wie im Beispiel (c), sprich um die *Überwindung technischer Heterogenität* und der *Datenmodell-Heterogenität*. Eines der in Abschnitt 4.1 präsentierten Ziele der Sprache NotaQL ist es, die Sprache *Datenmodell-überbrückend* nutzen zu können. Um dies zu realisieren, erforschen wir zunächst, welche Voraussetzungen dazu erfüllt sein müssen:

- *Unterstützung verschiedener Datenmodelle* Die in verschiedenen Systemen vorkommenden Datenmodelle und Konzepte (Arrays, Unterdokumente, u. s. w.) müssen von der Sprache unterstützt werden, damit die verschiedenen Systeme vollständig angebunden werden können. Dies ist aufgrund der weiter oben in diesem Kapitel gezeigten Pfadausdrücke, des internen NotaQL-Datenmodells sowie der Unterstützung für benutzerdefinierte Typen gegeben.

- *Anbindung verschiedener Systeme* Ein Anwender sollte beliebige Datenbanksysteme anbinden können. Dazu soll die native API des jeweiligen Systems oder ein DB-Gateway zum Einsatz kommen. Hat ein System ein zu einem anderen unterstützten System äquivalentes Datenmodell, können die Pfadausdrücke und die interne Repräsentation der Datensätze wiederverwendet werden. Andernfalls können spezielle Pfadausdrücke, ein Datensatz-Mapping oder die Erstellung benutzerdefinierter Typen und Funktionen vonnöten sein.

- *Sprachkonstrukte zur Ein- und Ausgabedefinition* Die Sprache NotaQL definiert *was* transformiert werden soll und *welche Gestalt* die Ergebnisse haben sollen. Für systemübergreifende Transformationen sind Sprachkonstrukte vonnöten, um festzulegen *von wo* Daten gelesen und *wohin* das Ergebnis geschrieben werden soll. Wie genau dies definiert wird, hängt vom verwendeten System ab, beispielsweise über einen Dateipfad oder eine Datenbank-ID.

Da die erste Voraussetzung bereits gegeben ist, werden in diesem Abschnitt die beiden anderen Voraussetzungen behandelt. Dazu stellen wir die sogenannten *Engines* vor, welche die Schnittstelle zwischen der NotaQL-Plattform und konkreten Datenbanksystemen bilden.

### Engines

Eine Engine in NotaQL ist eine vom Benutzer entwickelte Anbindung an ein Datenbanksystem, ein Dateiformat oder an einen Dienst. Sie stellt eine Art Wrapper dar, um einen lesenden und schreibenden Zugriff auf die Daten abzubilden und um die Daten in ein internes Format zu überführen. Eine Engine wird definiert über einen Namen und eine Menge von Parametern, wobei ein Teil der Parameter als optional deklariert werden kann. Die beiden wichtigsten zu implementierenden Methoden bei der Erstellung einer Engine sind `evaluate` und `store`, in der Daten gelesen und geschrieben werden. In diesen Methoden kommen typischerweise die nativen API-Methoden des jeweiligen Systems zum Einsatz.

Betrachten wir als erstes Beispiel eine Engine für den Wide-Column-Store HBase [Apab]. Die Engine trägt den Namen `hbase`, sodass sie später in NotaQL-Skripten unter Verwendung dieses Namens zum Einsatz kommen kann. Ein obligatorischer Parameter ist der Name der Tabelle, aus der die Daten gelesen bzw. in die die Daten geschrieben werden sollen. Andere optionale Parameter sind denkbar, z. B. für den Namen einer Standard-Spaltenfamilie, für den Hostnamen des HBase-Masterknotens, den Port, u. s. w. Bei optionalen Parametern ist anzugeben, welchen Wert diese beim Weglassen erhalten sollen. Für die drei genannten Parameter beispielsweise die Spaltenfamilie `default`, aus der Daten gelesen werden bzw. in die Daten geschrieben werden, wenn in Attribut-Mappings keine Spaltenfamilie genannt wird, und den Server `localhost` auf Port `60000`.

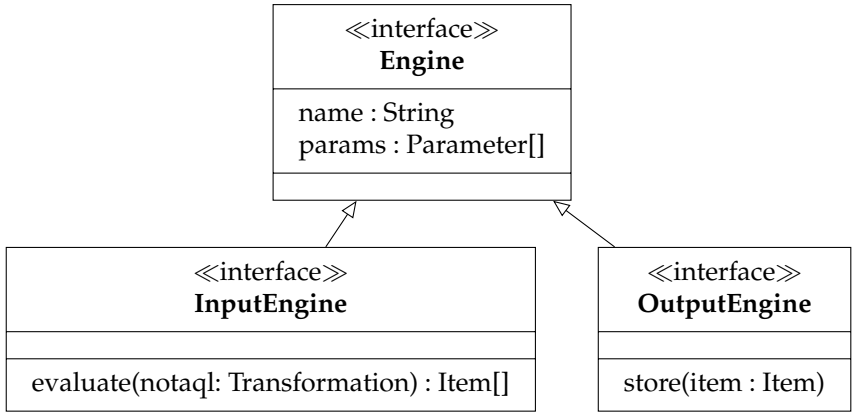



**Abbildung 4.12:** Engine-Schnittstellen

In Abbildung 4.12 ist ein Klassendiagramm der Engine-Schnittstellen dargestellt. In der Regel implementiert eine Engine-Klasse sowohl die Schnittstellen `InputEngine` als auch `OutputEngine`. Ein Engine-Entwickler gibt dazu den Namen und die Parameterliste an und implementiert die beiden Methoden `evaluate` und `store`. Es ist aber auch denkbar, dass nur eine der beiden Schnittstellen implementiert wird, falls eine Engine nur zum

Lesen oder nur zum Schreiben verwendet wird. Eine Engine, welche eine Schnittstelle zu einem Customer-Relationship-Management-System darstellt, soll vielleicht nur lesend auf die Daten zugreifen. Engines, die nur Schreibaktionen ermöglichen, sind beispielsweise Anbindungen zu Ausgabegeräten wie Druckern, Displays, Lautsprechern oder Ähnlichem. Sie können nur zur Ausgabe eines Transformationsergebnisses aber nicht als Eingabe dienen. Datenbanksysteme und Dateien eignen sich üblicherweise sowohl als Ein- als auch aus Ausgabe einer NotaQL-Transformation. Für diese ist eine `evaluate`-Methode zu entwickeln, welche die Datensätze der in der Parameterkonfiguration angegebenen Datenquelle liest und an die NotaQL-Plattform weitergibt, sowie eine `store`-Methode, welches einen gegebenen Ausgabedatensatz in die Datenbank schreibt. Der folgende Pseudocode zeigt eine Beispielimplementierung für eine HBase-Engine:

```
class HBaseEngine
implements InputEngine, OutputEngine {
 name = "hbase"
 params = [ table, family="default",
            host="127.0.0.1", port=60000 ]

 function evaluate(notaql) {
  items = []
  for(row : HBaseAPI.scan(params.table)) {
   item = {}
   for(column : row.getColumns()) {
    item.$(column.name) = column.value
   }
   items.put(item)
  }
  return items
 }

 function store(item) {
  put = {}
  for(attribute : item.getAttributes()) {
   put.add(attribute.name, attribute.value)
  }
  HBaseAPI.put(params.table, put)
 }
}
```

In der `evaluate`-Methode wird die native HBase-API verwendet, um einen Tabellenscan durchzuführen. Die ermittelten Zeilen werden samt aller ihrer Spalten in NotaQL-Objekte im internen Datenmodell (hier: `items`) gewandelt und schließlich ausgegeben. Vor dem Schreiben in der `store`-Methode muss die interne Repräsentation wieder zurückgewandelt werden, im gegebenen Beispiel in ein `Put`-Objekt, welches ein Schreiben in HBase auslöst.

Die Umwandlung in das interne Datenmodell macht die NotaQL-Plattform unabhängig von den verwendeten Systemen, da alle Operationen wie Attribut-Mappings, Selektionen, Gruppierungen und Aggregationen auf

den internen NotaQL-Objekten erfolgen. Zudem werden auf diese Art systemübergreifende Transformationen möglich gemacht. Entwickelt man zusätzlich zur HBase-Engine noch zwei weitere Engines, etwa für MongoDB [Mona] und Redis [Red], sind beliebige Transformationen von HBase zu Redis, von Redis zu MongoDB und so weiter möglich. Bei der Entwicklung von $n$ Engines erhält man durch die Implementierung von $2n$ Methoden insgesamt $2^n$ mögliche Kombinationen, um Daten von einem System zu einem anderen zu transformieren. Dabei ist es nur notwendig, dass die Methoden das interne NotaQL-Datenmodell verstehen und nicht die einzelnen Datenmodelle der anderen Engines.

Bei systemübergreifenden Transformationen erfolgt die *Engine-Spezifikation* gleich zu Beginn eines NotaQL-Skripts. Dazu wird in den Klauseln `IN-ENGINE` und `OUT-ENGINE` jeweils der Name der zu verwendenden Engine sowie die Parameterbelegung formuliert. Abbildung 4.13 zeigt die Syntax zur Engine-Spezifikation.

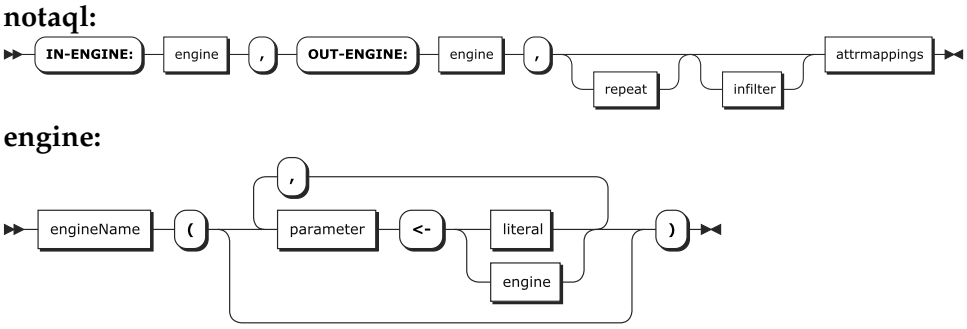



**Abbildung 4.13:** Engine-Spezifikation in einem NotaQL-Skript

Gegeben sei die Web-Anwendung eines sozialen Netzwerks, in welchem Benutzerprofile und Beziehungen zwischen Benutzern in der Graphdatenbank Neo4J [Neo] gespeichert werden und der Key-Value-Store Redis [Red] die Besucherzähler zu jeder Profilwebseite speichert. Das folgende NotaQL-Skript kann dazu eingesetzt werden, um bei Bedarf – beispielsweise einmal am Tag – die Besucherzähler-Werte in die Benutzerdatenbank zu übertragen:

```
IN-ENGINE: redis(database <- 0),
OUT-ENGINE: neo4j(path <- '/data/net'),
OUT._id <- IN._k,
OUT.profilbesucher <- IN._v
```

In der ersten Zeile wird definiert, dass die Redis-Engine verwendet werden soll und dass sich die zu lesenden Daten in der Datenbank mit der ID $0$ befinden. Gleichzeitig wird durch die Wahl der Eingabe-Engine festgelegt, welche Pfadausdrücke in Verbindung mit `IN.` in Attribut-Mappings und der Filterdefiniton gültig sind. Auf die in Redis gespeicherten Schlüssel-Wert-Paare erfolgt im gegebenen Beispiel der Zugriff über `IN._k` (Benutzernummer) und `IN._v` (Anzahl der Profilbesucher). Analog wird in der Spezifikation der Ausgabe-Engine auf die Neo4J-Engine verwiesen, welche den Graph modifiziert, der unter `/data/net` im Dateisystem abgelegt ist. Durch die Festlegung auf eine Graphdatenbank als Ausgabe ist es möglich, Kanten zu erzeugen und Propertywerte zu setzen. Im gezeigten Beispiel kommt nur letzteres zum Einsatz.

### Engine-Evolution

In der Mediator-basierten Datenbank-Middleware *Garlic* [RS97] bedeutet der Begriff *Wrapper-Evolution*, einen Wrapper, welcher auf ein externes Datenbanksystem zugreift, nach und nach weiterzuentwickeln, damit eine erste funktionierende Version bereits früh zum Einsatz kommen kann, und dass fehlende Funktionalität – wie Selektionen und Verbundoperationen – zunächst von der Middleware kompensiert werden. In späteren Versionen übernimmt der Wrapper selbst die jeweiligen Aufgaben, was meist zu einer Performanzsteigerung führt. In NotaQL ist ein analoges Vorgehen bei der Entwicklung von Engines möglich. Dazu fällt zunächst bei Betrachtung der Signatur der `evaluate`-Methode auf, dass sie als Eingabeparameter ein NotaQL-Skript entgegen nimmt (siehe Abbildung 4.12). Dies ermöglicht die Entwicklung „schlauer“ Engines, welche beim Lesen der Eingabedaten das NotaQL-Skript evaluieren. Aufgrund dieser Tatsache heißt die Methode auch `evaluate` und nicht etwa `read`. Unter der sogenannten *Engine-Evolution* versteht man den Prozess, zunächst eine möglichst simple Engine zu entwickeln, welche das übergebene NotaQL-Skript schlicht ignoriert und lediglich die vollständige Datensammlung zurückgibt. Diese Engine kann nun nach und nach bei Bedarf weiterentwickelt werden, um die Performanz von Transformationen zu steigern. Dies ermöglicht schnelle Entwicklungen von Engines und ein frühstmögliches Verwenden. Abbildung 4.14 zeigt den Prozess der Engine-Evolution. Bereits nach der Entwicklung einer schlichten ersten Engine-Version können NotaQL-Skripte definiert und auf dem verwendeten System ausgeführt werden. Da alle Aktionen wie Selektionen und Projektionen von der NotaQL-Plattform und nicht auf dem Datenbanksystem selbst ausgeführt werden, muss sich der Engine-Entwickler darum nicht kümmern. Dennoch gibt es Potential zur Optimierung. Falls ein Filter im NotaQL-Skript definiert ist, kann eine verbesserte Version der Engine einen *Predicate-Pushdown* durchführen, das heißt sie interpretiert die Fiterdefinition und leitet sie an das darunterliegende System weiter. Dadurch wird verhindert, dass unnötige Daten gelesen und danach direkt wieder verworfen werden. Auch ermöglicht dies die Ausnutzung von eventuell im Datenbanksystem vorliegenden Indexen. Ferner kann das übergebene NotaQL-Skript als Hinweis dafür dienen, welche Attribute gelesen werden müssen und welche nicht. Eine frühzeitige Projektion verringert den Leseaufwand und damit die Größe der im Arbeitsspeicher zwischengespeicherten Objekte. Die Engine-Evolution ermöglicht es, Engines nach und nach effizienter zu machen, damit Transformationen schneller durchgeführt werden können. Existierende NotaQL-Skripte können nach der Weiterentwicklung einer Engine unverändert weiterverwendet werden. Das Ergebnis einer Transformation bleibt das gleiche.
Eine Engine-Evolution auf der oben stehenden HBase-Engine kann erfolgen, indem dem Scan-Kommando ein Filter übergeben wird:

```
9  filter = toFilter(notaql.inFilterPredicate)
10 for(row : HBaseAPI.scan(params.table, filter)) ...
```

Die `toFilter`-Methode ist zu implementieren. Sie wandelt ein NotaQL-Eingabefilterprädikat in eine HBase-Filterdefinition. Diese Umwandlung erfolgt üblicherweise rekursiv, da Klammern, Negationen, Konjunktionen

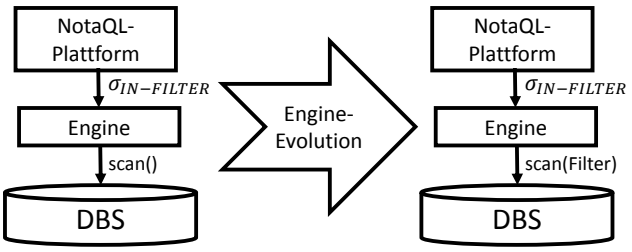



**Abbildung 4.14:** Engine-Evolution: Die rechte Engine führt den Filter direkt auf dem Datenbanksystem aus

und Disjunktionen von boolschen Ausdrücken wieder Teilprädikate beinhalten, welche umgewandelt werden müssen. Nach der Engine-Evolution werden Transformationen, welche aus HBase lesen, deutlich beschleunigt, da das Filter-Objekt jedem HRegionServer – das sind die Rechner, auf welchen die Tabellendaten gespeichert werden – übergeben wird und jeder Server den Filter lokal auswerten kann. Die Folge ist, dass keine Zeilen über das Netzwerk transportiert werden, welche das gegebene Prädikat nicht erfüllen. Jedoch ist auch ohne die Entwicklung Engine-spezifischer Filter ein lokales Auswerten der Prädikate möglich, was die Netzwerklast so gering wie möglich hält. Dazu muss die NotaQL-Plattform selbst als verteilte Anwendung implementiert sein. Trotzdem ist das Filtern innerhalb der Engine immer die beste Lösung, da weniger Objekte von einer Festplatte gelesen werden müssen und da nur so Indexe ausgenutzt werden können.

Es ist auch denkbar, dass eine Engine nur einen Teil des Prädikats auswertet und eine Übermenge der Datensätze zurückliefert. Ein vollständiges Filtern, also das Entfernen der zu viel geladenen Datensätze, erfolgt durch die NotaQL-Plattform. Ein Beispiel ist ein System, welches nur String-Containment-Queries und keine Gleichheit unterstützt. Die Engine würde in diesem Falle alle Datensätze ausgeben, bei denen die im Eingabefilter vorkommende Zeichenkette im gegebenen Attribut irgendwo vorkommt. Die NotaQL-Plattform würde anschließend auf String-Gleichheit überprüfen. Bei konjunktiven Prädikaten, also solche, die aus Und-verknüpften Teilprädikaten bestehen, kann eine Engine ohne Weiteres solche Teilprädikate ignorieren, die es zu interpretieren nicht in der Lage ist. Bei der Verwendung von (benutzerdefinierten) Funktionen im Eingabefilter kann das endgültige Filtern in der Regel ohnehin erst durch die NotaQL-Plattform erfolgen.

### Join-Engines

Bei der Verwendung von NoSQL-Datenbanken gilt es, Joins zu vermeiden. Da die Systeme Datenmodellkonzepte wie Arrays, Kanten und verschachtelte Dokumente zu bieten haben, wird empfohlen, diese Konzepte gegenüber Modellierungen mittels Fremdschlüsseln vorzuziehen. Aus diesem Grund bieten die meisten NoSQL-Datenbankmanagementsysteme keine, oder wenn überhaupt nur eine ineffiziente Möglichkeit, Verbundoperationen durchzuführen. Meist ist zur Ausführung von Joins die Entwicklung eines speziellen Programms vonnöten, beispielsweise unter der Verwendung von MapReduce [DG04] oder Spark [Zah+10]. Alternativ kann eine höhere Sprache wie Pig [Ols+08], Hive [Thu+09] oder die MongoDB Aggregation Pipeline [Mona] eingesetzt werden. NotaQL bietet keine direkte

Unterstützung für Joins, ermöglicht aber die Entwicklung von *Join-Engines* auf zweierlei Arten: Zum einen zur Ausführung von lokalen Joins auf einem speziellen Eingabesystem, zum anderen mittels einer generischen *delegierenden Engine*, wodurch globale systemübergreifende Joins ermöglicht werden. Die beiden Alternativen werden in diesem und im nächsten Abschnitt detaillierter beschrieben.

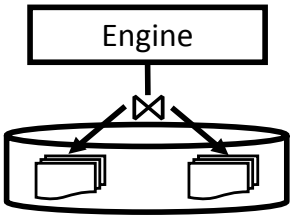


**Abbildung 4.15:** Join-Engine

Die performanteste Möglichkeit für Verbundoperationen in NotaQL ist es, den Verbund direkt auf dem Eingabedatenbanksystem auszuführen. Dies ist natürlich nur dann möglich, wenn das gegebene System überhaupt Joins unterstützt. Weiterhin gilt, dass nur lokale Joins ausgeführt werden können, also nur zwischen den verschiedenen Datenquellen aus ein und demselben Datenbanksystem (siehe Abbildung 4.15), z. B. zwischen zwei MongoDB-Kollektionen.

Im nun folgenden Beispiel möchten wir eine gegebene Engine – eine JDBC-Engine, die als Zugriff auf relationale Datenbanken dient – zu einer Join-Engine erweitern. Auf die ursprüngliche Engine wird wie folgt in NotaQL verwiesen:

```
IN-ENGINE: jdbc(url <- 'jdbc:db2://localhost:5021
                /MyDB:user=dbadm;password=dbadm',
                table <- 'personen')
```

In der JDBC-Engine selbst wird eine Verbindung zur in der URL angegebenen Datenbank hergestellt und eine `SELECT`-Anfrage auf der gegebenen Tabelle ausgeführt. Eine optimierte Implementierung würde den im NotaQL-Skript vorkommenden Eingabefilter in ein `WHERE`-Prädikat überführen und in die Anfrage einbauen. Die JDBC-Engine kann zu einer Join-Engine erweitert werden und gleichzeitig abwärtskompatibel bleiben, indem zwei optionale Engine-Parameter eingeführt werden, `table2` für die zu verbindende Tabelle und `joinPredicate` für das Verbundsprädikat. Sind die beiden Parameter gesetzt, kann die Engine den Verbund direkt auf der Datenbank durchführen:

```
SELECT * FROM %table JOIN %table2 ON %joinPredicate
```

Das hier verwendete Verbundprädikat bleibt – anders als die Eingabefilterdefinition – von der NotaQL-Platform uninterpretiert. Es wird über den Engine-Parameter als String übergeben. Die folgende Transformation verbindet zwei relationale Tabellen anhand eines Fremdschlüssels und schreibt das Ergebnis in Form von Schlüssel-Wert-Paaren in eine Redis-Datenbank. Die Schlüssel sind Orte und die Werte jeweils eine Liste von IDs derjenigen Personen, die in diesem Ort arbeiten:

```
IN-ENGINE: jdbc(url <- 'jdbc:db2://localhost:5021
                /MyDB:user=dbadm;password=dbadm',
                table <- 'personen',
                table2 <- 'firmen', joinPredicate <-
                'personen.firma=firmen.id'),
OUT-ENGINE: redis(database <- 0),
OUT._k <- IN.firmen.ort,
```

```
OUT._v <- LIST(IN.personen.id)
```

Es ist zu beachten, dass die Join-Engine die Namen der ursprünglichen Tabellen in die internen NotaQL-Objekte aufnehmen muss, da sonst Konflikte bei gleichnamigen Spalten auftreten können. Im gegebenen Beispiel wurde dazu die gängige Dot-Notation verwendet. Das gezeigte Beispiel kann nur genau zwei Tabellen miteinander verbinden. Denkbar sind aber auch Implementierungen, in denen beliebig lange Listen von Tabellen und Prädikaten an die Engine übergeben werden.

Der Vorteil einer Join-Engine ist, dass der Verbund lokal auf dem Datenbanksystem ausgeführt wird und dadurch die maximale Performanz erreicht werden kann. Das verwendete Datenbanksystem kümmert sich automatisch um eine optimale Ausführung des Verbunds. Es wählt die bestmögliche Verbundreihenfolge und -strategie (Nested-Loop-Join, Hash-Join, ...) und kann von existierenden Indexen und Datenbankstatistiken profitieren. Der Engine-Entwickler muss sich darüber keine Gedanken machen. Der Nachteil einer Join-Engine ist jedoch, dass sie Verbundoperationen nur lokal und nicht systemübergreifend ausführen kann. Des Weiteren muss für jedes System eine eigene Join-Engine implementiert werden. Da manche Systeme nativ gar keine Joins unterstützen, kann für solche keine entsprechende Engine entwickelt werden.

### Delegierende Engines

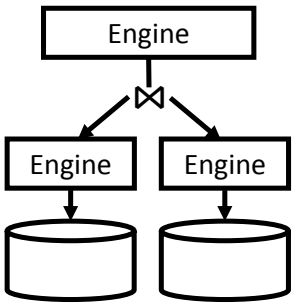



**Abbildung 4.16:** Delegierende Engine

Unter einer *delegierenden Engine* versteht man eine NotaQL-Engine, die wiederum andere existierende Engines aufruft. Wie in Abbildung 4.16 zu sehen, interagiert sie nicht direkt mit einem Datenbanksystem, sondern delegiert den Befehl zum Lesen der Daten an eine oder mehrere andere Engines. Die aufgerufenen Engines müssen die *InputEngine*- oder *StreamingEngine*-Schnittstelle (siehe Abschnitt 4.9.2) implementieren. Da die delegierende Engine auch selbst mindestens eine der beiden genannten Schnittstellen implementiert, ist sogar eine mehrstufige Delegation möglich. Mit diesem Ansatz lässt sich eine *generische Join-Engine* implementieren, die die im vorherigen Absatz erläuterten Probleme löst [Bru16]. Die im Folgenden vorgestellte generische Join-Engine nimmt drei Parameter entgegen, und zwar zwei Engine-Spezifikationen `left` und `right` sowie ein Prädikat `joinPredicate`. Dadurch wird es möglich, Joins zwischen beliebigen Systemen – auch systemübergreifend – durchzuführen. Folgendes NotaQL-Skript führt die gleiche Transformation wie oben durch, allerdings liegen die Eingabedaten nun in zwei verschiedenen NoSQL-Datenbanksystemen:

```
IN-ENGINE: join(left <- hbase(table <- 'personen'),
  right <- mongodb(db<-'test', collection<-'firmen'),
  joinPredicate <- 'left.firma=right._id'),
OUT-ENGINE: redis(database <- 0),
```

```
OUT._k <- IN.right.ort,
OUT._v <- LIST(IN.left._r)
```

Die in Redis geschriebenen Ausgabedatensätze sind Schlüssel-Wert-Paare mit einem Ort als Schlüssel, der aus einer MongoDB-Kollektion gelesen wurde, und einer Liste von Row-IDs der in diesem Ort arbeitenden Personen aus der HBase-Tabelle als Wert. Anders als im zuvor gezeigten Beispiel müssen die Attributzugriffe sowie das Verbundprädikat die Alias-Präfixe `left` und `right` statt der originalen Tabellen- oder Kollektionennamen enthalten. Auf die tatsächlichen Namen hat die delegierende Engine typischerweise keinen Zugriff mehr, denn sie stehen lediglich in Engine-spezifischen Parametern. Des Weiteren wird es durch diese generischen Präfixe möglich, dass Datensammlungen in den beiden aufgerufenen Systemen den gleichen Namen tragen.

Der große Vorteil der delegierenden Join-Engine ist die universelle Anwendbarkeit. Der Nachteil ist jedoch, dass sie die Verbundoperation meist nur sehr inperformant ausführen kann. Beide Datenquellen werden vollständig eingelesen. Beim Predicate-Pushdown, also beim Durchreichen des in der NotaQL-Transformation spezifizierten Eingabefilters, welcher in die für Unter-Engines spezifischen Teile zerlegt werden muss, sind die zu verbindenden Teilmengen jedoch in der Regel deutlich kleiner, was eine spürbare Beschleunigung der Transformation zur Folge hat. Wenn jedoch wie im gezeigten Beispiel kein Eingabefilter zum Einsatz kommt, also zwei Datenquellen vollständig miteinander verbunden werden, ist die einzige Lösung das Abholen der kompletten Datenbestände beider Quellen und die anschließende Ausführung des Joins.

Die delegierende Join-Engine kann intern einen Nested-Loop-Join, Hash-Join, Sort-Merge-Join oder Ähnliches durchführen [HR05, Kapitel 11]. Bei stark verlustbehafteten Joins können Bloom-Filter [Blo70] zum Einsatz kommen, um Datensätze einer Quelle von vornherein zu überspringen, wenn diese ohnehin keinen Join-Partner in der anderen Datenquelle finden. Es ist denkbar, dass die Verbundstrategie automatisch gewählt wird oder aber über einen Engine-Parameter angegeben werden kann. Des Weiteren könnte man über einen weiteren Parameter die Art des Verbunds angeben, z. B. ein linker äußerer Verbund oder ein natürlicher Verbund.

Die Engine-Evolution kann auch beim Verwenden von Verbundoperationen eingesetzt werden. Zum einen kann die Funktionalität einer delegierenden oder lokalen Join-Engine im Laufe der Zeit erweitert werden, um die Performanz zu steigern, beispielsweise mittels eines Predicate-Pushdowns. Zum anderen kann aber auch eine zunächst delegierende Engine später von einer lokalen Join-Engine abgelöst werden. Da es erstere ermöglicht, jede Engine als Eingabe für einen Verbund zu verwenden, ist kein zusätzlicher Entwicklungsaufwand erforderlich. Ist es bei häufiger Ausführung oder bei steigenden Datenmengen vonnöten, eine Optimierung vorzunehmen, ist dies durch die Entwicklung einer speziellen Engine möglich.

Die in diesem Abschnitt präsentierte generische Join-Engine ist nur ein Beispiel für eine delegierende Engine. Andere Anwendungen sind beispielsweise Engines zur Ausführung eines kartesischen Produkts oder einer Vereinigung mehrerer Datenquellen.

## 4.9 Datenstrom-Transformationen

*Datenströme* [KS09] unterscheiden sich von Datenbanken dadurch, dass die darin verwalteten Daten nicht an einem festen Ort persistent gespeichert sind und von dort aus jederzeit abgefragt werden können. Stattdessen sind die Daten *flüchtig*. Sie bilden einen zeitlich unbegrenzten Strom und können gewisse Aktionen auslösen oder über einen gewissen Zeitraum beobachtet und analysiert werden. Stellt ein Benutzer eine Anfrage an ein Datenbanksystem, erhält er typischerweise in kurzer Zeit eine Ergebnismenge als Antwort. In einem *Datenstrommanagementsystem* werden sogenannte *kontinuierliche Anfragen* [Ter+92] gespeichert. Genau wie Eingabedaten ständig einströmen, produziert eine einmal definierte Anfrage ständig Ausgaben in Echtzeit. Somit ist das Ergebnis einer kontinuierlichen Anfrage wieder ein Strom.

NotaQL ist für kontinuierliche Anfragen gut geeignet, da NotaQL-Transformationen Ähnlichkeiten zu materialisierten Sichten [GM95] haben. Während klassische NotaQL-Datenbanktransformationen solche materialisierte Sichten darstellen, die bei Bedarf aktualisiert werden, erzeugen oder modifizieren kontinuierliche Anfragen auf Datenströmen sofort bei einströmenden Änderungen ihr Ergebnis. In [JMS95] werden materialisierte Sichten um sogenannte *Chroniken* erweitert, welche Sequenzen von Tupeln – also quasi Datenströme – sind. Es wird ein Operator definiert, um aus einer Chronik den Zustand einer materialisierten Sicht abzuleiten. Einige Systeme ermöglichen die Definition kontinuierlicher Anfragen in einer Datenbanksprache. Diese Ansätze werden sowohl von alten Systemen wie *Tapestry* [Ter+92] unter Verwendung einer zu SQL ähnlichen Sprache verfolgt, als auch von aktuellen NoSQL-Datenbanken wie *MongoDB* [Mona].[4] Die Idee dahinter ist, eine Anfrage immer dann neu auszuführen, wenn neue Datensätze in die Datenbank eingefügt wurden. Da die Eingabe der Anfrage immer nur der eine neu hinzugefügte Datensatz ist, sind bei Verfolgung dieses Ansatzes keine Aggregationsanfragen möglich.

### Kontinuierliche Anfragen

Die folgenden zwei Beispiele stellen typische Anfragen auf Datenströmen dar:

- $q_1$: Ein Datenstrom enthält die aktuell auf der Plattform Twitter[5] erstellten Beiträge (Tweets). Die Anfrage $q_1$ soll einen Strom erzeugen, welcher nur die deutschsprachigen Tweets enthält.
- $q_2$: Ein Wettersensor produziert einen kontinuierlichen Datenstrom von Temperatur- und anderen Wetterdaten. Die Anfrage $q_2$ soll zu jeder vollen Stunde die Durchschnittstemperatur der vergangenen Stunde ausgeben.

Die beiden gezeigten Anfragen $q_1$ und $q_2$ transformieren jeweils einen Eingabestrom in einen Ausgabestrom. Streaming-Plattformen wie Apache Flink

[4] MongoDB verwendet dazu sogenannte *Tailable Cursors*. Mehr dazu folgt später in diesem Abschnitt.

[5] http://www.twitter.com

[Apa15b] oder Spark Streaming [Zah+12] bieten Operatoren, um Selektionen, Projektionen, Joins, Aggregationen und mehr direkt auf Strömen zu definieren. Die Anfrage $q_1$ könnte in Spark Streaming wie folgt definiert werden:

```
twitterStream.filter(_.lang.equals("de"))
```

Für die zweite Anfrage ist deutlich mehr Programmieraufwand nötig, da zum einen ein *Fenster* auf dem Datenstrom definiert werden muss und zum anderen eine Funktion zur Berechnung der Durchschnittswerte. Andere Systeme verwenden Sprachen, die auf SQL basieren und erweitern diese um Fensterdefinitionen. Folgendes Beispiel zeigt die Anfragen $q_1$ und $q_2$ in der Sprache TelegraphCQ [Cha+03b]:

```
SELECT * FROM twitterStream WHERE lang = 'de';
SELECT AVG(temperature) FROM
 weatherStream [RANGE BY '1 hour' SLIDE BY '1 hour'];
```

Um Streams in NotaQL zu unterstützen, orientieren wir uns an einem Ansatz aus der *Continous Query Language* (CQL) [ABW06]. Die Sprache bietet keine direkten Strom-zu-Strom-Operationen, sondern lediglich Operationen, um ein Datenstrom in eine Relation zu wandeln und umgekehrt. Die Sprache CQL basiert auf SQL, mit welcher alle übrigen Operationen direkt auf den Relationen ausgeführt werden können. Dieser Ansatz hat mehrere Vorteile, welche sich auf NotaQL-Transformationen übertragen lassen: Erstens muss keine komplett neue Sprache kreiert und gelernt werden. Zweitens kann die komplette Funktionalität aus NotaQL ausgenutzt und auf Datenströme übertragen werden. Dazu wandelt man zunächst den Strom in eine virtuelle Relation, transformiert diese wie gewohnt und wandelt das Ergebnis schließlich wieder in einen Strom zurück. Drittens erlaubt dieser Ansatz auch das kombinierte Verwenden von Datenströmen und Datenbanken. CQL unterstützt beispielsweise Verbundoperationen zwischen einem Datenstrom und einer relationalen Tabelle. In NotaQL ermöglicht es dieser Ansatz, dass sowohl Eingabe als auch Ausgabe einer Transformation ein Datenstrom sein kann, oder aber nur eines oder – wie gehabt – keines davon.

Die Wandlung eines Stroms in eine virtuelle Relation erfolgt in CQL ähnlich zur oben gezeigten TelegraphCQ-Anfrage mittels einer *Fensterdefintion* in der `FROM`-Klausel. Über die Größe des Fensters (in Sekunden, Minuten oder Stunden) wird angegeben, aus welchen Tupeln die virtuelle Relation bestehen soll, nämlich aus denen, die innerhalb der angegebenen Zeit im Eingabestrom erschienen sind. Alternativ kann ein Fenster auch über eine Anzahl von Datensätzen definiert werden und somit die $n$ zuletzt gesehenen Objekte enthalten. In CQL muss eine virtuelle Relation nach Anwendung von SQL-Operationen wieder in einen Datenstrom zurückgewandelt werden. Dazu kann entweder ein *Insert-Stream*, ein *Delete-Stream* oder ein *Relationen-Stream* definiert werden. Die ersten beiden genannten enthalten ein Tupel aus der virtuellen Tabelle, sobald es in dieser erstmalig erscheint bzw. aus der Relation entfernt wurde. Der Relationen-Stream liefert in vordefinierten Zeitintervallen die komplette virtuelle Relation. Für die oben definierte Anfrage $q_1$ wird ein Fenster unbegrenzter Größe und ein Insert-Stream verwendet:

```
SELECT Istream(*) FROM
 twitterStream [Range Unbounded] WHERE lang = 'de';
```

Wegen des zeitlich unbegrenzten Fensters beinhaltet die virtuelle Relation alle seit Erstellung der Anfrage entdeckten Tweets. Das in der Anfrage definierte `WHERE`-Prädikat ist eine Relation-zu-Relation-Transformation und reduziert die Anzahl der Datensätze in der virtuellen Relation. Der `Istream`-Operator liefert zu einem Zeitpunkt $t$ diejenigen Objekte zurück, die zum Zeitpunkt $t-1$ noch nicht in der gefilterten Relation vorlagen, also üblicherweise unmittelbar nach dem Erscheinen im Eingabestrom.

Ein weiterer Weg, die virtuelle Relation in einen Stream zurück zu wandeln ist es, in regelmäßigen Abständen die komplette Relation – also alle aktuell enthaltenen Tupel – auszugeben. Ähnlich zur Fensterdefintion wird dieses Verhalten über die Angabe einer Zeit $T$ oder einer Anzahl von Elementen $n$, die im Eingabestrom gesehen werden, definiert. Dazu kommt in CQL die `SLIDE`-Klausel zum Einsatz. Immer wenn $T$ Sekunden vergangen sind – oder wenn $n$ Elemente im Strom gesehen wurden –, erfolgt eine Ausgabe. Die oben stehende Anfrage $q_2$ lässt sich in CQL definieren, indem wir ein einstündiges Fenster auf dem Wetterdaten-Strom definieren, anschließend den Durchschnitt der darin enthaltenen Temperaturwerte bilden und die daraus entstehende Relation – welche nur aus einem Tupel besteht, da kein `GROUP BY` verwendet wird – jede Stunde im Ausgabestrom ausgeben:

```
SELECT Rstream(AVG(temperature)) FROM
 weatherStream [Range 1 hour Slide 1 hour];
```

Durch die gegebene Fensterdefinition in der `FROM`-Klausel wird jede Stunde eine Relation aus den in der vergangenen Stunde im Datenstrom gesehenen Wetterdatenwerten erzeugt. Das gebildete Fenster wird ein *tumbling Window* genannt, da `Range` und `Slide` die gleichen Zeitangaben haben. Jeder gesehene Temperaturwert geht in genau eine Durchschnittsberechnung ein, es gibt keine Überlappungen in den einzelnen Fenstern. Bei einer größer gewählten `Slide`-Zeit gingen ein Teil der Messwerte verloren, bei einer kürzeren Zeitangabe würden sich die dadurch entstehenden *sliding Windows* überlappen. Über ein sliding Window könnte beispielsweise alle fünf Minuten die Durchschnittstemperatur der letzten Stunde ausgegeben werden. Ein `Slide`-Wert von $n = 1$ (ohne Zeiteinheit) würde ebendiesen Durchschnittswert kontinuierlich ausgeben; immer dann, wenn ein Eintrag im Eingabestrom erfasst wird.

### Streaming-Engines

Um NotaQL auf Datenströmen nutzbar zu machen, führen wir *Streaming-Engines* [Emd16] ein. Diese speziellen Engines liefern genau wie die im vorherigen Abschnitt präsentierten Engines für Datenbanken und Dateien die Eingabedaten für eine Transformation, jedoch nicht als gesamte Einheit, sondern in Form eines kontinuierlichen Datenstroms. Beim Schreiben der Transformationsergebnisse verhält sich eine Streaming-Engine jedoch absolut äquivalent zu den klassischen Engines, da diese ohnehin immer nur einen Datensatz nach dem anderen schreiben. Während Datenstrom-Frameworks wie Apache Flink [Apa15b] und Sprachen wie CQL [ABW06]

vor allem dazu gedacht sind, dass in Anwendungsprogrammen auf Datenströme analytisch zugegriffen werden kann, werden NotaQL-Transformation üblicherweise dazu eingesetzt, um Ergebnisse persistent abzuspeichern. Denn anders als CQL ist NotaQL keine Anfragesprache. Stattdessen verfolgt sie auf Datenströmen einen ähnlichen Ansatz wie *Apache Flume* [Hof13]. Mit Flume können Datenströme analysiert, transformiert und Aggregationen auf diesen ausgeführt werden. Das Ergebnis wird in HDFS [Apaa] oder HBase [Apab] geschrieben. Basierend auf Floom existieren SQL-ähnliche Sprachen [Kim11], um Analysen zu definieren.

In NotaQL erfolgt die Definition, wie ein Ergebnis ausgegeben werden soll, wie gehabt mit der `store`-Methode. In dieser muss nicht zwangsläufig eine Einfügeoperation in eine Datenbank erfolgen. Die Ausgabedatensätze können auch als Nachrichten über einen Kanal wie Twitter oder Kafka [Apac] verschickt werden, sodass die Ausgabe wieder als ein Datenstrom angesehen werden kann. Betrachten wir als Beispiel die `store`-Methode einer Output-Engine für Twitter-Nachrichtenströme:

```
1  class TwitterEngine implements OutputEngine {
2   name = "twitter"
3   params = [ consumerKey, consumerSecret,
4              accessToken, accessTokenSecret ]
5   twitter = null
6
7   function TwitterEngine() {
8    twitter = TwitterFactory.getInstance(params)
9   }
10
11  function store(item) {
12   twitter.updateStatus(item.text)
13  }
14 }
```

Die Twitter-Engine kann nach dieser Implementierung direkt als Ausgabe-Engine in NotaQL-Skripten zum Einsatz kommen. Die Benutzerdaten werden über die vier Parameter übergeben, welche in der Konstruktorfunktion beim Initialisieren der Klasse zum Login genutzt werden. Das Attribut `OUT.text` wird im Transformationsskript auf den zu schreibenden Text gesetzt. Damit lässt sich nun beispielsweise eine Transformation definieren, welche die Anzahl der Personendatensätze in einer Dokumentendatenbank zählt und diese Zahl „twittert“:

```
IN-ENGINE: mongodb(db<-'test', collection<-'pers'),
OUT-ENGINE: twitter(consumerKey<-'xyz', ...),
OUT.text <- COUNT() + ' Personen'
```

Diese Transformation ist jedoch genau genommen keine Datenstrom-Transformation. Anders als eine kontinuierliche Query wird diese nur ein einziges Mal ausgeführt. Man müsste die Transformation in Form eines Cronjobs in regelmäßigen Zeitabständen ausführen lassen, um ein Datenstromverhalten zu simulieren.
Eine NotaQL-Transformation wird dann zur *Datenstromtransformation*, wenn eine *Streaming-Engine* als Eingabe verwendet wird. Das Klassendiagramm

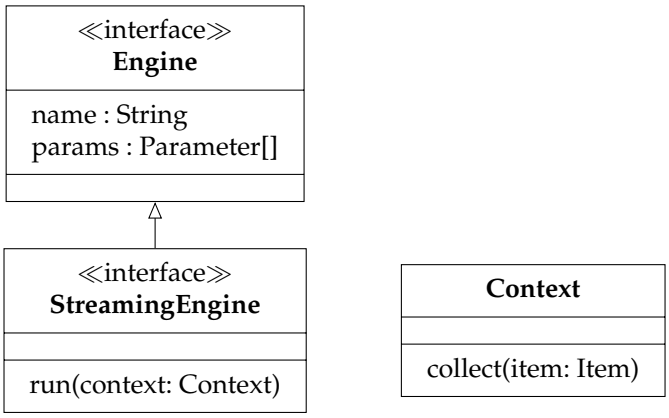



**Abbildung 4.17:** Streaming-Engine-Schnittstelle

in Abbildung 4.17 zeigt, dass dazu die Methode `run` implementiert werden muss. Anders als bei der `evaluate`-Methode von Eingabe-Engines ist ihr Rückgabetyp `void`. Die ermittelten Eingabedatensätze werden also nicht als Funktionsresultat zurückgegeben, sondern über eine in einem *Kontext*-Objekt befindliche Methode `collect`. Innerhalb der `run`-Methode findet sich in StreamingEngines üblicherweise eine Dauerschleife wieder, die eine NotaQL-Transformation zu einer kontinuierlichen Transformation macht. Die `run`-Methode sollte erst terminieren, wenn der Benutzer die Transformation wieder löscht oder pausiert. Betrachten wir eine beispielhafte Implementierung der `run`-Methode für die oben gezeigte Twitter-Engine:

```
1  class TwitterEngine
2  implements StreamingEngine, OutputEngine {
3   // Parameter, Konstuktor und store() wie oben
4
5   function run(context) {
6    twitter.addListener(function onStatus(status) {
7     context.collect(convertStatus(status))
8    })
9    while(true) { }
10  }
11 }
```

Die auf der Twitter-Verbindung eingerichtete Listener-Funktion wird immer dann aufgerufen, wenn eine Tweet-Nachricht entdeckt wird. Hier kann ein Predicate-Pushdown des eventuell im NotaQL-Skript verwendeten Eingabefilters erfolgen, um mittels eines Parameters der Twitter-API nur diejenigen Tweets abzurufen, die das Filterkriterium erfüllen. Eine zu implementierende `convertStatus`-Methode wandelt ein Status-Objekt der Twitter-API in ein NotaQL-Objekt um. Über die `collect`-Methode wird schließlich der NotaQL-Plattform dieses Objekt übermittelt.

Die zu Beginn dieses Abschnittes genannten Twitter-Beispielanfrage $q_1$ lässt sich nun wie folgt in NotaQL ausdrücken:

```
IN-ENGINE: twitter(consumerKey<-'xyz', ...),
OUT-ENGINE: hbase(table <- 'de_tweets'),
IN-FILTER: IN.lang = 'de',
```

```
OUT._r <- IN.id,
OUT.$(IN.*.name()) <- IN.@
```

Da die Twitter-Engine als Eingabe verwendet wurde, handelt es sich bei der gezeigten Transformation um eine *kontinuierliche Transformation*. Sie terminiert nicht automatisch sondern schreibt kontinuierlich Daten in das Ausgabesystem – in diesem Fall in eine HBase-Tabelle.

Es fällt auf, dass sich die Ausführungssemantik von der der klassischen NotaQL-Transformationen unterscheidet. Statt eine komplette Datenmenge zu verarbeiten und dabei eventuell Gruppen zu bilden, wird immer nur ein Datensatz auf einmal verarbeitet und unmittelbar in die Ausgabe geschrieben. Die Verwendung von Aggregatfunktionen ergibt jedoch nur dann einen Sinn, wenn wir mehrere Datensätze auf einmal betrachten können. Dazu ist – genau wie in anderen Datenstrommanagementsystemen und Sprachen – eine *Fensterdefinition* vonnöten. Analog zu CQL [ABW06] verwenden wir die vier Parameter `range`, `rows`, `partition` und `slide`, welche zusätzlich zu den Engine-spezifischen Parametern in der Eingabe-Engine-Spezifikation angegeben werden können:

- `range`: Fenstergröße in Millisekunden, Sekunden, Minuten, Stunden oder Tagen; z. B. `'30s'`.
- `rows`: Fenstergröße in Anzahl der Datensätze; Standardwert: `1`.
- `partition`: Ein Attributname, nach dem partitioniert wird; nur in Verbindung mit `rows`.
- `slide`: Aktualisierungsintervall (zeitlich oder Anzahl der Datensätze); Standardwert: `1`.

Werden die Parameter `range` und `rows` in Kombination verwendet, enthält das Fenster die im gegeben Zeitraum gesehenen Datensätze, allerdings nur maximal so viele wie angegeben. Lässt man beides weg, besteht das gebildete Fenster nur aus dem zuletzt gesehen Datensatz. Wird wie im obigen NotaQL-Beispiel für die Anfrage $q_1$ auch die `slide`-Angabe weggelassen, erhält die NotaQL-Plattform den Fensterinhalt bei jedem neu ankommenden Objekt. Bei der Angabe einer `partition` gilt die in `rows` angegebene Anzahl pro distinktem Wert im angegebenen Attribut. Ein Beispiel: Das Fenster auf einem Twitter-Stream mit `rows<-1` und `partition<-'user'` beinhaltet die zuletzt platzierte Nachricht von jedem einzelnen Benutzer, unabhängig davon, wie alt diese ist. Ein zusätzlicher `range`-Wert kann das maximale Alter begrenzen.

Gehen wir zur Betrachtung der Anfrage $q_2$ nun davon aus, dass eine Wetter-Engine definiert wurde, welche einen kontinuierlichen Datenstrom über die Messwerte eines Wettersensors liefert. Die NotaQL-Transformation, um die Durchschnittstemperatur der vergangenen Stunde jede Stunde bei Twitter auszusenden, lautet wie folgt:

```
IN-ENGINE: weather(range <- '1h', slide <- '1h'),
OUT-ENGINE: twitter(consumerKey<-'xyz', ...),
OUT.text <- 'Es ist ' + AVG(IN.temperature) + '°C'
```

Es ist durchaus möglich, Engines zu entwickeln, welche sowohl die Schnittstellen `InputEngine` als auch `StreamingEngine` implementieren. Wird

eine solche Engine als Eingabe verwendet, agiert sie als normale Eingabe-Engine, es sei denn, mindestens einer der drei Parameter `range`, `rows` und `slide` wurde gesetzt. In diesem Fall wird eine Datenstrom-Transformation ausgeführt. Die Weiterentwicklung einer klassischen Engine um eine Streaming-Funktionalität macht es möglich, Datenbanksysteme und Dateien kontinuierlich zu überwachen und als Ströme zu verarbeiten. Zur Realisierung der `run`-Methode können in Datenbanksystemen Trigger verwendet werden, welche die NotaQL-Plattform über eingefügte, geänderte oder gelöschte Tupel informieren. Einige Systeme wie PostgreSQL [Pos] oder Redis [Red] bieten eine Publish-Subscribe-Funktionalität. Damit kann eine Anwendung – in diesem Falle eine NotaQL-Engine – einen *Kanal* abonnieren und auf diesem die Datenbankänderungen abhören. Andere Systeme ermöglichen kontinuierliche Datenbankanfragen. Beispielsweise kann für eine MongoDB-Anfrage [Monb] ein *Tailable Cursor* geöffnet werden. Anders als bei klassischen Cursors werden diese beim Abschluss einer Anfrage nicht geschlossen, sondern weiterhin offen gehalten. Die Anwendung erhält somit alle in Zukunft kommenden Datensätze, die auf die Anfrage passen, unmittelbar zum Zeitpunkt ihres Entstehens.

Streaming-Engines können innerhalb von delegierenden Engines aufgerufen werden. Dadurch wird beispielsweise ein Verbund zwischen zwei Datenströmen oder zwischen einem Strom und einer Datenbank möglich gemacht. Kommt in einer delegierenden Engine mindestens eine Streaming-Engine zum Einsatz, handelt es sich beim vorliegenden NotaQL-Skript um eine kontinuierliche Anfrage. Die gelesenen Daten, welche keine Datenströme darstellen, werden durch den Aufruf der `evaluate`-Methode abgerufen und in einem Cache zwischengelagert, um beispielsweise bei Eintreffen eines Strom-Elements mit diesem eine Verbundoperation durchzuführen. Folgendes abschließende NotaQL-Skript twittert basierend auf zwei Datenströmen – weltweit platzierte Kameras mit Personenerkennung sowie Wettersensoren – und einer Personen-Kollektion jeden Tag die Namen der Personen, die sich im Moment in einer warmen Gegend aufhalten:

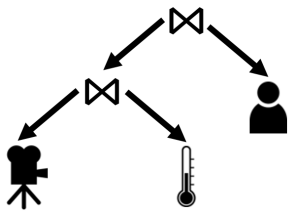


**Abbildung 4.18:** Datenstrom-Joins

```
IN-ENGINE: join(left <- join(
  left <- camera(rows<-1, partition<-'person',
                 range<-'24h', slide<-'24h'),
  right <- weather(rows<-1, partition<-'location',
                   range<-'24h', slide<-'24h'),
  joinPredicate <- 'left.location=right.location'),
 right <- mongodb(db<-'test', collection<-'pers'),
 joinPredicate <- 'left.right.person=right._id'),
OUT-ENGINE: twitter(consumerKey<-'xyz', ...),
IN-FILTER: IN.left.right.temperature >= 30,
OUT.text <- IMPLODE(IN.right.name, ',') + ' ist warm'
```

Der Verbundbaum zu diesem Skript ist graphisch in Abbildung 4.18 dargestellt. Zuerst wird der Kamera-Stream mit dem Wetterdaten-Stream verbunden, um jeder Personenbeobachtung die am betreffenden Standort aktuelle Temperatur anzufügen. Da sich die beiden verbundenen Ströme alle vierundzwanzig Stunden aktualisieren, wird auch der Verbund dementsprechend oft berechnet. Der für beide Eingabeströmen gesetzte `partition`-Parameter sorgt dafür, dass zu jedem Standort und zu jeder Person höchstens eine Beobachtung vorliegt – und zwar jeweils die aktuellste. Mit dem Parameter `range <- '24h'` wird verhindert, dass es sich um einen veralteten Wetter- oder Personenerfassungswert handelt. Das Ergebnis wird mit einer MongoDB-Kollektion verbunden, um von jeder beobachteten Person den Namen zu finden und auszugeben. Bei der Ausführung wird die Selektion, um nur die hohen Temperaturwerte zu berücksichtigen, entweder am Ende durchgeführt oder über einen Predicate-Pushdown an die Wetterdaten-Engine weitergeleitet.

Zusammenfassend lässt sich für Datenstrom-Transformationen sagen, dass in NotaQL Datenströme sowohl als Eingabe als auch als Ausgabe angebunden werden können. Neben den weiter vorne in dieser Arbeit vorgestellten Transformationen, um persistent gespeichert Daten von einem System in ein anderes zu übertragen, ist es mittels Streaming-Engines möglich, Datenströme zu transformieren, um entweder wiederum einen Datenstrom zu erzeugen oder Ergebnisse kontinuierlich in eine Datenbank oder eine Datei zu speichern.

## 4.10 Transformationsketten

NotaQL ist eine erweiterbare und sehr mächtige Sprache. Wegen des schlichten Ausführungsmodells ist es jedoch nicht immer möglich, beliebig komplexe Transformationen in einem einzigen NotaQL-Skript zu formulieren. Stattdessen sind *Ketten von Transformationen* vonnöten, die nacheinander geschaltet und ausgeführt werden. Eine Transformation inmitten einer solchen Kette erhält als Eingabe das Ergebnis einer vorangegangenen Transformation.

Betrachten wir als Beispiel eine MongoDB-Kollektion, welche ähnlich wie in vorherigen Beispielen Personendaten der folgenden Form speichert:

```
{ _id: 1, name: "Ute",
  arbeit: { firma: "IBM", ort: "Boeblingen" } }
```

Möchte man basierend auf dieser Kollektion zu jedem Ort die durchschnittliche Anzahl von Mitarbeitern berechnen, welche die Firmen, die in dem jeweiligen Ort ansässig sind, dort beschäftigen, ist dies nicht in einem einzigen NotaQL-Skript formulierbar. Auch in SQL wäre zur Beantwortung dieser Frage eine verschachtelte Anfrage mit zwei `GROUP BY`-Klauseln vonnöten. Die innere Anfrage berechnet die Anzahl der Mitarbeiter pro Firma und Ort, die äußere schließlich die Durchschnittszahlen pro Ort:

```
SELECT ort, AVG(mitarbeiter) FROM
  (SELECT arbeit.firma, arbeit.ort AS ort,
   COUNT(*) AS mitarbeiter
   FROM pers GROUP BY arbeit.firma, arbeit.ort) temp
```

```
GROUP BY ort;
```

In NotaQL gehen wir äquivalent vor. Die erste Teil-Transformation bildet Gruppen anhand des zusammengesetzten `arbeit`-Feldes:

```
IN-ENGINE: mongodb(db<-'test', collection<-'pers'),
OUT-ENGINE: mongodb(db<-'test', collection<-'temp'),
OUT._id <- IN.arbeit,
OUT.mitarbeiter <- COUNT()
```

Die Zwischenergebnisse in der Kollektion `temp` haben die folgende Form:

```
{ _id: { firma: "IBM", ort: "Boeblingen" },
  mitarbeiter: 1111 }
```

Basierend auf den Zwischenergebnissen berechnet folgendes Skript das Endergebnis:

```
IN-ENGINE: mongodb(db<-'test', collection<-'temp'),
OUT-ENGINE: mongodb(db<-'test', collection<-'orte'),
OUT._id <- IN._id.ort,
OUT.mitarbeiter <- AVG(IN.mitarbeiter)
```

Die MongoDB-Kollektion `temp` wurde lediglich zum Ablegen der Zwischenergebnisse verwendet und kann nach dem Abschluss wieder gelöscht werden. Vorteile dieser temporären Kollektion sind, dass man sich die Zwischenergebnisse zu Debugging-Zwecken anschauen kann und dass das Ergebnis der ersten Teilberechnung nicht verloren geht, wenn die zweite Berechnung fehlschlagen sollte. Nachteile sind jedoch, dass durch das materialisierte Zwischenspeichern viele Schreib- und Lesezugriffe erfolgen. Transformationsketten könnten effizienter ausgeführt werden, wenn Zwischenergebnisse nicht persistent geschrieben werden. Genau dies war eines der größten Kritikpunkte an MapReduce [DG04] und wurde durch neuartige Systeme wie Spark [Zah+10] dadurch gelöst, dass diese kein starres Programmiermodell haben, sondern beliebige Operationsketten erlauben. Eine simple Lösung in NotaQL ist die Verwendung einer In-Memory-Datenbank zur Speicherung der Zwischenergebnisse. Allerdings hat auch dies mehrere Nachteile. Zum einen kann es passieren, dass das Datenmodell der In-Memory-Datenbank nicht zu dem der Zwischenergebnisse passt. In diesem Fall ist eine aufwändige Überführung zwischen den verschiedenen Datenmodellen vonnöten. Zum anderen – und diesen Nachteil haben wir auch bei der zuvor gezeigten MongoDB-Lösung – kann nicht garantiert werden, dass die Zwischenergebnisse auf demjenigen Rechner abgelegt werden, auf dem sie berechnet wurden. In verteilten Umgebungen ist stets ein Datentransport über das Netzwerk vonnöten.

Zur Lösung des Problems bietet NotaQL eine *virtuelle Engine* [Emd16], welche als Ein- und Ausgabe-Engine verwendet werden kann. Da sie nur virtuell ist und keine Daten tatsächlich auf ein Medium schreibt oder sie von dort liest, darf sie immer nur paarweise innerhalb von Transformationsketten vorkommen. Eine Transformationskette darf weder mit einer virtuellen Engine starten noch mit einer solchen enden. Als Parameter wird ein beliebiger *Virtual-Engine-ID*-Wert gesetzt, welcher verschiedene Transformationen mit der gleichen ID miteinander verknüpft:

```
OUT-ENGINE: virtual(id<-'firmen'),
```

Die von dieser Transformation lesende Transformation erhält dementsprechend die gleiche Spezifikation für die Eingabe-Engine. Ein NotaQL-Skript, in welchem eine virtuelle Engine als Ein- oder Ausgabe verwendet wird, kann nicht ohne weiteres eigenständig ausgeführt werden. Das Starten ist nur für eine komplette *Transformationsgruppe* möglich. Diese Gruppe besteht aus denjenigen Transformationen, die über identische Virtual-Engine-IDs miteinander verknüpft sind. Eine Gruppe ist nur dann ausführbar, wenn jede verwendete ID in mindestens einer Eingabe- und mindestens einer Ausgabe-Engine vorkommt. Abbildung 4.19 zeigt die oben beschriebene Kette von Transformationen unter Verwendung einer virtuellen Engine.

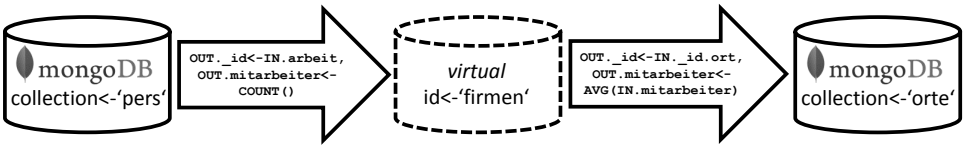



**Abbildung 4.19:** Transformationskette mit einer virtuellen Engine

Durch die virtuellen Engines bleibt zum einen die Ausführungssemantik von NotaQL-Transformationen simpel und es erlaubt zum anderen komplexe Transformationen ohne zusätzliche Zwischenspeicherung. Bei der Ausführung werden alle zu einer Gruppe gehörenden Transformationen zusammengeschaltet und als eine Einheit ausgeführt. Dadurch bleiben Zwischenergebnisse im Arbeitsspeicher, sie verbleiben im internen NotaQL-Datenmodell und der Ausführungsplattform werden Möglichkeiten geschaffen, um den Gesamtprozess zu optimieren. In Kapitel 5.2 wird beschrieben, wie Ketten, in denen virtuelle Engines zum Einsatz kommen, auf Plattformen wie Spark [Zah+10] oder Flink [Apa15b] ausgeführt werden können.

Mit virtuellen Engines ist es möglich, neben einfachen Transformationsketten auch komplexere Abläufe mittels *gerichteter azyklischer Graphen* zu modellieren. Dazu erlauben wir, dass ein und die selbe Virtual-Engine-ID in mehr als einer Eingabe-Engine und in mehr als einer Ausgabe-Engine Verwendung finden kann. Abbildung 4.20 veranschaulicht drei solcher Fälle. Die Zylinder stehen in der Abbildung für Engines, die gestrichelten Zylinder für die virtuellen. Die NotaQL-Transformationen sind als Pfeile dargestellt.

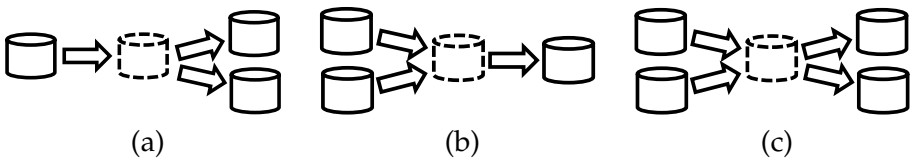



**Abbildung 4.20:** Einsatzmöglichkeiten virtueller Engines

In Fall (a) dient die Ausgabe einer Transformation als Eingabe zweier nachfolgender Transformationen. Dies erspart Rechenaufwand, da das gemeinsam genutzte Zwischenergebnis nur ein einziges mal berechnet werden muss. Eine geschickte Ausführungsart ist es, das Zwischenergebnis in einem flüchtigen Cache abzulegen, sodass alle nachfolgenden Transformationen darauf zugreifen können. In Fall (b) erhält die Transformation auf der rechten Seite zwei Eingabemengen. Die Zwischenergebnisse sind in

diesem Fall eine Vereinigung der Ergebnisse der beiden linken Transformationen. Fall (c) stellt eine Kombination aus den beiden anderen Fällen dar. Die beiden linken Transformationen schreiben ihre Ergebnisse in eine gemeinsame virtuelle Ausgabe, welche wiederum von zwei nachfolgenden Transformationen als Eingabe gelesen wird. Die Fälle können beliebig erweitert werden und virtuelle Eninges als Ein- und Ausgabe beliebig vieler anderer Transformationen dienen. Ziel ist es, ein äquivalentes Verhalten zu simulieren, als ob statt der virtuellen Engine eine echte Engine zum Einsatz käme, welche die Daten persistent ablegt. Wäre diese Zwischenablage beispielsweise eine CSV-Datei, würden in den Fällen (a) und (c) zwei NotaQL-Skripte auf ein und dieselbe Datei lesend zugreifen. In den Fällen (b) und (c) würden zwei Transformation ihre Ergebniszeilen in eine Datei zusammenlegen. Bei der Verwendung von virtuellen Engines muss sich der Anwender um die Optimierung und um das Datenmodell der Zwischenablage keine Gedanken machen. Während eine klassische Engine festlegt, welche Pfadausdrücke bei Verwendung dieser Engine erlaubt sind (`OUT._k`, `IN._e_._id`, ...), können bei Benutzung von virtuellen Engines beliebige Ausdrücke verwendet werden. Die finale Umwandlung in das jeweilige Zieldatenmodell erfolgt durch die letzte NotaQL-Transformation einer Kette bzw. eines Ausführungsgraphen, in welcher eine klassische Ausgabe-Engine eingesetzt werden muss.

Werden innerhalb einer Transformationsgruppe virtuelle Engines in Verbindung mit mindestens einer *Streaming-Engine* eingesetzt, darf es in dieser Gruppe keine klassischen Eingabe-Engines geben. Andernfalls erhielte eine Transformation als Eingabe die Vereinigung zwischen einer Menge von Datensätzen und einem Datenstrom, was nicht sinnvoll nutzbar ist. Daher müssen – mit Ausnahme virtueller Engines – entweder alle Eingabe-Engines Streaming-Engines sein oder keine. Im ersteren Fall handelt es sich um eine kontinuierliche Transformation, im zweiten Fall um eine Stapelverarbeitung, die abgeschlossen ist, sobald alle Eingabedaten verarbeitet wurden.

## 4.11 Zusammenfassung

Zu Beginn dieses Kapitels wurden die Ziele der Sprache für NoSQL-Datenbanken festgelegt und anschließend die Sprache NotaQL präsentiert. Als Abschluss untersuchen wir, ob und wie die Ziele umgesetzt wurden.

Die *Schlichtheit* wurde in NotaQL dadurch erreicht, dass sich Transformationsskripte in kompakter Form definieren lassen. Stößt die Sprache an ihr Limit, können zusätzliche Funktionalitäten in benutzerdefinierte Funktionen ausgelagert werden. Im Wesentlichen besteht eine NotaQL-Transformation aus einer Menge von Attribut-Mappings, durch die festgelegt wird, wie eine Eingabezeile in eine Ausgabezeile transformiert wird. Vor diesem Transformationsschritt kann ein Filter gewisse Eingabezeilen aussortieren, nach dem Schritt können Aggregationsfunktionen genutzt werden, um aus den für ein und die selbe Zelle vorliegenden Werten die Endwerte der Ausgabezellen zu berechnen. Dieses *einfache Ausführungsmodell* erleichtert nicht nur die Arbeit mit der Sprache, es bietet auch hohes Potenzial für verschiedene Ausführungsarten, inkrementelle Berechnungen und Optimierungen,

wie in den nächsten Kapiteln noch dargestellt wird. Komplexe Transformationen können mit Hilfe von Transformationsketten und virtuellen Engines durchgeführt werden. NotaQL unterstützt ist eine *Vielfalt von Datenmodellen*. Zwar existiert ein eigenes Datenmodell, welches auf dem JSON-Datenmodell basiert, jedoch ermöglichen interne Abbildungen auf andere Datenmodelle die Anwendbarkeit auf einer Vielzahl von Systemen. Über das Konzept der Engines ist es möglich, Daten von einem System in ein anderes System zu transformieren und dabei *Datenmodelle zu überbrücken*. Der Benutzer wählt beim Formulieren eines NotaQL-Skripts stets die zum Eingabe- und Ausgabesystem passende Sprachsystax. Bei einfachen Key-Value-Stores sind dies lediglich Schlüssel- und Wertezugriffe, bei komplexen Graphdatenbanken können Kanten traversiert und erzeugt werden. Da in NoSQL-Datenbanken meist kein Schema existiert und verschiedene Datensätze verschiedene Attribute haben können, verfügt NotaQL über Sprachkonstrukte wie den Stern- oder Indirektionsoperator, um über die Menge der Attribute zu iterieren, Metadaten auszulesen und Daten in Metadaten zu wandeln. Dies macht NotaQL zu einer *schemaflexiblen* Sprache. Eine *Erweiterbarkeit* ist über Engines, benutzerdefinierte Typen und Funktionen möglich. Im nächsten Kapitel stellen wir verschiedene Ausführungsarten vor, um NotaQL-Transformationen mittels Datenverarbeitungsframeworks *verteilt* und *inkrementell* auszuführen.

# 5 Ausführung von NotaQL-Transformationen

Ein Anwender beschreibt beim Formulieren eines NotaQL-Skripts, *was* transformiert werden soll und nicht *wie*. Um das *Wie* zu klären, betrachten wir in diesem Kapitel verschiedene Ausführungsarten und analysieren deren Vor- und Nachteile. Wir beschränken uns dabei zunächst auf nicht-inkrementelle Transformationen, also solche, die die vollständige Eingabemenge lesen und das Ergebnis einer vorherigen Berechnung nicht mit einbeziehen. Im nächsten Kapitel betrachten wir unterschiedliche inkrementelle Vorgehensweisen, welche prinzipiell von der Ausführungsart, wie sie in diesem Kapitel beschrieben werden, unabhängig sind.

Das oberste Ziel einer jeden Ausführungsart ist die korrekte Ausführung eines gegebenen Transformationsskripts. Obwohl sich die nun vorgestellten Ansätze teilweise sehr stark unterscheiden, muss für jeden Ansatz gelten, dass eine gegebene Transformationsvorschrift bei gleichen Eingabedaten stets die gleichen Ausgabedaten produziert. Dies ermöglicht einen einfachen Wechsel von einer Ausführungsart zur anderen. Es soll nicht Ziel in diesem Kapitel sein, den besten Ansatz zu finden, welcher alle anderen Ansätze in jeglicher Hinsicht dominiert. Zum einen gibt es diesen besten Ansatz nicht, zum anderen werden ständig neue Technologien, Frameworks und Sprachen entwickelt und verbessert, welche neue Möglichkeiten bringen. Stattdessen präsentieren wir in diesem Kapitel Ideen zur Umsetzung, welche sich auf existierende und kommende Technologien übertragen lassen.

Die Gemeinsamkeit aller Ausführungsarten ist, dass sie äquivalente Ergebnisse liefern. Die Unterschiede können vielfältiger Natur sein. Bei unserer Bewertung der Ansätze betrachten wir folgende Eigenschaften:

- *Geschwindigkeit*: Die Zeit, bis eine Transformation abgeschlossen ist, kann bei einer Ausführungsart kürzer sein als bei einer anderen. Oft hängt es jedoch von der Komplexität eines NotaQL-Skripts ab, welcher von zwei Ansätzen der schnellere ist.

- *Verteiltheit*: Berechnungen können zentral auf einem Rechner oder verteilt erfolgen. In letzterem Fall kann eine eventuelle verteilte Datenspeicherung ausgenutzt werden und es ist ein horizontales Skalieren durch Hinzufügen weiterer Rechenknoten möglich.
- *Mächtigkeit*: Manche Ausführungsarten unterstützen nicht alle Sprachkonstrukte und können deshalb nicht universell eingesetzt werden.
- *Enginge-Unterstützung*: Manche Ansätze eignen sich nur für spezielle Ein- und Ausgabe-Engines.
- *Inkrementelle Berechenbarkeit*: Möglicherweise werden nicht alle im nächsten Kapitel vorgestellten Ansätze für inkrementelle Berechnungen unterstützt.

Abbildung 5.1 stellt die logische Ausführung eines NotaQL-Skripts dar. Zunächst durchlaufen die Eingabedatensätze den Eingabefilter und anschließend erfolgt die Anwendung der Attribut-Mappings. Dabei können Aggregatfunktionen, die auf Listen-wertigen Attributen aufgerufen werden, bereits ausgeführt werden, alle anderen werden in diesem Schritt ignoriert. Die daraus entstehenden *Ausgabefragmente* werden anschließend nach ihrer ID gruppiert – z. B. bei MongoDB nach der `_id`, bei HBase nach der Row-ID. Aus jeder gebildeten Gruppe wird schließlich durch Aufruf der im NotaQL-Skript definierten Aggregatfunktionen ein Ausgabedatensatz erzeugt.

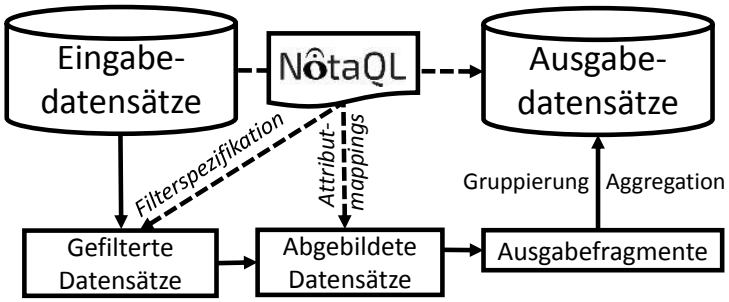



**Abbildung 5.1:** Logische Ausführung eines NotaQL-Skripts

Das gezeigte Vorgehen ist jedoch nur logischer Natur. Es hilft dem Anwender beim Formulieren und Verstehen von NotaQL-Skripten. Die genaue Implementierung der einzelnen Transformationsschritte hängt von der konkreten Ausführungsart ab.

Im Nachfolgenden stellen wir verschiedene Ansätze vor. Zunächst Ansätze, die auf einem Daten- oder Graphverarbeitungsframework basieren (MapReduce, Spark, Flink, ...), danach solche, bei denen NotaQL in eine andere Sprache abgebildet wird (SQL, Pig, Cypher,...), und schließlich beschäftigt sich der letzte Ansatz in diesem Kapitel mit virtuellen Transformationen. Letztere werden erst dann durchgeführt, wenn ein Lesen auf den Ergebnissen erfolgt – also wie bei einer Datenbanksicht.

## 5.1 MapReduce

Das MapReduce-Programmiermodell [DG04] ist schlicht und passt zur Ausführungssemantik von NotaQL-Skripten. Der Ansatz ist *datensatzorientiert*, das heißt, man betrachtet nicht die Eingabemenge als gesamte Einheit sondern immer nur einen Datensatz nach dem anderen. Zunächst wird eine Selektion und Umformung vorgenommen und schließlich Gruppen von Datensätzen gebildet, um auf diesen Aggregatfunktionen auszuführen und die Ausgabedatensätze zu genieren. Das MapReduce-Framework kümmert sich um das Einlesen der Datensätze, die Gruppierungen zwischen den beiden Phasen *Map* und *Reduce* und um das Schreiben der Ergebnisse. Zudem beherrscht es Fehlerbehandlungsmechanismen und Monitoring-Komponenten zur Überwachung einer Transformation. Abbildung 5.2 zeigt die Map- und Reduce-Funktion zur generischen Ausführung von NotaQL-Transformationen.

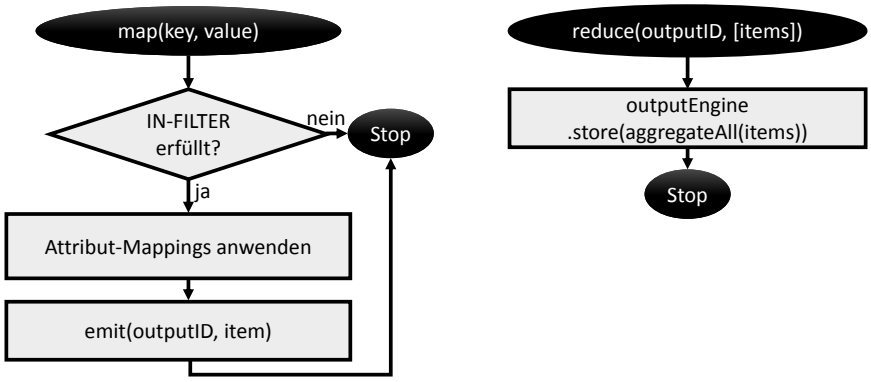



**Abbildung 5.2:** Map- und Reduce-Funktionen

**Map** Die Eingabe der Map-Funktion ist ein Datensatz, also z. B. ein Schlüssel-Wert-Paar, ein Dokument oder eine Zeile aus einer Tabelle oder Datei. Bei einem gegebenen Eingabefilter kann ein Datensatz sofort aussortiert werden, wenn er das Filterkriterium nicht erfüllt. In diesem Falle, ist die Arbeit der Map-Funktion für diesen Datensatz ohne Ausgabe beendet und es kann mit dem nächsten Datensatz fortgefahren werden. Andernfalls erfolgt eine Transformation des Elements gemäß der gegebenen Transformationsvorschrift. Dabei werden *Ausgabefragmente* erzeugt, welche alle Attribute der Ausgabeobjekte besitzen, jedoch wurde noch keine Aggregation durchgeführt. Die Ausgabe der Map-Funktion sind Schlüssel-Wert-Paare, wobei der Schlüssel den Attributswert des gesetzten Ausgabeschlüssels (`OUT._k`, `OUT._id`, `OUT._r`, ...) beinhaltet und der Wert die übrigen Attribute des Ausgabefragments.

**Reduce** Nach der Gruppierung anhand des Schlüssels, welche durch das MapReduce-Framework erfolgt, liegt einer Reduce-Funktion ein bestimmter Schlüssel sowie eine Liste von Ausgabefragmenten vor. Die Aufgabe dieser Funktion ist nun, aus den Fragmenten einen Ausgabedatensatz zu erzeugen. Die beim Zusammenfügen durch Werte-Kollisionen entstehenden Listen-wertigen Attribute werden gemäß der im gegebenen NotaQL-Skript definierten Aggregatfunktionen zu atomaren Attributswerten kombiniert. Bei NotaQL-Skripten ohne Aggregatfunktion darf nur ein Ausgabefragment vorliegen. In diesem Fall kann als Optimierung die Reduce-Phase sogar weggelassen werden.

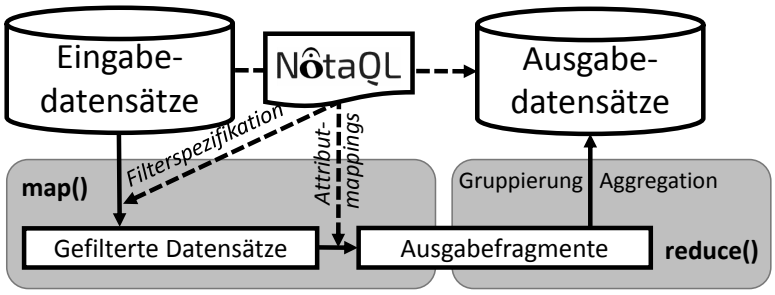



**Abbildung 5.3:** Ausführung eines NotaQL-Skripts mittels MapReduce

Wie in Abbildung 5.3 zu sehen, teilt sich die Ausführung einer NotaQL-Transformation in zwei Teile auf. Der erste Teil ist in der Map-Funktion implementiert und beinhaltet das Filtern sowie das Erstellen der Ausgabefragmente. Die Anwendung der Aggregatfunktionen auf den zusammengehörenden Ausgabefragmenten ist in der Reduce-Funktion implementiert.

Um die Netzwerklast zu verringern und damit die Ausführung von MapReduce-Jobs zu beschleunigen, bietet MapReduce die Möglichkeit zur Entwicklung einer *Combine-Funktion*. Mittels dieser Funktion werden Zwischenergebnisse, die auf einem Rechner nach der Verarbeitung der Eingabedaten durch die Map-Funktion lokal vorliegen, zu einem einzigen Zwischenergebnis zusammengefasst, bevor dieses an die zuständigen Reduce-Rechner weitergegeben wird. Es ist möglich, die Combine-Funktion von der Reduce-Funktion abzuleiten, sofern letztere aus einer Initialisierung, einer Akkumulation in einer Schleife und einer anschließenden Finalisierung besteht [Liu+14]. Dies ist bei der NotaQL-Reduce-Funktion der Fall. Da Aggregatfunktionen in NotaQL jedoch alle diese Schritte auf einmal übernehmen und nicht nur die reine Akkumulation, lässt sich ein generischer Combiner nur dann anwenden, wenn die Aggregatfunktion zusammen mit dem zugrundeliegenden Datentypen einen Monoiden bildet. Das ist für assoziative Funktionen wie `SUM`, `COUNT`, `MIN` und `MAX` der Fall. Liegen lediglich ebensolche Funktionen vor, kann eine generische Combine-Funktion die in Abbildung 5.2 dargestellte `aggregateAll`-Funktion durchführen und das Zwischenergebnis ausgeben. Die finale Aggregation findet wie gehabt in der Reduce-Funktion statt.

### Einbindung der NotaQL-Engines

Bei der NotaQL-Realisierung mittels MapReduce ist es neben der Implementierung der Map- und Reduce-Funktion auch vonnöten, sich um das Lesen der Eingabedaten und Schreiben der Ausgabedaten zu kümmern. Bei der Verwendung der freien MapReduce-Implementierung *Apache Hadoop* [Apaa] erfolgt die Eingabe und Ausgabe mit der Implementierung von `InputFormat`- und `OutputFormat`-Klassen. Da es für viele Datenbanksysteme und Dateiformate fertige Hadoop-Anbindungen gibt, kann auf diese zurückgegriffen werden. Um jedoch auch weitere Anbindungen zu unterstützen und um es Anwendern zu ermöglichen, – wie in Kapitel 4.8 beschrieben – eigene NotaQL-Engines zu entwickeln, bedarf es ein generisches Eingabe- und Ausgabeformat.

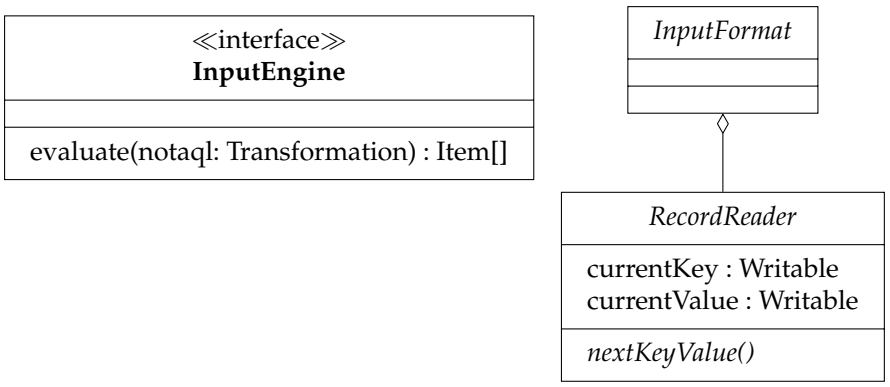



**Abbildung 5.4:** Eingabe-Klassen in NotaQL und Hadoop

Abbildung 5.4 zeigt auf der linken Seite die Schnittstellenbeschreibung einer NotaQL-Eingabe-Engine. Auf der rechten Seite sind die abstrakten Hadoop-Klassen dargestellt, welche zum Lesen der Eingabedaten eines MapReduce-Jobs dienen. Zur Implementierung eines generischen NotaQL-Eingabeformats werden zwei Klassen benötigt: das `NotaQLInputFormat` und der `NotaQLRecordReader`. In letzterem erfolgt das eigentliche datensatzorientierte Lesen der Daten. Beim Initialisieren des Record-Readers wird eine Instanz der im gegebenen NotaQL-Skript angegebenen Eingabe-Engine erzeugt und die entsprechende `evaluate`-Methode aufgerufen. Dieser Aufruf stößt ein Lesen der Eingabedaten an und liefert einen Cursor über eine Liste von Items zurück, welcher bei jedem `nextKeyValue`-Aufruf einen Datensatz zurückliefert. Hadoop verlangt zwar als Map-Eingabe Schlüssel-Wert-Paare, jedoch kann der Schlüssel einfach leer gelassen werden und der Wert das vollständige Objekt enthalten. Auf diese Art muss keine Unterscheidung in Abhängigkeit der Eingabe-Engine erfolgen, welches Attribut als Schlüssel extrahiert werden muss.
Das Schreiben der Ausgabedaten einer Transformation ist leicht umzusetzen, da sowohl die NotaQL-Ausgabe-Engines als auch der Hadoop-RecordWriter Methoden zum *datensatzorientierten Speichern* anbieten. Wie in Abbildung 5.5 zu sehen, muss dazu lediglich der Aufruf der `write`-Methode in der `RecordWriter`-Klasse an die die `store`-Methode der im NotaQL-Skript spezifizierten Ausgabe-Engine umgeleitet werden.

In den beiden gezeigten UML-Diagrammen kommt auf NotaQL-Seite der Datentyp `Item` zum Einsatz. Dieser Typ stellt eine Superklasse der in

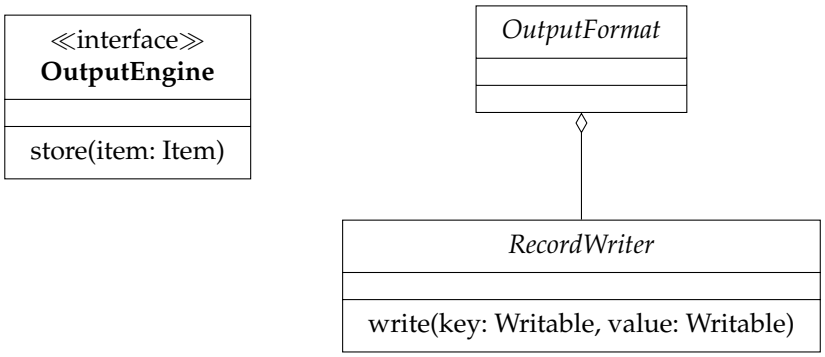



**Abbildung 5.5:** Ausgabe-Klassen in NotaQL und Hadoop

Kapitel 4.6 vorgestellten NotaQL-Typen sowie der benutzerdefinierten Typen dar. Damit er im Hadoop-Framework eingesetzt werden kann, sind Implementierungen der in der `Writeable`-Schnittstelle verlangten Methoden `readFields` und `write` vonnöten. Dies ist für die sechs NotaQL-Typen Number, String, Object, Array, Boolean und Null durch einfache Umwandlung in Byte-Arrays schnell gelöst. Benutzerdefinierten Typen können jedoch auf einer Hadoop-basierten NotaQL-Ausführungsplattform nur dann eingesetzt werden, wenn diese ebenfalls die `Writable`-Schnittstelle implementieren oder wenn eine Abbildung auf einen eingebauten NotaQL-Typen (z. B. Object) existiert. Letzteres ist bei der Verwendung von Graphdatenbanken der Fall. Die dort vorkommenden Kanten sind intern vom Object-Typ, in welchem die Zielknoten-ID, die Richtungsangabe, Label und Properties in Attributen gehalten werden. Die Kantenobjekte werden in der Eingabe-Engine erzeugt und können anschließend an das Hadoop-Framework übergeben werden. Die Ausgabedaten aus der Reduce-Phase, in welchen ebendiese Kantenobjekte vorkommen, werden schließlich von der Ausgabe-Engine in die Graphdatenbank geschrieben, um zwei Knoten miteinander zu verbinden.

### Optimierungen

Die Verwendung von Join-Engines sowie delegierenden Engines (siehe Kapitel 4.8) stellt auf der MapReduce-Plattform kein Problem dar, da die Daten aus den verschiedenen Quellen bereits innerhalb der Engine-Methode `evaluate` miteinander verbunden werden und die Map-Funktion die bereits verbundenen Daten als Eingabe erhält. Jedoch lassen sich diese speziellen Engines nur sehr schlecht parallelisieren. Während einfache Eingabe-Engines auf jedem Rechner im MapReduce-Cluster diejenigen Datensätze der Map-Funktion übergeben, die auf ebendiesem Rechner lokal gespeichert sind, ist es bei Verbundsdatensätzen wahrscheinlich, dass die beiden Join-Partner von unterschiedlichen Maschinen stammen. Das heißt, dass für ein Großteil der Datensätze aus der einen Join-Datenquelle der Join-Partner von einem anderen Rechner geladen werden muss. Dies ist mit sehr viel Kommunikationsaufwand verbunden und verlangsamt die Berechnung. Eine mögliche Lösung ist die Implementierung des Joins in MapReduce. Eine speziell zur Ausführung auf dem MapReduce-Framework

implementierte delegierende Engine sorgt nicht nur für das einfache Lesen der Daten, sondern fungiert auch selbst als ein eigenständiger MapReduce-Job, welcher vor den Job geschaltet wird, der die eigentliche NotaQL-Transformation durchführt. Dadurch wird ein gegebenes Skript als eine Kette von zwei MapReduce-Jobs ausgeführt. Zur Implementierung von Joins existieren mehrere verschiedene Ansätze [Lee+11]. Wenn eine Datenquelle sehr klein ist, bietet es sich an, deren Daten auf alle Rechner zu replizieren und den Join bereits auf Map-Seite durchzuführen. Bei größeren Datenmengen werden beide Datenquellen in der Map-Funktion in Schlüssel-Wert-Paare aufgeteilt, wobei der Schlüssel das Verbundattribut ist und der Wert der komplette Datensatz. Der eigentliche Verbund findet innerhalb der Reduce-Funktion statt. Bei stark verlustbehafteten Verbünden kann ein Bloom-Filter [Blo70; Bru16] eingesetzt werden, um diejenigen Datensätze bereits in der Map-Phase herauszufiltern, die später ohnehin keinen Join-Partner finden würden.

Weitere Optimierungen sind mittels speziellen *Partitionierern* möglich. Während die Map-Funktion typischerweise auf demjenigen Rechner aufgerufen wird, auf dem der zu verarbeitende Datensatz gespeichert ist, bestimmt für die Reduce-Phase ein Partitionierer, welcher Rechner sich um welche Schlüssel kümmert. Der Standard-Partitionierer in Hadoop ist der *Hash-Partitionierer*. Er nimmt einen Schlüssel entgegen – also beispielsweise die Row-ID einer in HBase zu schreibenden Zeile – und erzeugt aus diesem einen Hash-Wert zwischen $0$ und $N - 1$, wobei $N$ die Anzahl der *Reducer* ist. Diese Zahl wird üblicherweise auf etwas leicht geringeres oder höheres als die Anzahl der Rechner im Cluster gesetzt, je nachdem wie gleichmäßig die Werte auf die verschiedenen Schlüssel verteilt sind. Nachdem der jeweilige Reducer das Zusammenfügen der Ausgabefragmente abgeschlossen hat, schreibt er den kombinierten Ausgabedatensatz unter Verwendung der Ausgabe-Engine in die Zieldatenbank. Ist letztere eine verteilte Datenbank, wäre es geschickt, wenn sich immer genau derjenige Rechner um die Reduce-Ausführung eines Datensatzes kümmert, auf welchem das Speichern ebendieses Datensatzes erfolgen soll. Bei der Verwendung des Hash-Partitionierers ist dies jedoch so gut wie nie der Fall. Das zu schreibende Objekt muss mit hoher Wahrscheinlichkeit zu einem anderen Rechner transferiert werden. Falls bekannt ist, welcher Rechner für die Speicherung welcher Datensätze zuständig ist, kann ein spezieller Partitionierer entwickelt werden, der sich ebendieser Information bedient und somit ein geschicktes Partitionieren vornimmt. Am einfachsten ist dies, wenn das Datenbanksystem *Bereichpartitionierung* einsetzt. Ein Partitionierer für MongoDB muss beispielsweise die Bereichsgrenzen der einzelnen Chunks vom Konfigurationsserver laden und darauf achten, dass während der Transformation durch Balancierungsvorgänge keine Änderungen daran vorgenommen werden. Andernfalls ist ein Nachladen vonnöten. Für HBase gibt es einen fertigen *HRegion-Partitionierer*, der die Daten an den Rechner mit dem zuständigen Schlüsselbereich weiterleitet. Um einen allgemeinen Partitionierer zu entwickeln, welcher universell für alle Ausgabe-Engines eingesetzt werden kann, ist eine enge Kopplung mit dem jeweiligen System vonnöten. Diese kann mittels Ergänzung der `OutputEngine`-Schnittstelle um eine Methode `getPartition` hergestellt werden, sodass der universelle Partitionierer ebendiese Methodenaufrufe an die Engine-Klasse delegiert. Bleibt die Methode unimplementiert, wird eine Hash-Partitionierung

vorgenommen.

### Bewertung

MapReduce ist hervorragend zur Ausführung von NotaQL-Skripten geeignet. Das liegt daran, dass beide Ansätze die gleiche Ausführungssemantik haben. Eine NotaQL-Transformation (ohne Verbundoperation) kann immer als exakt ein MapReduce-Job ausgeführt werden. Lediglich bei Transformationsketten sind auch Ketten von MapReduce-Jobs vonnöten. Dies hat jedoch zur Folge, dass selbst bei der Verwendung virtueller Engines (siehe Kapitel 4.10) die Zwischenergebnisse auf ein persistentes Speichermedium geschrieben werden, was zu einer langsamen Ausführung führt. Das Format dieser Zwischenablage kann beispielsweise eine JSON-Datei im Hadoop Distributed File System sein. In diesem Falle kommt ein weiter Nachteil zum Vorschein, nämlich dass keine benutzerdefinierten Typen beim Verwenden virtueller Engines zum Einsatz kommen können, für welche keine Abbildung auf einen JSON-Typ existiert, z. B. Bilddaten.

Da MapReduce keine Anfragesprache, sondern ein Programmiermodell und ein Framework ist, kann eine MapReduce-basierte Ausführungsplattform für NotaQL-Transformationen von der kompletten Mächtigkeit der zugrundeliegenden Programmiersprache Gebrauch machen. Diese *computational Completeness* ermöglicht das Einbinden benutzerdefinierter Engines, Funktionen und Datentypen. Lediglich bei letzteren können – zusätzlich zur Transformationsketten-Problematik – Probleme auftreten, wenn diese nicht die Methoden zur Serialisierung und Deserialisierung anbieten, die zur Übertragung der Daten über das Netzwerk verlangt werden.

Durch das batchorientierte Verhalten haben MapReduce-Jobs eine lange Initialisierungsphase und ermöglichen keinerlei Interaktion. Somit werden keine Datenstrom-Transformationen unterstützt.

## 5.2 Spark

Beim Framework *Apache Spark* [Zah+10] können Jobs flexibler gestaltet werden als bei MapReduce. Das Programmiermodell legt nicht fest, aus welchen Phasen ein Spark-Job besteht, sondern es erlaubt beliebige Abfolgen von *Transformationsschritten* auf *Resilient Distributed Datasets* (*RDDs*). Abbildung 5.6 zeigt jedoch, dass, wenn wir NotaQL-Transformationen auf Spark ausführen möchten, der Ablauf fast identisch zu der im vorherigen Abschnitt präsentierten MapReduce-basierten Ausführung ist [SLD16]. Lediglich die Anwendung des Eingabefilters wurde aus der Map-Funktion in einen Filter-Schritt ausgelagert.
Die Vorteile von Spark kommen zum Vorschein, wenn NotaQL-Transformationen Verbundoperationen durchführen oder wenn Transformationsketten vorliegen. Bevor auf diese Besonderheiten eingegangen wird, betrachten wir zunächst jedoch die allgemeine Ausführung von NotaQL-Skripten auf dem Spark-Framework im Detail.

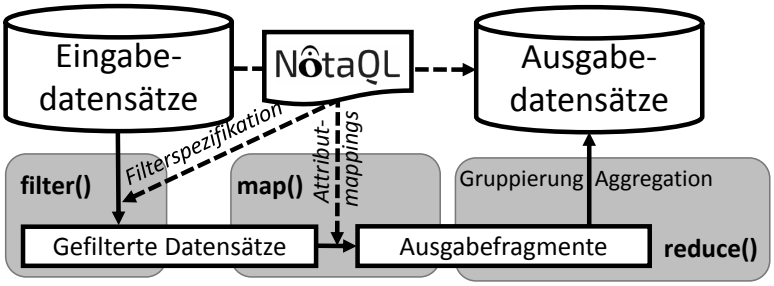



**Abbildung 5.6:** Ausführung eines NotaQL-Skripts mittels Apache Spark

### Resilient Distributed Datasets

Jeder Spark-Transformationsschritt wird auf einem RDD aufgerufen und liefert wiederum einen RDD. Diese Abgeschlossenheit macht ein beliebiges Hintereinanderschalten von Schritten möglich. Wie in Abbildung 5.6 zu sehen, ist der Ablauf einer NotaQL-Transformation folgender: `filter` → `map` → `reduce`. Bevor jedoch der erste Schritt durchgeführt werden kann, müssen die Eingabedaten als RDD vorliegen. Entweder werden diese innerhalb der Eingabe-Engine erzeugt oder außerhalb. Ersteres würde bedeuten, dass ein Engine-Entwickler seine `evaluate`-Methode so entwickeln muss, dass sie einen RDD zurückliefert. Im zweiten Fall kann ebendiese Methode ein beliebiges iterierbares Objekt liefern, also eine Liste, eine Map, ein Cursor oder Ähnliches. Wird dieses anschließend in ein RDD gewandelt, kann Spark die Transformationsschritte darauf anwenden.

Die Lösung, dass Eingabe-Engines direkt RDDs ausgeben, ist vor allem dann sinnvoll, wenn die Eingabedaten verteilt vorliegen. Dann nämlich ist Spark in der Lage, die beiden nachfolgenden Schritte *Filter* und *Map* lokal dort auszuführen, wo die Daten gespeichert sind. Diese beiden Schritte sind vom Typ der *engen Abhängigkeiten*, was bedeutet, dass sie lokal und innerhalb einer einzigen Phase ausgeführt werden können. Erst der *Reduce*-Schritt hat *weite Abhängigkeiten*. Ähnlich wie bei MapReduce ist dazu ein Datentransport zwischen verschiedenen Rechnern vonnöten. Der *Reduce*-Schritt kann erst dann ausgeführt werden, wenn die vollständige Menge an Ausgabefragmenten zu einem Ausgabedatensatz vorliegen.

### Transformationsketten und Verbundoperationen

Dadurch, dass Spark beliebige Abfolgen von Transformationsschritten erlaubt und eingebaute Operationen für Vereinigungen und Verbunde bietet, sind sowohl NotaQL-Transformationsketten als auch Skripte mit Join-Engines effizient ausführbar. Auch iterative Algorithmen, in welchen die `REPEAT`-Klausel verwendet wird, können innerhalb eines einzigen Spark-Jobs ausgeführt werden. Abbildung 5.7 zeigt eine Beispielkette aus Kapitel 4.10. Die linken beiden Transformationen $t_1$ und $t_2$ verwenden jeweils die Ausgabe-Engine-Spezifikation `OUT-ENGINE: virtual(id<-'ab')`, die beiden rechten Transformationen $t_3$ und $t_4$ die dementsprechende Eingabe-Engine-Spezifikation `IN-ENGINE: virtual(id<-'ab')`. Letztere erhalten also als Eingabe jeweils die Vereinigung der Ausgaben der beiden linken Transformationen.

**Vereinigung** Da die Ausgaben von $t_1$ und $t_2$ jeweils als RDD vorliegen, kann der Spark-Schritt `union()` zum Einsatz kommen, um die beiden RDDs zu vereinigen. Nun ist $ab$ das RDD, welches das Resultat der Vereinigung beinhaltet.

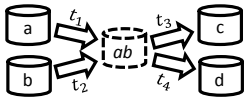



**Abbildung 5.7:** Transformationskette

**Mehrfachverwenden von RDDs** Die Transformationen $t_3$ und $t_4$ erhalten keinen RDD als Eingabe, welcher von einer Eingabe-Engine produziert wurde, sondern das fertige vorliegende RDD aus den vorangegangenen Transformationen, in unserem Falle also das RDD $ab$. Eine Möglichkeit wäre nun, die zur Ausführung eines NotaQL-Skripts nötigen Schritte direkt auf dieser Variable auszuführen. Das Problem daran ist jedoch, dass Spark Transformationen *lazy* ausführt. Das bedeutet, dass die Schritte in $t_1$, $t_2$ sowie der Vereinigungsschritt erst dann ausgeführt werden, wenn eine *Aktion* erfolgt. Die in einer NotaQL-Transformation vorkommende Aktion ist üblicherweise ein `collect()`, um die im RDD befindlichen Daten zu erhalten und sie anschließend im Ausgabesystem zu speichern. Bei der gegebenen Transformationkette würde dann beim Speichern der Ausgabe von $t_3$ und $t_4$ jeweils der komplette Prozess ablaufen. Die Zwischenergebnisse $ab$ können nicht wiederverwendet werden. Eine Lösung des Problems ist durch Einfügung des Zwischenschritts `cache()` möglich. Dieser materialisiert den Zustand des RDDs $ab$ und behält ihn im Arbeitsspeicher. `cache()` ist genau wie `collect()` eine Spark-Aktion, die die Ausführung der vorangegangenen Transformationsschritte anstößt. Die vollständige Abbildung der gegebenen NotaQL-Kette aus Abbildung 5.7 in Spark sieht also wie folgt aus[1]:

```
t1 = a.filter().map().reduce()
t2 = b.filter().map().reduce()
ab = t1.union(t2).cache()
t3 = ab.filter().map().reduce()
t4 = ab.filter().map().reduce()
c = t3.collect()       // Schreiben der Endergebnisse
d = t4.collect()
```

**Joins** Lokale Verbunde, die direkt innerhalb einer Eingabe-Engine ausgeführt werden, sind genau wie bei der MapReduce-basierten Ausführung in Spark trivial. Spark erhält in dem Fall direkt als Eingabe die bereits verbundenen Daten. Bei der Verwendung von systemübergreifenden Joins mittels der in Kapitel 4.8.4 vorgestellten generischen delegierenden Join-Engine kann die Spark-eigene Verbundoperation zum Einsatz kommen. Das erspart zum einen die Implementierung des Joins und zum anderen ist dadurch eine verteilte Verbund-Ausführung möglich. Jedoch werden in Spark nur Gleichverbunde unterstützt, keine beliebigen Theta-Joins. Sei nun im NotaQL-Skript ein Gleichverbund zwischen den Datenquellen $a$ und $b$ über das Verbundattribut $k$ gegeben, dann können die folgenden Schritte ausgeführt werden, um den Verbund mit Spark auszuführen:

[1] Zur besseren Lesbarkeit wurden die Argumente der Funktionen ausgelassen.

```
a' = a.mapToPair(x => (x.k, x))
b' = b.mapToPair(x => (x.k, x))
verbund = a'.join(b')
```

Es ist zu sehen, dass ein `mapToPair`-Schritt vonnöten ist, um die Eingabedatensätze in Schlüssel-Wert-Paare aufzuteilen. Das Verbundsattribut wird aus dem Datensatz extrahiert und als Schlüssel verwendet. Der `join()`-Schritt verbindet anschließend die Datensätze der beiden vorliegenden *Pair-RDDs* anhand gleicher Schlüsselwerte. Auf dem als RDD vorliegenden Verbundergebnis können anschließend wie gehabt die NotaQL-Transformationsschritte ausgeführt werden.

Ist kein Gleichverbund gegeben, ist der Spark-Schritt `cartesian()` hilfreich. Dieser erzeugt ein kartesisches Produkt zwischen den beiden Eingabequellen. Im Anschluss darauf kann mittels eines `filter()`-Schritts das Verbundsprädikat angewandt werden. Da zuvor jedoch die Datensätze in allen möglichen Kombinationen miteinander vorliegen, ist diese Berechnung üblicherweise nicht besonders effizient. Mögliche Optimierungen sind mit einem *Predicate-Pushdown* möglich. Dabei werden Selektionen, die nur eine Eingabequelle betreffen, bereits vor dem Verbundschritt ausgeführt.

### Datenströme

Das Projekt *Spark Streaming* erweitert Apache Spark um die Echtzeitverarbeitung von Datenströmen. Dazu wird das Konzept der *diskretisierten Ströme* (*DStream*[2]) eingeführt. In festen Zeitabständen werden die eintreffenden Elemente in einem als DStream vorliegendem Eingabedatenstrom zu einem *Micro-Batch* zusammengefasst. Dieser Micro-Batch kann als RDD angesehen werden, sodass man auf diesem die gewohnten Spark-Transformationsschritte anwenden kann. Das Zeitintervall wird per Parameter konfiguriert. Es legt zum einen die *Fenstergröße* eines Micro-Batches fest und zum anderen gleichzeitig auch das *Aktualisierungsintervall*. In NotaQL erfolgt das Lesen eines Datenstroms ebenfalls über die Konfiguration einer Fenstergröße (`range`/`rows`) und eines Aktualisierungsintervalls (`slide`). Weitere Details dazu befinden sich in Kapitel 4.9.2. Es fallen jedoch zwei große Unterschieden im Vergleich zu Spark Streaming auf. Zum ersten muss in Spark die Fenstergröße zeitlich definiert sein. Ein Micro-Batch, welcher über die Anzahl der darin enthaltenen Datensätze definiert ist, ist nicht möglich. Zum zweiten wird in Spark nur ein einziges Intervall angegeben, welches sowohl für `range` als auch `slide` gilt. In Kapitel 4.9 wurde das Beispiel präsentiert, in welchem jede Stunde die Durchschnittstemperatur der vergangenen Stunde ausgeben wird. Dazu wird in Spark die Micro-Batch-Länge auf eine Stunde gesetzt. Soll die Ausgabe allerdings öfter oder seltener, beispielsweise alle halbe Stunde oder alle zwei Stunden erfolgen, stoßen wir auf einen Konflikt.

Zur Lösung des gerade geschilderten Problems gibt es in Spark Streaming sogenannte *Fensteroperationen*, um mehrere Micro-Batches eines DStreams zusammenfassen. Die auf einem DStream anwendbare `window`-Methode erhält als Parameter zwei Zeitdauern für die Fenstergröße und für das Aktualisierungsintervall. Die Ausgabe ist wiederum ein DStream. Durch

[2] Nicht zu verwechseln mit den *Delete-Streams* aus CQL.

diese Funktionalität ist es sehr einfach, die in einer NotaQL-Streaming-Engine-Spezifikation angegebenen `range`- und `slide`-Werte an Spark zu übergeben. Zunächst wird das Zeitintervall des zugrundeliegenden DStreams auf das Minimum der beiden Angaben gesetzt. Sind `range` und `slide` verschieden groß, kommt die `window`-Methode zum Einsatz, um die tatsächliche Fenstergröße und das Intervall zu setzen.

Als Resultat liegt der Spark-Streaming-basierten NotaQL-Ausführungsplattform nach der Fensterdefinition ein RDD vor, welches alle gewünschten Datensätze (gemäß der Engine-Spezifikation) enthält. Auf diesem RDD kann wie gehabt mit `filter()`,`map()`,`reduce()`,`join()`,`union()` etc. gearbeitet werden. Auch Verbunde zwischen Datenstrom-Fenstern und Datenbankdaten sind möglich.

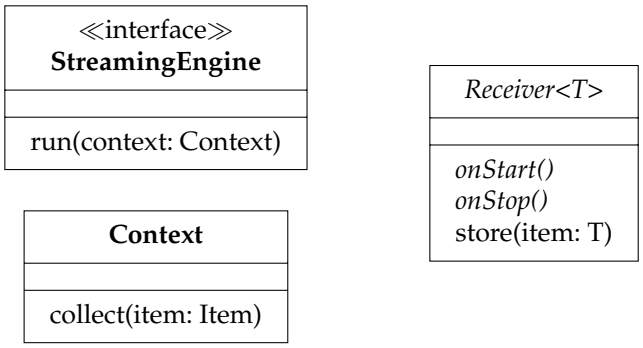



**Abbildung 5.8:** Streaming-Klassen in NotaQL und Spark Streaming

Das UML-Diagramm in Abbildung 5.8 zeigt die korrespondierenden Klassen in NotaQL und Spark Streaming. Die Aufgaben dieser Klassen ist es, die Eingabedaten für eine Datenstrom-Transformationen zu liefern. Der Entwickler einer NotaQL-Streaming-Engine implementiert eine Methode run(), in welcher die Verbindung mit dem Eingabestrom hergestellt wird. Sobald in diesem Strom ein Datensatz gesehen wird, leitet ein Aufruf der Context-Methode `collect` diesen Datensatz an die NotaQL-Plattform weiter. Auf der Spark-Streaming-Seite ist das Vorgehen ähnlich. Damit ein Datenstrom als Eingabe für einen Spark-Streaming-Job dienen kann, implementiert man die Methode `onStart()` und ruft dort die `store()`-Methode für jeden erhaltenen Datensatz auf. Um einen allgemeingültigen *NotaQL-Receiver* zu entwickeln, muss in der `onStart()`-Methode die `run()`-Methode der im NotaQL-Skript spezifizierten Eingabeengine aufgerufen werden. Als Parameter erhält diese ein *Context*-Objekt, in welchem die `collect`-Aufrufe an die Receiver-eigene `store`-Methode delegiert werden.

### Bewertung

Verwendet man lediglich die Spark-Schritte `map()` und `reduce()`, lässt sich alles Gelernte aus der Hadoop-basierten Plattform auf Spark übertragen. Dies ermöglicht eine ähnlich simple Ausführung von NotaQL-Transformationen auf Spark wie auf MapReduce. An den Punkten, an denen MapReduce an seine Grenzen stößt – also bei Transformationsketten und Verbundoperationen –, bietet Apache Spark passende Funktionalität, um auch diese Arten von Transformationen ausführbar zu machen. Da Spark Zwischenergebnisse im Arbeitsspeicher behält und nicht wie Hadoop auf ein persistentes Medium schreibt, können Ketten von Transformationen effizient ausgeführt werden. Bei den in delegierenden Engines durchgeführten Verbundoperationen hilft die Spark-eigene Verbundfunktionalität, Joins schnell und verteilt auszuführen.

Spark kann als einer der erfolgreichsten Nachfolger von MapReduce gesehen werden. Dies ist auch daran zu erkennen, dass viele Erweiterungen sowie Anbindungen zu Datenbanksystemen und Datenstromsystemen existieren. Das einfache Verarbeitungsmodell, die verteilte Ausführung, die Skalierbarkeit, Monitoring-Tools sind einige der Stärken von Spark, die MapReduce aber auch hat. Das flexiblere Ausführungsmodell und die Unterstützung von Datenströmen sind jedoch zwei zusätzliche Pluspunkte, die zur Ausführung von NotaQL-Transformationen durchaus von Nutzen sind.

Ursprünglich war Apache Spark für die Stapelverarbeitung großer Datenmengen gedacht. Erst durch die Streaming-Erweiterungen werden Datenströme unterstützt. Die Anbindung erfolgt durch die Erzeugung von Zeitintervall-basierten Micro-Batches, welche sich als RDD weiterverarbeiten lassen. Durch diese Abbildung von Strömen auf RDDs gehen allerdings einige für Datenstrom-Transformationen nützliche Funktionalität verloren, beispielsweise die Tupel-basierte Definition von Fenstern.

Ein zu Spark ähnliches Framework ist *Apache Flink* [Apa15b]. Anders als Spark war es von Anfang an für die Datenstromverarbeitung gedacht. Flink unterstützt Tupel-basierte Fenster und Aktualisierungsintervalle und eignet sich daher besser für eine NotaQL-Ausführungsplattform für Datenströme. Im Wesentlichen ist die Implementierung einer Flink-basierten Plattform [Emd16] ähnlich zu der in diesem Abschnitt präsentierten Spark-basierten. Beide bieten Transformationsschritte wie `filter()` und `map()`, sowohl für die Stapelverarbeitung von Datenbankdaten als auch für Datenströme. Da Spark jedoch das prominenteste und meistgenutzte Framework dieser Art ist, wurde sich in dieser Arbeit auf Spark fokussiert. Alle Ansätze lassen sich jedoch auf Flink übertragen.

## 5.3 Graph-Frameworks

Ein Graph in NotaQL kann als eine Menge von Knoten angesehen werden, welche nacheinander verarbeitet werden – ähnlich zu Dokumenten in einer Dokumentendatenbank. Zu jedem vorliegenden Knoten können die Properties, Labels und Kanten abgerufen werden. Bei letzteren hat der NotaQL-Anwender Zugriff auf Kantenproperties sowie wiederum die Properties, Labels und Kanten der Nachbarknoten. Logisch gesehen erhält der Anwender also eine gesamtheitliche Sicht auf jeden einzelnen Knoten. Ein Knoten steht mit samt seinen Attributen und den im NotaQL-Skript gewünschten Informationen über Kanten und Nachbarn zur Verarbeitung zur Verfügung. Beim Formulieren eines NotaQL-Skriptes sieht es für den Benutzer so aus, als ob Nachbarknoten ähnlich wie verschachtelte Unterdokumente in Dokumentendatenbanken Teil des Knotens sind. Dies ist mit den zuvor präsentierten Ansätzen wie MapReduce oder Spark realisierbar, indem die Graph-Traversierung innerhalb der Eingabe-Engine durchgeführt wird. Sie ruft die benötigten Attributwerte ab und legt sie in ein NotaQL-Objekt. In den nachfolgenden Verarbeitungsschritten kann schließlich eine Selektion, Projektion und Aggregation auf diesen erfolgen.

Graph-Verarbeitungsframeworks haben meist ein Nachrichten-basiertes Ausführungsmodell [Mal+10; Ave11; Xin+13]. Die Stärken dieser Frameworks gegenüber MapReduce kommen vor allem bei iterativen Graph-Algorithmen zum Vorschein. Diese sind in MapReduce sehr aufwändig, da Zwischenergebnisse nach jeder Iteration auf die Festplatte geschrieben werden und da ständig Kantendaten zur Erhaltung der Graphstruktur zwischen den Phasen Map und Reduce übertragen werden müssen [Lin13]. In einem Graph-Framework wird ein über mehrere Iterationen definierter Algorithmus als ein einziger Gesamtprozess angesehen. In jeder Iteration schicken Knoten Nachrichten zu anderen Knoten und aggregieren Daten aus eingehenden Nachrichten aus der vorherigen Iterationsrunde. Im System *Pregel* realisiert ein Anwender dieses Verhalten durch die Implementierung einer `compute()`-Methode. Diese wird für jeden Knoten aufgerufen und erhält als Eingabe die Nachrichten aus der vorherigen Runde. In der ersten Iteration ist dies eine leere Liste. Abbildung 5.9 zeigt, wie ein NotaQL-Skript mittels Pregel ausgeführt werden kann. Da mit Pregel nur In-Place-Transformationen möglich sind, ist die Eingabe und Ausgabe der gleiche Graph.

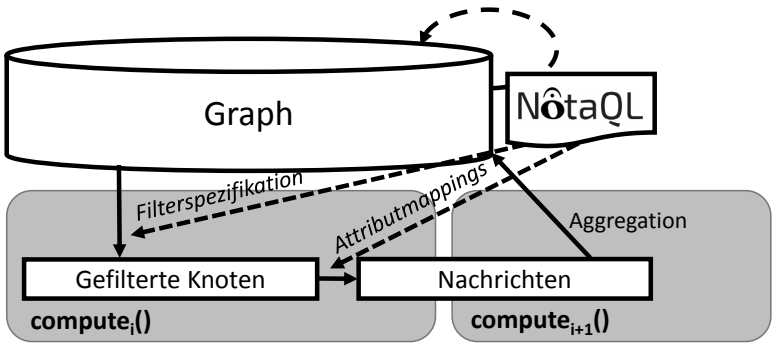



**Abbildung 5.9:** Ausführung eines NotaQL-Skripts mittels Pregel

In der `compute()`-Methode in Runde $i$ erfolgt das Filtern sowie die Anwendung der Attribut-Mappings. Das Resultat sind Ausgabefragmente in Form von Nachrichten. Die Belegung von `OUT._id` bestimmt den Empfänger einer Nachricht. Meist ist dies die ID eines Nachbarknotens. Die übrigen `OUT`-Attribute beinhalten den Nachrichteninhalt, also zum Beispiel eigene Property-Werte. Genau wie bei der Ausführung mit MapReduce und Spark wird beim Erzeugen von Ausgabefragmenten die im NotaQL-Skript verwendeten Aggregatfunktionen ignoriert. Während die genannten Frameworks diese in einer `reduce()`-Funktion ausführen, erfolgt die Aggregation im Graph-Framework innerhalb der `compute()`-Methode der nächsten Iterationsrunde. In dieser liegen jedem Knoten die Nachrichten der vorherigen Runde vor, sodass die neuen eigenen Property-Werte daraus berechnet und Nachrichten für die nächste Runde erzeugt werden können.

### Bewertung

An der grob vorgestellten Ausführungsidee lassen sich bereits die Grenzen des Ansatzes erkennen. Die speziell für Graph-Analysen gedachten Frameworks können nur solche NotaQL-Transformationen ausführen, für welche die Eingabe ein Graph ist und in welcher ebendieser Graph auf der Stelle geändert werden soll. Weder sind Erzeugungen neuer Graphen noch Transformationen von oder zu andersartigen Datenbanksystemen möglich. Graph-Verarbeitungsframeworks sind auch primär nicht für Graphdatenbanken gedacht. Sie kommen zum Einsatz, wenn Graphen beispielsweise als CSV-Dateien oder als Knoten- und Kantentabelle vorliegen. Da auf diesen kein effizientes Traversieren möglich ist, ist das Nachrichten-Pattern von Pregel und anderen Frameworks gut zur Verarbeitung geeignet. In Graphdatenbanken ist jedoch die interne Speicherung von Knoten und Kanten darauf ausgelegt, ein effizientes Navigieren zu ermöglichen. Dies kann beispielsweise in einer Spark-basierten NotaQL-Plattform ausgenutzt werden, indem in Eingabe-Engines die native Graphdatenbank-API zum Einsatz kommt.

Man muss unterscheiden, in welcher Form ein gegebener Graph vorliegt. Ist er in einer Graphdatenbank gespeichert, eignen sich zur Analyse und zur Transformation die native APIs der Systeme sowie Graph-Sprachen wie Cypher oder Gremlin. Liegt der Graph jedoch als einfache Datei vor, sind Graph-Frameworks wie *Pregel* [Mal+10], *Apache Giraph* [Ave11] oder *GraphX* [Xin+13] zur Verarbeitung besser geeignet. *Mazerunner* [Bas14] ist ein Programm zum Exportieren eines Neo4J-Graphen [Neo] in HDFS-Dateien [Apaa]. Es ermöglicht die anschließende Analyse des Graphens mittels eines der genannten Frameworks. Die Sprache NotaQL eignet sich für alle hier vorgestellten Ansätze. Also sowohl für Graphdatenbanken, als auch für Graphen in HDFS-Dateien, als auch für die Konvertierung vom einen ins andere. Die Ausführung ist in jedem Falle mit der im vorherigen Abschnitt präsentierten Spark-basierten Plattform möglich. Für ganz bestimmte NotaQL-Transformationen kann jedoch auch auf die in diesem Abschnitt vorgestellte Graph-Framework-basierte Ausführungsart ausgewichen werden.

## 5.4 Abbildung auf SQL

Statt NotaQL-Skripte zu interpretieren und sie direkt auszuführen, kann man sie auch lediglich in eine andere Sprache übersetzen und das übersetzte Skript zur Ausführung an eine existierende Plattform übergeben. Das hat den Vorteil, dass existierende Funktionalität, Datenbank-Anbindungen und Optimierungen voll ausgenutzt werden können. Die beiden Systeme *Grail* und *SQLGraph* verfolgen ebenfalls diesen Übersetzer-Ansatz. Die *Grail*-Plattform [FRP15] ist eine Alternative zu Nachrichten-basierten Graph-Verarbeitungsframeworks, wie sie im vorherigen Abschnitt präsentiert wurden. Die Autoren zeigen, dass die Funktionalität dieser Frameworks auch mit relationalen Datenbanken erreicht werden kann. Dazu formuliert ein Benutzer ein *Grail-Skript*. Dieses beschreibt, wie eine Transformation initialisiert wird und was in jedem einzelnen Iterationsschritt passieren soll, also beispielsweise das Senden von Nachrichten zu Nachbarknoten oder das Ändern von Knoten-Properties basierend auf den eingegangenen Nachrichten. Das Skript wird schließlich in eine SQL-Anfrage der Form `INSERT INTO ... SELECT` übersetzt, sodass diese in einer Schleife ausgeführt werden kann, um die iterative Berechnung auf Knoten- und Kantentabellen in der relationalen Datenbank auszuführen. Bei *SQLGraph* [Sun+15] werden Graphen ebenfalls in einer relationale Datenbank gespeichert. Die Plattform übersetzt Anfragen aus der Graph-Anfragesprache *Gremlin* [Rod15] in SQL.

In diesem Abschnitt stellen wir einen Übersetzungsansatz für NotaQL-Skripte am Beispiel der Datenbanksprache SQL vor. Er lässt sich aber auch auf andere Sprachen wie Pig [Ols+08], Gremlin [Rod15], Cypher [Neo] oder Sawzall [Pik+05] übertragen. Alle diese Sprachen stoßen jedoch an ihre Grenzen, wenn es darum geht, die Sprache NotaQL vollständig zu unterstützen. Auch bei der eigentlich sehr mächtigen Sprache SQL ist dies der Fall, beispielsweise wenn es darum geht, flexible Schemata zu unterstützen. Mit dieser Herausforderung hatten auch die Entwickler von SQLGraph zu kämpfen [Sun+15].

Liegt einer SQL-basierten Ausführungsplattform ein NotaQL-Skript vor, welches nicht direkt in SQL übersetzt werden kann, gibt es zwei Alternativen. Die erste ist, die Ausführung schlichtweg zu verweigern. Die oberste Regel bei der Ausführung ist nämlich – wie weiter oben beschrieben –, dass jede Plattform bei gleichen Eingabedaten die gleichen Ausgabedaten erzeugt. Ist diese korrekte Ausführung nicht möglich, ist es besser, gewisse Skripte gar nicht erst zu unterstützen. Für diese muss dann auf eine andere Ausführungsart zurückgreifen werden. Die zweite Alternative ist, die Lücken einer Sprache zu kompensieren. Dadurch wird das Ziel verfolgt, möglichst jedes Skript in eine Folge von SQL-Befehlen zu übersetzen, auch wenn SQL dazu zunächst ungeeignet ist. Da der erste Ansatz trivial und zudem nicht zufriedenstellend ist, diskutieren wir in diesem Abschnitt vor allem den zweiten Ansatz.

Da SQL eine andere Ausführungssemantik hat als NotaQL, können die Schritte bei der NotaQL-Skript-Auswertung nicht eins zu eins auf die SQL-Klauseln abgebildet werden. Stattdessen soll Abbildung 5.10 zeigen, welche Klauseln typischerweise erzeugt werden und aus welchen Teilen des NotaQL-Skripts diese ihre Informationen entnehmen. SQL ist eine Anfragesprache, NotaQL eine Transformationssprache. Um mit SQL tatsächliche Transformationen durchzuführen, werden Kommandos vom Typ `INSERT`

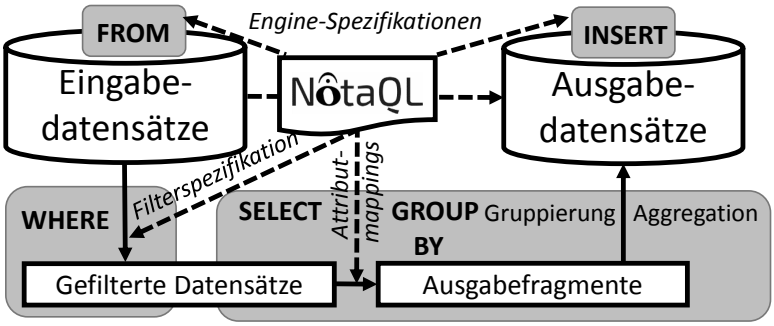



**Abbildung 5.10:** Übersetzung von NotaQL in SQL

oder `UPDATE` erzeugt. Zunächst gehen wir davon aus, dass die Eingabedaten in einer relationalen Datenbank vorliegen und auch in eine solche die Ausgabe geschrieben wird. Weiter unten in diesem Abschnitt betrachten wir die Anbindung von Engines. Wenn wir zusätzlich davon ausgehen, dass die Ausgabetabelle zu Beginn leer ist, können wir uns auf `INSERT`-Kommandos beschränken. Die andernfalls auftretenden Spezialfälle werden ebenfalls weiter unten diskutiert. Eine aus einem NotaQL-Skript erzeugte SQL-Anfrage hat in der Regel die folgende Form:

```
INSERT INTO ?ausgabetabelle
SELECT ?ausgabeschluessel, ?ausgabeattribute
FROM ?eingabetabelle
WHERE ?eingabefilter
GROUP BY ?ausgabeschluessel
```

Die Namen der Ein- und Ausgabetabelle stammen aus der Engine-Spezifikation im NotaQL-Skript. Das Filterprädikat kann ebenfalls vom Skript entnommen und in ein `WHERE`-Prädikat übersetzt werden. Beinhaltet das zu übersetztende Skript Aggregatfunktionen, ist der Ausgabeschlüssel, nachdem gruppiert wird, die Menge der Attribute ohne Aggregatfunktion. Bei Transformationen ohne Aggregation ist keine `GROUP BY`-Klausel nötig.

Das folgende Beispiel zeigt ein NotaQL-Skript, in dem zu jeder Firma die Durchschnittsgehälter der Programmierer berechnet werden, und die daraus generierte SQL-Anfrage:

```
IN-ENGINE: jdbc(url <- '...', table <- 'personen'),
OUT-ENGINE: jdbc(url <- '...', table <- 'firmen'),
IN-FILTER: IN.beruf = 'Programmierer',
OUT.firma <- IN.firma,
OUT.avgGehalt <- AVG(IN.gehalt)
```

```
INSERT INTO firmen
SELECT firma, AVG(gehalt) AS avgGehalt
FROM personen
WHERE beruf = 'Programmierer'
GROUP BY firma
```

Die Übersetzung und Ausführung von Skripten ähnlich zu dem gezeigten Beispiel sind problemlos möglich, sofern die Ein- und Ausgabetabelle in der gleichen Datenbank liegen. Ist dies nicht der Fall, oder kommen

NotaQL-Sprachkonstrukte zum Einsatz, für die es kein Äquivalent in SQL gibt, sind komplexere Übersetzungsschritte notwendig. Des Weiteren ist vonnöten, dass die Ausgabetabelle existiert und die korrekten Spalten besitzt.

### Plangenerierung und -ausführung

Anders als bei den weiter oben vorgestellten Plattformen, die auf einem Framework wie MapReduce oder Spark basieren, existiert in der SQL-basierten Plattformen kein allgemeingültiger Ausführungsweg, der sich für alle NotaQL-Skripte gleichermaßen eignet. Stattdessen teilen wir die Ausführung in zwei Teile [Sch15]: Im ersten Teil wird aus einem gegebenen NotaQL-Skript ein *Ausführungsplan* erzeugt, im zweiten Teil wird dieser ausgeführt. Der Plan beschreibt das Vorgehen, wie die Ausführung vonstattengehen soll, als eine Folge von *Schritten*. Abbildung 5.11 visualisiert die verschiedenen Schritt-Typen.

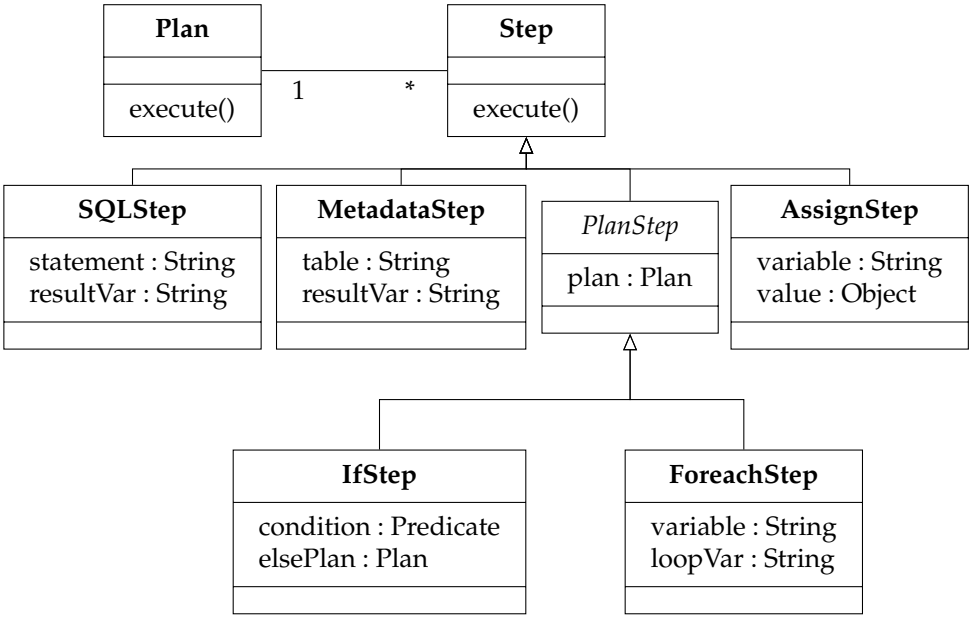



**Abbildung 5.11:** Mögliche Schritte in einem Ausführungsplan

Der Plan für das weiter oben in diesem Abschnitt gezeigte NotaQL-Skript, welches auf einfache Weise in eine SQL-Anfrage übersetzt werden kann, besteht nur aus einem einzigen Schritt, und zwar einem *SQL-Schritt*. Bei der Generierung des Plans erhält der SQL-Schritt als Parameter das oben stehende SQL-Statement, welches die Transformation ausführt. Der Ergebnisvariablen-Parameter wird leer gelassen, da der Befehl keine Ergebnisdaten zurückliefert.

Die im UML-Diagramm dargestellten Parameter, über die ein Schritt konfiguriert wird, werden aus einem gegebenen NotaQL-Skript genau wie die Schritte selbst generiert und sind unabhängig von den Eingabedaten. Da es jedoch durchaus vorkommen kann, dass Teile von SQL-Statements und andere Parameterwerte noch nicht zum Zeitpunkt der Plangenerierung feststehen, sondern erst zur Laufzeit, haben alle Schritte Zugriff auf einen *Kontext*. In diesem werden Variablen samt ihren meist erst zur Laufzeit gesetzten Werten verwaltet. Wenn ein SQL-Schritt beispielsweise eine

`SELECT`-Anfrage ausführt, kann in diesem Schritt die zurückgelieferte Ergebnismenge in eine Kontext-Variable geschrieben werden. Anschließend könnte beispielsweise ein *Foreach-Schritt* über diese Ergebnismenge iterieren. Auf diese Art und Weise können auch flexible Schemata unterstützt werden. Im Folgenden stellen wir die einzelnen Schritte kurz vor und diskutieren anschließend, wie typische Ausführungspläne generiert werden.

**SQL-Schritt** Ein SQL-Schritt erhält als Parameter die auszuführende SQL-Anfrage und sowie den Namen einer Variable, in die das Resultat der Anfrage abgelegt wird. In späteren Schritten kann über diese Variable auf die Ergebnismenge zurückgegriffen werden. Der Typ der Variable ist eine Liste von Tupeln. Auf die einzelnen Tupel kann unter Angabe der Spaltennamen auf einzelne Spaltenwerte zugegriffen werden.

Oft steht zum Zeitpunkt der Plangenerierung noch nicht die genaue SQL-Anfrage fest, beispielsweise wenn dynamische Spaltennamen verwendet werden oder wenn Werte von Kontext-Variablen in die Datenbank eingefügt werden sollen. In solchen Fällen kann mittels `${var}` auf Variablen – in diesem Fall auf den Wert der Variable `var` – zugegriffen werden.

**Metadaten-Schritt** Während der SQL-Schritt für DML- und DDL-Kommandos verwendet wird, bietet der Metadaten-Schritt einen lesenden Zugriff auf das Datenbankschema. Als Parameter wird ein Tabellenname übergeben sowie eine Variable, in welche eine Liste von Spaltennamen abgelegt wird. Auch hier kann der Tabellenname dynamisch festgelegt werden, z. B. `temp_${tabelle}`.

**If-Schritt** Der If-Schritt ist eine Subklasse des abstrakten *Plan-Schritts*. Ihm wird eine Bedingung übergeben sowie zwei Sub-Pläne. Je nachdem, ob die Bedingung zur Laufzeit als wahr oder falsch evaluiert wird, wird entweder der erste oder der zweite Sub-Plan ausgeführt. Die übergebene Bedingung kann aus booleschen Operatoren, Vergleichsoperatoren sowie Variablennamen und Literalen bestehen.

**Foreach-Schritt** Der Plan, welcher dem Foreach-Schritt übergeben wird, wird für jedes Element in einer angegebenen Liste einmal ausgeführt. Das aktuelle Element der Liste wird in eine Variable mit dem im zweiten Parameter übergebenen Namen abgelegt, sodass im Sub-Plan darauf zugegriffen werden kann.

**Zuweisungs-Schritt** Der Zuweisungsschritt dient lediglich dazu, um einer Kontekt-Variable einen neuen Wert zuzuweisen. Dieser Wert kann aus einer anderen Variable entnommen werden oder das Ergebnis einer Berechnung sein.

Betrachten wir im Folgenden ein Beispiel, in welchem eine komplexe Transformation ausgeführt wird. Eine Personentabelle habe eine Namensspalte sowie beliebige andere Spalten, in denen Gehaltskomponenten stehen, beispielsweise die Spalten `grundgehalt`, `bonus` und `weihnachtsgeld`. Es liegt also ein flexibles Schema vor, die genauen Spaltennamen der Eingabetabelle sind unbekannt.

```
IN-ENGINE: jdbc(url <- '...', table <- 'personen'),
OUT-ENGINE: jdbc(url <- '...', table <- 'personen2'),
OUT.name <- IN.name,
OUT.gehalt <- SUM(IN.?(IN.name() != 'name'))
```

Das vorliegende NotaQL-Skript ist eine leicht abgewandelte Version des Beispiels zur horizontalen Aggregation aus Kapitel 4.4. Die Ausgabetabelle besitzt zwei Spalten. In der ersten steht der Name einer Person, in der zweiten die Summe aller Werte der übrigen Spalten. Aus dem gegebenen Skript wird folgender Plan generiert:

1. *SQL-Schritt*: Erstellen der Ausgabetabelle
   `statement = "CREATE TABLE personen2 (name VARCHAR(1000) PRIMARY KEY, gehalt DOUBLE)"`
2. *Metadaten-Schritt*: Holen der Spaltennamen
   `table = "personen"`
   `resultVariable = "spalten"`
3. *Zuweisungsschritt*: Initialisierung der Summenliste
   `variable = "summe", value = 0`
4. *Foreach-Schritt*: Horizontale Aggregation
   `variable = "spalten", loopVariable = "spalte"`
   4.1. *If-Schritt*: Herausfiltern der Name-Spalte
   `condition = spalte!='name'`
   4.1.1. *Zuweisungsschritt*:
   `variable = "summe",`
   `value = "${summe+'+'+spalte}"`
5. *SQL-Schritt*: Transformation
   `statement = "INSERT INTO personen2 SELECT name, ${summe} FROM personen"`

Die im ersten Schritt verwendeten Spaltennamen und Datentypen zur Erzeugung der Tabelle wurden bei der Plangenerierung aus dem gegebenen NotaQL-Skript abgeleitet. Da die Gehaltsspalte eine Summe ist, muss ihr Wert nummerisch sein. Über die Namensspalte kann keine Aussage getroffen werden, deshalb wird hier VARCHAR mit einer (hoffentlich ausreichenden) Länge gewählt. Als Optimierung könnte ein Schritt die tatsächlichen Typen und Längen aus der Eingabetabelle extrahieren.

Da im gegebenen Skript mittels des `?`-Operators über die Eingabespalten iteriert wird, liegt der Summenfunktion ein listenwertiges Objekt vor, und zwar eine Liste von Spaltenwerten. Da es in SQL keine Möglichkeit gibt, diese Liste abzufragen, muss die Summenbildung unter Angabe aller vorhandenen Spaltennamen erfolgen. Das im letzten Schritt erzeugte SQL-Statement führt die eigentliche Transformation aus. Dieses Statement könnte wie folgt aussehen:

```
INSERT INTO personen2
SELECT name, 0+grundgehalt+bonus+weihnachtsgeld
FROM personen
```

In den im diesem Abschnitt gezeigten SQL-Anfragen vom Typ `INSERT INTO ... SELECT` werden alle Datensätze der Eingabetabelle auf einmal transformiert. Dies ist zwar in der Regel sehr effizient ausführbar, jedoch nicht bei jedem NotaQL-Skript möglich. Wenn das Schema der Ausgabetabelle zum Zeitpunkt der Plangenerierung unbekannt ist, kann nur ein Tupel-weises Einfügen erfolgen. In diesem Fall wird in einem Foreach-Schritt über die gefilterte Menge der Eingabedatensätze iteriert. In jedem Schleifendurchlauf transformiert und speichert ein SQL-Schritt einen Datensatz nach dem anderen mittels eines `INSERT`-Kommandos. Aufgrund des unbekannten Ausgabeschemas kann zudem in den einzelnen Schleifendurchläufen die Ausführung von `ALTER TABLE`-Kommandos vonnöten sein, um die Tabelle um zusätzliche Spalten zu erweitern.

### Systemübergreifende Transformationen

Die zuvor gezeigten NotaQL-Skripte lassen sich ausführen, wenn sich die Quelle und das Ziel in ein und derselben relationalen Datenbank befinden. Für Integrations- und Migrationszwecke sowie für alle anderen Transformationen, die beispielsweise auf NoSQL-Datenbanken oder Dateien arbeiten, sind jedoch weitere Maßnahmen vonnöten, um Datenbankgrenzen zu überwinden. Da in diesem Bereich schon viele Arbeiten und Werkzeuge existieren, können wir diese ausnutzen. Der Standard SQL/MED [Mel+02] sowie Wrapper-basierte Systeme [For; Cal; RS97; RLMW14] erlauben das Lesen von *externen Tabellen* in der `FROM`-Klausel. In unserem Falle wird dazu ein *generischer NotaQL-Wrapper* implementiert, welcher sich mit der im gegebenen NotaQL-Skript spezifizierten Engine verbindet und die Daten dem SQL-System liefert. Der SQL/MED-Standard beschreibt zwar nur lesenden Zugriff auf externe Tabellen, diverse Implementierungen bieten jedoch auch eine Schreib-API [For; Mul], sodass Statements vom Typ `INSERT`, `UPDATE` und `DELETE` auf ebendiesen Tabellen erfolgen kann, welche in einem `store`-Aufruf der jeweiligen Ausgabeengine resultiert.

Eine alternative Anbindung einer relationalen Datenbank an NotaQL-Engines ist mittels *benutzerdefinierter Routinen* möglich. So kann eine Prozedur `notaql_store`, welche als Paramenter einen Engine-Namen, die Engine-Parameter sowie einen Datensatz erhält, die betreffende Schreiboperation mittels der angegebenen Engine ausführen. Für Leseoperationen bieten sich *Tabellenfunktionen* an, welche sich genau wie externe Tabellen in der `FROM`-Klausel zum Einlesen der Eingabedaten verwenden lassen.

Benutzerdefinierte NotaQL-Funktionen (siehe Kapitel 4.7) können ebenfalls mittels generischen benutzerdefinierten Datenbankroutinen aufgerufen werden. So kann die Funktion `notaql_udf` den Namen einer benutzerdefinierten Funktion und die Funktionsparameter entgegennehmen. Um eine allgemeingültige Signatur definieren zu können, müssen beliebige Parameter und beliebige Rückgabewerte unterstützt werden. Dies ist beispielsweise möglich, indem Parameter und Rückgabe den Typ VARCHAR haben und einen JSON-String beinhalten. Dabei können sogar benutzerdefinierte Typen zum Einsatz kommen, sofern diese über eine Abbildung auf ein NotaQL-Objekt und damit auch auf ein JSON-Objekt definiert sind.

### Bewertung

Zusammenfassend lässt sich mit den in diesem Abschnitt präsentierten Ansätzen eine NotaQL-Ausführungsplattform realisieren, welche auf einem relationalen Datenbanksystem basiert. Interessant dabei ist, dass das Datenbanksystem nur zur reinen Ausführung und nicht zwangsläufig auch zur Datenspeicherung dient. Die größten Vorteile hat die Ausführungsplattform jedoch erst dann, wenn auch die Daten im relationalen Datenbanksystem liegen und nur selten bis nie externe Engines zum Einsatz kommen. Die vorgestellten Ansätze können auch dazu eingesetzt werden, um Systeme an die NotaQL-Plattform anzubinden, die eine SQL-Schnittstelle zu NoSQL-Datenbanken bieten, beispielsweise Hive [Thu+09], Impala [Kor+16] oder Phoenix [Apad]. Diese Systeme werden kontinuierlich auf bessere Performanz und Ausdrucksmächtigkeit optimiert, sodass eine NotaQL-Plattform von diesen Optimierungen profitieren kann. Der generische Ansatz über die Plangenerierung und Planausführung macht es möglich, NotaQL-Transformationen auch dann auszuführen, wenn es kein direktes Äquivalent in SQL gibt. Meist sind jedoch in diesen Fällen die entsprechenden Transformationen deutlich langsamer als auf Spark- oder MapReduce-basierten Plattformen.

Über die Performanz des SQL-basierten Ansatzes kann keine pauschale Aussage getroffen werden, da die Transformationsgeschwindigkeit vom NotaQL-Skript, von dem darunterliegenden Datenbanksystem und von der Art der Datenspeicherung – entweder in der Datenbank selbst oder via externer Engines – abhängt. Unterstützt das zugrundeliegende Datenbankmanagementsystem verteilte Speicherungen und verteilte Berechnungen, kommt auch dies der NotaQL-Ausführung zugute.

Die Grenzen des Ansatzes kommen vor allem bei der Behandlung von flexiblen Schemata zum Vorschein. Durch die planbasierte Ausführung werden solche Transformationen zwar unterstützt, sie sind aber langsam. Hat die Ausgabetabelle ein flexibales Schema, sind häufige `ALTER TABLE`-Kommandos zum Hinzufügen von Spalten vonnöten, welche sehr kostspielig sind. Für Datenstrom-Transformationen eignet sich die SQL-basierte Plattform nicht,

da Skripte nicht persistent abgelegt werden können. Das SQL-Äquivalent zu Strömen sind Trigger, die Änderungen in einer Basistabelle überwachen. Da die Basistabelle in dem hier vorgestellten Ansatz jedoch nur virtueller Natur ist und eine mittels eines Wrappers oder einer benutzerdefinierten Tabellenfunktion angebundene externe Engine darstellt, sind Trigger hier nicht einsetzbar.

Der Ansatz der Plangenerierung und -ausführung lässt sich auch auf Big-Data-Frameworks wie *ASTERIX* [Beh+11] übertragen. Dazu kann mittels des Compiler-Backends *Algebrix* [Bor+15] ein NotaQL-Compiler erstellt werden, welcher eine NotaQL-Vorschrift in einen Plan übersetzt. Dieser besteht aus algebraischen Operatoren und lässt sich direkt auf der ASTERIX zugrundeliegenden Plattform Hyracks ausführen. ASTERIX eignet sich jedoch überwiegend nur für in verteilten Dateisystemen wie HDFS gespeicherte Daten und nicht für Datenbanken. Die Plangenerierung erfolgt dennoch analog zum in diesem Abschnitt vorgestellten SQL-basierten Ansatz.

## 5.5 Zellen-basierte Speicherung und Ausführung

Das Problem beim im vorherigen Abschnitt vorgestellten SQL-basierten Ansatz ist die Schwäche von SQL, die bei der Behandlung von Metadaten zum Vorschein kommt. Wenn NotaQL-Transformationen auf flexiblen Schemata ausgeführt werden und auch die Ausgabe ein flexibles Schema besitzen kann, gibt es keine äquivalente SQL-Anfrage, in sich die Transformation übersetzen lässt. Stattdessen sind in Schleifen ausgeführte `INSERT`- und `ALTER TABLE`-Befehle vonnöten. Zum Ermitteln der in einer Tabelle vorhandenen Spalten muss auf Metadaten zugegriffen werden, was üblicherweise nicht mittels einfacher SQL-Befehle möglich ist.

Der nun folgende Ansatz basiert ebenfalls auf SQL, ihm liegt jedoch eine andere Art der Datenspeicherung zugrunde. Die Idee hinter der Speicherung ist es, Metadaten – in unserem Fall sind dies hauptsächlich Attributsnamen – auf der Datenebene abzuspeichern. Realisiert wird dies mittels einer *Zellen-basierten Speicherung*. Das heißt, dass ein Datensatz in seine Zellen aufgeteilt und jede Zelle als einzelnes Tupel gespeichert wird. In der Literatur sind für diesen Ansatz auch die Bezeichnungen *EAV* (Entity-Attribute-Value) oder *AOW* (Attribut-Objekt-Wert) üblich [Nad+99]. Eine EAV-Tabelle hat die drei Spalten *Entität*, *Attribut* und *Wert*, welche eine Subjekt-Prädikat-Objekt-Beziehung modellieren. Der Primärschlüssel ist zusammengesetzt aus der Entität und dem Attribut. Eine Instanz der EAV-Tabelle könnte beispielsweise `(1, vorname, Anita)` sein. Anders als in klassischen relationalen Tabellen sind die einzelnen Attributwerte einer Entität auf mehrere Zeilen aufgeteilt. Zur Beantwortung gängiger Anfragen sind viele Verbundoperationen nötig. Soll beispielsweise der Vor- und Nachname jener Personen gefunden werden, die in Kaiserslautern wohnen, würde die EAV-Tabelle dreimal in der `FROM`-Klausel vorkommen. Die Vorteile dieser Speicherung sind ein hohes Maß an *Flexibilität*, eine effiziente Speicherung von dünn besetzten Spalten (*sparse Data*) und der Fakt, dass die Daten *selbstbeschreibend* sind [Nad+99]. Das EAV-Modell hat große Ähnlichkeit zu NoSQL-Datenbanken. Daher lässt es sich eine NotaQL-Plattform realisieren, die auf dieser Art der Speicherung basiert. Wir werden einen generischen Ansatz vorstellen, um ein gegebenes NotaQL-Skript stets in eine

einzige SQL-Anfrage zu übersetzen, welche die gewünschte Transformation ausführt, sofern die Daten in ebendiesem Format vorliegen.

In der Literatur werden ähnliche Speicherungsansätze vorgestellt, beispielsweise das *Decomposition Storage Model* [CK85], in welchem die Tabellen nur zwei Spalten (Entität und Wert) besitzen. Bei dem vorgestellten Ansatz existiert für jedes einzelne Attribut eine solche Tabelle. Zwar resultieren aus dieser Art der Speicherung ähnliche Anfragen und auch die Vorteile bezüglich der Effizienz zur Speicherung von dünnbesetzten Spalten ist gegeben, jedoch geht die Schema-Flexibilität verloren, da Attributnamen weiterhin als Metadaten gespeichert sind; in diesem Fall in Form von Tabellennamen. Der Vorteil des Ansatzes ist jedoch, dass in jeder Tabelle ein für das jeweilige Attribut am besten passender Datentyp für die Werte-Spalte gewählt werden kann, während bei der EAV-Tabelle ein generischer Typ gewählt werden muss oder für verschiedene Datentypen separate Werte-Spalten existieren müssen.

Das im *Monet*-System implementierte Speichermodell [ASX01] basiert auf dem EAV-Ansatz. Agrawal et al. bezeichnen die Speicherung als *vertikal* im Gegensatz zu einer klassischen horizontalen Speicherung, in der jede Zeile alle Attribute besitzt. Sie stellen einen Ansatz vor, um Anfragen, welche auf dem horizontalen Modell basieren, auf die vertikale interne Speicherung zu übersetzen.

In einer auf dem EAV-Ansatz basierenden NotaQL-Plattform [Süß15] wird ein gegebenes Transformationsskript in eine SQL-Anfrage der Form `INSERT INTO ... SELECT` übersetzt. In der `FROM`-Klausel kommt die Eingabetabelle $n$ mal vor, wobei $n$ die Anzahl der Attribute ist, die entweder in der Eingabefilterspezifikation oder zum direkten Lesen von Eingabeattributen hinter `IN.` im NotaQL-Skript stehen. Durch einen Self-Join über die Entität-Spalte, welche die Objekt-Identität repräsentiert, liegen die einzelnen Datensätze samt aller zur Weiterverarbeitung benötigten Spalten vor. In der `WHERE`-Klausel kann danach der Eingabefilter überprüft werden und im `SELECT`-Teil werden stets genau drei Werte abgefragt, nämlich das, was gemäß des vorliegenden NotaQL-Skriptes Entität, Attribut und Wert der Ausgabe sein soll. Dies garantiert eine Abgeschlossenheit der Transformation, damit auch das Ergebnis wieder als EAV-Tabelle vorliegt. In der `GROUP BY`-Klausel wird stets nach den ersten beiden in der `SELECT`-Liste stehenden Attributen gruppiert, also nach der Entität und nach dem Attributsnamen der Ausgabe. Dieses Vorgehen muss für jede Attribut-Spezifikation wiederholt und die einzelnen generierten `SELECT`-Abfragen mit einer `UNION`-Operation vereinigt werden. Jedes einzelne Teil-`SELECT` kümmert sich dabei um die Berechnung einer Ausgabezelle.

Gegeben sei als Beispiel die in Tabelle 5.1 dargestellte Eingabetabelle `eav_in` und ein NotaQL-Skript, um die Vornamen und Nachnamen derjenigen Personen zu finden, die in Kaiserslautern leben:

```
IN-FILTER: IN.ort = 'Kaiserslautern',
OUT._id <- IN._id,
OUT.vorname <- IN.vorname,
OUT.nachname <- IN.nachname
```

Für die im NotaQL-Skript verwendete Objekt-ID `_id` verwenden wir die Werte der `E`-Spalte. Das gegebene Skript wird die unten stehende SQL-Anfrage übersetzt.

| E | A | V |
|---|---|---|
| 1 | vorname | Anita |
| 1 | gehalt | 52000 |
| 1 | ort | Kaiserslautern |
| 2 | vorname | Bruno |
| 2 | nachname | Bayer |
| 2 | gehalt | 53000 |
| 2 | ort | Ulm |

**Tabelle 5.1:** Eingabetabelle

```
1  CREATE TABLE eav_out (E INT, A VARCHAR(100),
2    V VARCHAR(100), PRIMARY KEY (E, A));
3  INSERT INTO eav_out
4    SELECT vorname.E AS E, 'vorname' AS A,
5           vorname.V AS V
6    FROM eav_in vorname JOIN eav_in ort USING (E)
7    WHERE vorname.A = 'vorname' AND ort.A = 'ort'
8    AND ort.V = 'Kaiserslautern'
9    GROUP BY vorname.E, 'vorname'
10  UNION
11   SELECT nachname.E AS E, 'nachname' AS A,
12          nachname.V AS V
13   FROM eav_in nachname JOIN eav_in ort USING (E)
14   WHERE nachname.A = 'nachname' AND ort.A = 'ort'
15   AND ort.V = 'Kaiserslautern'
16   GROUP BY nachname.E, 'nachname'
```

Zunächst wird die Ausgabetabelle erstellt (Zeilen 1-2). Im Anschluss erfolgt die eigentliche Datentransformation. In der `FROM`-Klausel (Zeile 5) verwenden wir die Eingabetabelle für die jeweiligen im Filter und der Attribut-Spezifikation geforderten Attribute. Der Verbund der Tabellen basiert auf der Entity-ID. In Zeile 7 erfolgt eine Überprüfung, die garantiert, dass die Tabellenaliasse `vorname` und `ort` auch tatsächlich Vornamen und Orte in der `V`-Spalte enthalten. Der Rest der `WHERE`-Klausel ist die eigentliche Filter-Operation (Zeile 8). Transformationen sind abgeschlossen, das heißt die Ausgabe jeder Transformation ist stets wieder eine Liste mit den Werten für `E`, `A` und `V`. Deshalb und aufgrund von `OUT._id <- IN._id` wird `t.E AS E` in der `SELECT-Liste` in Zeile 4 erzeugt. Das Attribut-Mapping `OUT.vorname <- IN.vorname` beinhaltet zweierlei Informationen. Zum einen, dass die Ausgabe ein Attribut mit dem Namen `vorname` erhalten soll und zum anderen, dass der Wert dieses Attributs auf dem Vornamen in der Eingabe basiert. Beides findet sich wieder in der SQL-Anfrage in den Zeilen 4 und 5. Die `GROUP BY`-Klausel in Zeile 9 ist für das gegebene NotaQL-Skript eigentlich nicht nötig, da keine Aggregation vorgenommen wird, sie schadet aber auch nicht. Die genannte Gruppierungsmenge `t.E, 'vorname'` wurde aus den ersten beiden Spalten der `SELECT`-Liste entnommen, was ein allgemeingültiges Vorgehen ist. Für das gegebene Skript wird nach der ID sowie nach einem konstanten Wert gruppiert. Dies heißt: Wenn zwei Eingabedatensätze die gleiche ID haben und wenn der String

`'vorname'` gleich dem String `'vorname'` ist, bilden diese beiden Eingabezeilen eine Gruppe. Ersteres Kriterium ist nie der Fall, zweiteres ist immer der Fall. Also hat die Gruppenbildung in diesem Fall keine Auswirkung. In den Zeilen 11 bis 16 wird die gleiche Abfrage noch einmal für das andere Attribut-Mapping vorgenommen, dieses Mal mit den Nachnamen-Zellen.

Die erzeugte Abfrage mag kuriose Gruppierungen und Vereinigungen enthalten, welche sich eleganter ausdrücken lassen, jedoch basiert sie auf einem Vorgehen, welches allgemeingültig ist und jede NotaQL-Transformation zu übersetzen in der Lage ist. Vereinfachungen und Optimierungen bei der Anfrageerzeugung sind natürlich möglich, nichtsdestoweniger sollte in den meisten Fällen aber bereits der Anfrageoptimierer des zugrundeliegenden Datenbankmanagementsystems in der Lage sein, die Anfrage adäquat zu optimieren.

Auch komplexere NotaQL-Skripte können auf die vorgestellte Art und Weise in SQL übersetzt werden. Aggregatfunktionen werden dazu einfach in die SQL-Anfrage übernommen. Werden Werte von mehreren Attributen unabhängig von deren Namen mittels `IN.*` oder `IN.?(...)` gelesen, spiegelt sich dies in Form eines anderen oder sogar wegfallenden Prädikats in der `WHERE`-Klausel wieder. Dadurch, dass Attributsnamen in der Spalte `A` auf Datenebene gespeichert sind, ist sowohl der lesende Zugriff auf Attributsnamen möglich als auch können beliebige Ausgabeattribute erzeugt werden.

### Bewertung

Der in diesem Abschnitt vorgestellte EAV-Ansatz stellt im Gegensatz zu den weiter oben präsentierten Ansätzen keine praktisch nutzbare Möglichkeit dar, eine NotaQL-Plattform zu implementieren. Anwendbar ist der Ansatz nur, wenn entweder die Eingabe in Form einer EAV-Tabelle vorliegt oder wenn eine Abbildung auf eine solche angegeben werden kann. Denkbar sind Sichten, Tabellenfunktionen oder externe Tabellen, die eine vertikale Sicht auf die tatsächlichen horizontalen Datenquellen bieten. Während dies mittels Wrappern für die Eingabe einer Transformation durchaus umsetzbar ist, stellt sich beim Schreiben der Transformationsausgabe die Frage, wie die Zellen-basierten `INSERT`-Operationen auf Einfügungen im Zielsystem abgebildet werden. Eine Abbildung auf die in NotaQL-Engines implementierte `store`-Methode ist nicht ohne Weiteres möglich.

Der EAV-Ansatz eignet sich weniger gut für systemübergreifende Transformationen. Jedoch zeigt der Ansatz, dass durch eine Speicherung der Attributsnamen auf Datenebene die Anwendbarkeit der Sprache SQL deutlich verbessert wird. Systeme, die intern das EAV-Speichermodell verwenden, erhalten durch das in diesem Abschnitt vorgestellte Verfahren eine mächtige und benutzerfreundliche NotaQL-Zugriffsschnittstelle.

Eine zum Lernen der Sprache NotaQL und zum Ausprobieren von NotaQL-Skripten implementierte Web-basierte *Sandbox* [Süß15] verwendet den vorgestellten EAV-Ansatz auf einer SQLite-Tabelle [Owe03]. Diese ermöglicht einfache Experimente mit NotaQL, ohne sich komplexe Frameworks oder NoSQL-Datenbanksysteme installieren zu müssen.

## 5.6 Virtuelle Transformationen

Die bisher in diesem Kapitel vorgestellten Ansätze zur Ausführung von NotaQL-Transformationen sind *materialisierter* Natur. Das heißt, dass das Transformationsergebnis in der Regel in eine Datenbank oder in eine Datei festgeschrieben wird. Greift man im Anschluss auf die geschriebenen Daten zu, handelt es dabei um einen direkten physischen Zugriff auf das Transformationsergebnis. Bei einem solchen Zugriff bleibt die Datenmenge, welche als Eingabe für die Transformation dient, unberührt.

Als Alternative zu materialisierten Transformationen werden in diesem Abschnitt *virtuelle Transformationen* [Bha15] vorgestellt. Diese beschreiben eine Ausführungsart, bei der zum Zeitpunkt der NotaQL-Skript-Einreichung keine wirkliche Datentransformation erfolgt, sondern das Skript lediglich ähnlich zu einer Sicht in einem Datenbanksystem im Metadatenkatalog abgelegt wird. Ähnlich zu den Methoden, mit denen ein Benutzer oder eine Anwendung auf ein materialisiertes Ergebnis zugreift, erfolgt auch der Zugriff auf das virtuelle Ergebnis. Jedoch löst in diesem Fall ebendieser Zugriff eine Berechnung der Transformation aus, während diese beim klassischen Ansatz bereits im Vornherein durchgeführt wird.

Vorteile virtueller Transformationen sind analog zu den Vorteilen virtueller Datenintegrationen (siehe Kapitel 2.4.2). Da die Berechnung erst zum Zeitpunkt der Abfrage erfolgt, basiert sie stets auf den aktuellen und nicht auf veralteten Basisdaten. Des Weiteren muss bei einer virtuellen Transformation nicht zwangsweise der komplette Datenbestand verarbeitet werden, sondern nur derjenige Teil, welcher für die gegebene Abfrage relevant ist. Da jedoch Datentransformationen auf großen Datenmengen oft komplex und langdauernd sind, bieten sich virtuelle Transformationen in diesem Falle nur dann an, wenn nur selten und nur gezielt auf die Ergebnisse zugegriffen wird. Greifen Anwendung sehr oft auf virtuelle Ausgaben zu oder iterieren sie über große Ergebnismengen, sind virtuelle Transformationen ungeeignet.

Virtuelle Transformationen lassen sich auf Datenbanken, Dateien und Diensten ausführen, sie passen jedoch nicht zur Ausführungssemantik von Datenstrom-Transformationen. Datenströme sind per se nicht materialisiert, dementsprechend liegen die Transformationsergebnisse ohnehin nur in flüchtiger Form vor.

Das SQL-Äquivalent zu virtuellen NotaQL-Transformationen sind Sichten. Eine Sicht wird über eine Anfrage $q$ und über einen Namen $v$ definiert. Erfolgt eine Anfrage auf die Sicht in Form einer SQL-Anfrage, in der $v$ in der `FROM`-Klausel auftaucht, kann $v$ in ebendieser Anfrage einfach durch $q$ ersetzt werden, wodurch sich eine geschachtelte Anfrage ergibt. Dies ist so einfach möglich, da sowohl die Sichtdefinition als auch die Anfrage auf die Sicht in SQL erfolgt und SQL geschachtelte Anfragen unterstützt. In NotaQL ist das Vorgehen komplizierter, da die Sprache nur zur Sichtdefinition und nicht zur Anfrage der Sichtdaten verwendet wird. Letzteres erfolgt mit einer Sprache oder Programmierschnittstelle, welche für das in der Ausgabe-Engine definierte System spezifisch ist. Aufgrund der unterschiedlichen Zugriffsschnittstellen auf das Transformationsergebnis kann es kein allgemeingültiges Vorgehen zur Ausführung virtueller Transformationen geben.

Betrachten wir zunächst als Beispiel eine NotaQL-Transformation, welche Daten von Personen aus Kaiserslautern aus einer MongoDB-Kollektion in eine HBase-Tabelle überträgt:

```
IN-ENGINE: mongodb(db<-'test', collection<-'pers'),
OUT-ENGINE: hbase(table<-'personen'),
IN-FILTER: IN.ort = 'Kaiserslautern',
OUT._r <- IN._id,
OUT.$(IN.*.name()) <- IN.@
```

Die untere Zeile beschreibt, dass alle Attribute aus den Personendokumenten als HBase-Spalten übernommen werden. Im Anschluss an die Transformation liegt eine Tabelle `personen` vor, auf welche mit klassischen HBase-Zugriffsmethoden – also mit `scan` und `get` – zugegriffen werden kann.

Sei nun das präsentierte Skript virtueller Natur. Bei der Einreichung des Skriptes muss zunächst sichergestellt werden, dass keine HBase-Tabelle mit dem Namen `personen` existiert, da es ansonsten zur Anfragezeit zu Konflikten kommt. Zudem muss auch kontinuierlich darauf geachtet werden, dass auch in Zukunft keine Tabelle mit einem solchen Namen erstellt wird. Die virtuelle HBase-Tabelle `personen` existiert nach Einreichung des Skriptes nicht tatsächlich in der HBase-Datenbank, trotzdem soll es möglich sein, mit den gewohnten Methoden darauf zuzugreifen.

Bei der Entwicklung einer NotaQL-Plattform für virtuelle Transformationen werden zwei Arten von Modulen benötigt, und zwar ein *Transformationsmodul* sowie diverse *Zugriffsmodule*. Während ersteres generisch ist, müssen Zugriffsmodule für jede zu unterstützende Ausgabe-Engine entwickelt werden. Dabei ist eine starke Kopplung mit der nativen API des jeweiligen Systems vonnöten. Abbildung 5.12 zeigt den Aufbau der in diesem Abschnitt präsentierten Transformationsplattform. Das generische Transformationsmodul macht es möglich, dass jede vorhandene Engine als Eingabe-Engine für virtuelle Transformationen dienen kann. Lediglich für solche, die als Ausgabe-Engine verwendet werden sollen, ist die Entwicklung eines Zugriffsmoduls vonnöten.

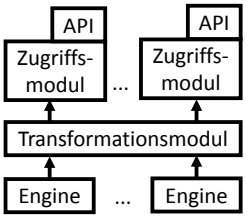



**Abbildung 5.12:** Virtuelle Transformationen

### Transformationsmodul

Die Aufgabe des Transformationsmoduls ist es, Eingabedaten mithilfe der angebundenen Engines zu lesen, diese zu filtern, zu transformieren, zu aggregieren und das Ergebnis in Form eines Cursors oder eines einzelnen Datensatzes den Zugriffsmodulen bereitzustellen. Zur Realisierung dieses Moduls sind verschiedene Techniken denkbar, die bereits vorher in diesem Kapitel präsentiert wurden, beispielsweise mittels Apache Spark. Hadoop bietet sich in diesem Falle nicht an, da MapReduce die Ergebnisse stets persitent schreibt. Des Weiteren gibt es noch eine primitive Möglichkeit, virtuelle Transformationen durchzuführen, nämlich direkt auf den Datenobjekten, welche die Eingabe-Engine liefert, ohne die Verwendung eines

Frameworks. In letzterem Falle ist jedoch keine verteilte Berechnung möglich und Optimierungen sind schwieriger. Betrachten wir aber dennoch zunächst eine Plattform, welche nicht auf einem Datenverarbeitungsframework basiert. Ein generischer Ansatz für das Transformationsmodul sieht wie folgt aus:

1. Lesen aller Daten mittels der Eingabe-Engine
2. Für jeden Datensatz:
   2.1. Filterbedingung überprüfen; bei Nichterfüllung mit dem nächsten Datensatz fortfahren
   2.2. Attribut-Mapping durchführen; dabei Gruppen bilden
3. Für jede Gruppe ein Endergebnis berechnen

Die in Schritt 3 erzeugten Endergebnisse können in einer Programmvariable als Liste, Cursor, Hash-Map oder Ähnliches vorliegen. Über das Zugriffsmodul kann anschließend über die Werte dieser Variable iteriert oder auf ein Element gezielt zugegriffen werden. Vor allem bei einem gezielten Zugriff fällt auf, dass das vorgestellte Vorgehen sehr ineffizient sein kann. Auch wenn lediglich ein einziger Datensatz abgefragt wird, wird die Transformation auf dem kompletten Datenbestand ausgeführt. Daher ist es sinnvoll, dass Zugriffsparameter – etwa die ID des abzurufenden Datensatzes – dem Transformationsmodul zur Verfügung gestellt werden. Das Modul kann diese Information nutzen, um nur diejenigen Basisdaten zu verarbeiten, die für den jeweiligen Zugriff benötigt werden. Betrachten wir im oben stehenden Beispiel, in dem die Personen aus Kaiserslautern von MongoDB über eine HBase-Schrittstelle zur Verfügung gestellt werden, einen gezielten Zugriff auf eine virtuelle Ergebniszeile mittels `GET 'P1'`. Nun ist es vonnöten, diese ausgabeorientierte Filterbedingung in eine eingabeorientierte zu überführen. Dies geschieht über die Attribut-Mappings im vorliegenden NotaQL-Skript. Der Zugriff `GET 'P1'` fordert den Ausgabedatensatz mit der Row-ID `P1`. Es liegt also folgende Bedingung vor: `OUT._r = 'P1'`. Bei Betrachtung des NotaQL-Skriptes wird klar, woher dieses Ausgabeattribut seine Werte bezieht, nämlich aus dem Eingabeattribut `IN._id`. Daher kann die Filterbedingung übersetzt werden in `IN._id = 'P1'`. Über eine Und-Verknüpfung mit der im NotaQL-Skript definierten Filterbedingung (der Ort soll Kaiserslautern sein) ergibt sich eine optimierte Ausführung. In der vorliegenden virtuellen Transformation erfolgt bei einem gezielten Zugriff auf ein Ausgabeelement nur ein einziger Zugriff auf ein Eingabeelement. Hierdurch wurde erreicht, dass wir mit HBase-Zugriffsmethoden effizient auf MongoDB-Daten zugreifen können. Bei der Portierung von Anwendungen, die ein bestimmtes Datenbanksystem voraussetzen, können so andere Datenbanksysteme angebunden werden, ohne den Anwendungscode verändern zu müssen.

Im allgemeinen Fall können jedoch ausgabeorientierte Filterkriterien nicht immer in eingabeorientierte Prädikate übersetzt werden. Wenn Schlüssel oder Attributwerte zusammengesetzt, gesplittet oder anderweitig modifiziert werden, ist es nicht immer möglich, diese Modifikation rückgängig zu machen. Auch bei Selektionen auf Ausgabefeldern, deren Werte mittels Aggregatfunktionen berechnet wurden, ist die Übersetzung nicht möglich.

Um ein Beispiel zu nennen, nehmen wir eine Transformation, die zu jedem Ort die Anzahl der dort wohnenden Personen berechnet. Eine Abfrage, wie viele Personen in einem bestimmten Ort sind, kann leicht über die gerade vorgestellte Art optimiert werden. Allerdings muss bei einer Anfrage, für welche Orte mindestens einhundert Personen gespeichert sind, die Transformation auf den kompletten Basisdaten ausgeführt werden.

Kommt zur Realisierung der Transformationsplattform für virtuelle NotaQL-Transformationen ein Datenverarbeitungsframework wie zum Beispiel Apache Spark zum Einsatz, müssen zuvor genannte Optimierungen nicht zwangsläufig manuell implementiert werden. Spark ist in der Lage, Folgen von Filteroperationen zusammenzufassen und Selektionen möglichst früh auszuführen. Eine geeignete Strategie ist es, dass alle Berechnungen auf RDDs basieren, um Spark möglichst viel Optimierungspotential zu bieten. Das heißt, Eingabeengines bieten einen Zugriff auf die gespeicherten Basisdaten als RDD an, das Transformationsmodul filtert, transformiert und aggregiert auf diesem RDD wie im NotaQL-Skript beschrieben, und ein Zugriffsmodul führt eine abschließende Filteroperation auf dem RDD durch, welche durch die Art des Zugriffs über die API definiert wird. Möchte eine Anwendung über das komplette Ergebnis iterieren, fällt letztere Filteroperation weg. Bei ID-basierten Zugriffen sowie bei Filtern über Sekundärattribute gilt es, diese im Zugriffsmodul in entsprechende Spark-Filterbedinungen zu übersetzen.

### Zugriffsmodul

Ein Zugriffsmodul bietet einen Zugriff auf ein Transformationsergebnis nach außen. Für Anwendungen, die über eine API mit einem Zugriffsmodul für ein gewisses System kommunizieren, soll es so aussehen, als ob sie tatsächlich mit ebendiesem System arbeiten. Daher ist das Ziel bei der Entwicklung eines solchen Moduls, die native API möglichst vollständig zu imitieren. Dies kann auf zwei Arten erfolgen. Zum einen kann der Modulentwickler die native API eines Systems hernehmen und modifizieren. In jeder Methode, in der Daten gelesen werden können, muss eine Fallunterscheidung eingebaut werden, die überprüft, ob das Lesen auf der echten physischen Datenquelle basiert oder auf einer virtuellen. In ersterem Falle bleibt das weitere Vorgehen in der Methode wie gehabt, im zweiten Fall wird der Aufruf an das Transformationsmodul weitergeleitet. Der Nachteil dieser Art ist, dass der Entwickler in einen existierenden Programmcode eingreifen muss. Nicht immer sind diese Codequellen frei verfügbar und gut verstehbar. Des Weiteren muss die Anpassung des Codes beim Erscheinen einer neuen Schnittstellenversion immer wieder von neuem erfolgen. Die alternative und elegantere Art ist es daher, eine eigene API zu implementieren, welche die Schnittstellenbeschreibung der ursprünglichen API erfüllt. Viele NoSQL-Datenbanken bieten dazu passende Interface-Klassen an, welche es zu implementieren gilt. Die meisten Methodenaufrufe können dabei einfach an die bisherige API delegiert werden. Lediglich in Methoden, die auch virtuelle Datenquellen betreffen können, muss die oben beschriebene Fallunterscheidung erfolgen, um entweder native oder virtuelle Zugriffe durchzuführen. Erstere erfolgen wieder über eine Delegation zu den nativen API-Methoden, zweitere über das NotaQL-Transformationsmodul.

Der folgende Code-Abschnitt zeigt, wie ein gezieltes Lesen einer bestimmten Zeile einer HBase-Tabelle erfolgt:

```
Configuration config = HBaseConfiguration.create();
HTable table = new HTable(config, "personen");
Get g = new Get(Bytes.toBytes("P1"));    /* Row-ID */
Result result = table.get(g);
```

Die zuvor erwähnte Fallunterscheidung erfolgt in der gezeigten HBase-API in der Methode `get` der Klasse `HTable`:

```
1  public class HTable implements HTableInterface {
2   private HTable original = null;
3   private byte[] virtualTableName = null;
4
5   public HTable(Configuration conf, String tableName){
6    if(isVirtualTable(tableName)) {      /* virtuell */
7     this.virtualTableName = tableName;
8    } else {                             /* physisch */
9     original = new org.apache.hadoop.hbase.client
10                  .HTable(conf, tableName);
11   }
12  }
13
14  public Result get(Get get) {
15   if(this.virtualTableName != null) {  /* virtuell */
16    Item item = TransfModul.get(this.virtualTableName,
17      "OUT._r = '"+Bytes.toString(get.getRow())+"'";
18    return itemToResult(item);
19   } else {                             /* physisch */
20    original.get(get);
21   }
22  }
23 }
```

In dem gezeigten und den weiteren Konstruktoren der `HTable`-Klasse wird ein tatsächliches `HTable`-Objekt erzeugt, sofern es sich um eine physische Tabelle handelt. Bei einer virtuellen Tabelle darf dies nicht geschehen, da sonst ein Fehler auftritt, weil die Tabelle mit dem gegebenen Namen nicht existiert. Die im Code erwähnte Methode `isVirtualTable` muss implementiert werden. Sie schaut in den Metadatenkatalog der NotaQL-Plattform, ob eine virtuelle Tabelle mit dem gegeben Namen existiert. Beim Einreichen eines NotaQL-Skripts wird dieses Skript zusammen mit dem in der Ausgabe-Engine angegebenen Namen in den Metadaten-Katalog eingefügt. In Zeile 16 wird das Transformationsmodul aufgerufen. Diesem wird der Name der virtuellen Tabelle sowie das Prädikat, welches aus dem `Get`-Objekt erzeugt wird, übergeben. Es ist zu beachten, dass die Schnittstelle des Transformationsmoduls generisch ist, daher kann diese nicht mit `Get`-Objekten und anderen HBase-spezifischen Parametern arbeiten, sondern nur mit allgemenen Filterspezifikationen. Wie in Zeile 18 zu sehen ist, muss der Rückgabewert der Methode wieder in ein Objekt der HBase-spezifischen Klasse `Result` gewandelt werden.

In den anderen hier nicht abgedruckten Methoden wird analog vorgegangen. Manche Methoden können Aufrufe generell immer an die zugrundeliegende native API delegieren, bei anderen Methoden ist eine Fallunterscheidung vonnöten, und bei wieder anderen Methoden muss eine Exception geworfen werden. Letzteres ist zum Beispiel der Fall, wenn ein schreibender Aufruf auf eine virtuelle Tabelle erfolgt. Weitere Anpassungen sind in Klassen, die sich auf die Metadaten beziehen, notwendig, in HBase bespielsweise `HBaseAdmin`. Die dort befindliche Methode `tableExists` muss nun auch `true` zurückliefern, wenn die Tabelle zumindest virtuell existiert. Die Methode `listTables` sollte die Namen der virtuellen Tabellen aus dem Metadatenkatalog zusammen mit den physischen Tabellen ausgeben.

### Bewertung

Das Ziel virtueller Transformationen ist es, dass Benutzer die Mächtigkeit von NotaQL ausnutzen können, ohne die Transformationen tatsächlich komplett und materialisiert durchführen zu müssen. Das ist zum einen sinnvoll, wenn komplexe Abfragen benötigt werden, die API aber nur primitive Anfragen erlaubt. Über die HBase-API sind beispielsweise keine Aggregationen möglich. Möchte ein Benutzer jedoch in einer Anwendung Summen oder Ähnliches berechnen, kann er eine virtuelle Tabelle mittels eines NotaQL-Skripts erstellen und diese im Anschluss über die API abfragen. Ein anderer Anwendungsfall sind systemübergreifende virtuelle Transformationen. Sie fungieren als eine Art Wrapper zwischen verschiedenen Datenbanksystemen. Während in der Anwendung die API eines ganz bestimmten Systems verwendet wird, kann das darunterliegende Datenbanksystem ein völlig anderes sein. Auch ein nachträglicher Wechsel auf ein anderes System oder die Verbindung verschiedener Quellen mittels Join-Engines ist möglich.

Durch den vorgestellten Ansatz wird es auch möglich, Eingabeaufforderungen und graphische Verwaltungsprogramme für spezielle Datenbanksysteme für beliebige andere Systeme zu öffnen und um eine Unterstützung für Sichten zu erweitern. Anders als bei relationalen Datenbanksystemen bieten die meisten NoSQL-Datenbanken diese Funktionalität nicht von Haus aus. Üblicherweise ist dazu kein großer Anpassungsaufwand vonnöten, da lediglich die Bibliotheken der nativen API durch die der Nota-QL-basierten API ausgetauscht werden müssen. Lediglich für die Ergänzung der Programme um neue Funktionen, wie zum Beispiel ein Befehl zur Erzeugung einer NotaQL-Sicht in der HBase-Shell [Bha15], sind aufwändigere Anpassungen vonnöten.

Da virtuelle Transformationen nicht vorberechnet werden, sind die abgefragten Ergebnisse stets aktuell. Auf der anderen Seite heißt dies aber auch, dass die eigentliche Transformation erst bei einem Zugriff auf ein Ergebnis angestoßen wird. Dauert dieser Vorgang mehrere Sekunden oder gar noch länger, ist der Ansatz nicht praktizierbar. Mittels den oben präsentierten Predicate-Pushdown-ähnlichen Optimierungen können gezielte Zugriffe in vielen Fällen deutlich beschleunigt werden. Aggregationen, die auf großen Datenmengen basieren, benötigen aber dennoch zu lange.

Eine geschickte Strategie ist es, virtuelle Transformationen nur dann einzusetzen, wenn die Zugriffszeit im tolerierbaren Bereich ist. Sobald aufgrund wachsender Datenmengen die Zugriffszeiten länger werden, kann die virtuelle Transformation durch eine materialisierte Transformation mittels MapReduce, Spark oder Ähnlichem ersetzt werden. Weder muss dazu das NotaQL-Skript noch die Anwendung angepasst werden. Der einzige Unterschied ist, dass das Transformationsergebnis nun tatsächlich auf einem physischen Medium liegt. Ab nun gilt es zu beachten, dass die Ergebnisse veralten können und dass daher die Transformation in regelmäßigen Abständen erneut erfolgen sollte.

Virtuelle Transformationen sind ein geeigneter Weg, Transformationsketten zu realisieren. Dadurch, dass ein System-spezifisches Zugriffsmodul eine API bereitstellt, die einen Zugriff auf virtuelle Transformationsergebnisse ermöglicht, können Eingabe-Engines ebendiese API nutzen, um Daten einzulesen. Anders als mit den in Kapitel 4.10 vorgestellten virtuellen Engines, ist es mit dem in diesem Abschnitt präsentierten Ansatz auch möglich, dass Anwendungen auf Zwischenergebnisse, die zwischen zwei NotaQL-Transformationen anfallen, zugreifen, ohne dass diese physisch gespeichert werden müssen. Kann man auf diese Eigenschaft verzichten, sollten dem Ansatz jedoch virtuelle Engines vorgezogen werden, da diese ein höheres Optimierungspotenzial bieten.

Der Entwicklungsaufwand einer Plattform für virtuelle Transformationen ist höher als bei den zuvor präsentierten Ansätzen, da die Plattform nicht vollständig generisch entwickelt werden kann. Stattdessen muss für jede zu unterstützende Ausgabe-Engine ein Zugriffsmodul implementiert werden.

## 5.7 Vergleich der Ansätze

In diesem Kapitel wurden sechs Ansätze vorgestellt, die sich für die Realisierung einer NotaQL-Ausführungsplattform eignen. Weitere Ansätze, z. B. eine Ausführung mittels Apache Flink [Apa15b; Emd16], eine Übersetzung in Pig-Skripte [Ols+08] oder in Aggregation-Pipeline-Definitionen für MongoDB [Mona] sind analog zu den präsentierten Ansätzen. Zum Abschluss dieses Kapitels möchten wir die vorgestellten Ansätze auf die zu Beginn erwähnten Eigenschaften Geschwindigkeit, Verteiltheit, Mächtigkeit, Engine-Unterstützung und inkrementelle Berechenbarkeit analysieren und die Ansätze gegeneinander vergleichen.

Tabelle 5.2 gibt einen Überblick über die vorgestellten Ansätze. Die Bewertungen gehen von -- (am schlechtesten) bis ++ (am besten). Ein ○ steht für eine mittelgute Bewertung, N/A bedeutet, dass das Kriterium für den jeweiligen Ansatz nicht anwendbar ist.

| | Geschw. | Verteiltheit | Mächtigkeit | Engines | Inkr. |
|---|---|---|---|---|---|
| **MapReduce** | ○ | ++ | ++ | + | + |
| **Spark** | + | ++ | ++ | ++ | ++ |
| **Pregel** | + | ++ | – | + | ++ |
| **SQL** | ○ | ○ | ++ | + | ○ |
| **SQL/EAV** | ++ | -- | ++ | - | + |
| **Virtuell** | N/A | ++ | ++ | ○ | N/A |

**Tabelle 5.2:** Bewertung der Ansätze zur NotaQL-Ausführung

**Geschwindigkeit** Um die Geschwindigkeit adäquat bewerten zu können, sind vergleichende Performanztests vonnöten. Diese werden im Rahmen dieser Arbeit jedoch nicht durchgeführt, da es kein einheitliches Testszenario gibt, um die Geschwindkeit eines Ansatzes im Ganzen beurteilen zu können. Ein wichtiger Faktor, der die Performanz beeinflusst, ist die Komplexität der NotaQL-Transformation. Je nachdem, ob Joins, Transformationsketten, Kanten-Traversionen, Aggregationsfunktionen oder flexible Schemata zum Einsatz kommen, sind einige Ansätze geeigneter als andere. Deshalb verzichten wir auf konkrete Messungen und bewerten die Geschwindigkeit lediglich danach, in wie fern die genannten Arten von Transformationen effizient ausgeführt werden können. Weitere Faktoren, die die Geschwindigkeit einer Transformation beeinflussen, sind die zugrundeliegenden Daten, die verwendeten Engines und inwiefern eine Engine in der Lage ist, Filter und weiteres direkt an das Datenbanksystem weiterzuleiten.

Die höchste Geschwindigkeit ist beim SQL-basierten EAV-Ansatz möglich, da bei diesem die Daten in einem effizient zu durchsuchenden Format vorliegen. Dadurch werden nicht nur Selektionen und Aggregationen effizient unterstützt, sondern auch flexible Schemata. Beim klassischen SQL-basierten Ansatz sind ebendiese flexiblen Schemata der Grund für eine Abwertung. Komplexe Anfragen können nicht eins zu eins in eine SQL-Anfrage übersetzt werden, sondern erfordern die wiederholte Ausführung einzelner `INSERT`- oder `ALTER TABLE`-Befehle. Insgesamt haben SQL-basierte Ansätze jedoch den Vorteil, dass die erzeugten Anfragen gut optimiert und Indexe ausgenutzt werden können, was vor allem für Joins und Transformationsketten hilfreich ist. MapReduce, Spark und Pregel sowie andere Datenverarbeitungs- und Graph-Frameworks liegt ein weniger starres Datenmodell zugrunde, was Optimierungen nicht in dem Ausmaß wie in SQL unterstützt. Bei MapReduce gibt es zudem eine Abwertung, da Transformationsketten nur durch persistentes Speichern der Zwischenergebnisse möglich sind. Bei virtuellen Transformationen kann die Geschwindigkeit nicht beurteilt werden. Die Einreichung eines NotaQL-Skriptes erfolgt quasi in Nullzeit, dafür sind Anfragen auf das virtuelle Transformationsergebnis mit einem erhöhten Aufwand verbunden.

**Verteiltheit** Die Frameworks MapReduce, Spark und Pregel sind für verteilte Berechnungen gemacht. Sie nutzen nicht nur die Lokalität der Daten aus, sie verteilen auch Berechnungen auf die verschiedenen Rechenknoten, minimieren den Kommunikationsaufwand und kümmern sich um Fehlerbehandlungen. Bei virtuellen Transformationen kann ebenfalls die Verteiltheit in vollem Maße unterstützt werden, beispielsweise wenn auch hier Apache Spark zum Einsatz kommt. Der SQL-basierte Ansatz bietet zunächst keine Unterstützung für verteilte Berechnungen, wenn die Datenspeicherung in einem nicht-verteilten relationalen Datenbanksystem erfolgt. Da sich der Ansatz jedoch auch auf NewSQL-Datenbanken [Gro+13] sowie auf SQL-Schnittstellen zu NoSQL-Datenbanken [Apad; Hiv] anwenden lässt, ist durch Verwendung dieser Technologien eine verteilte Speicherung und Ausführung möglich. Der SQL/EAV-Ansatz kann von diesen Technologien allerdings weniger profitieren, da zur Ausführung von Transformationen eine sehr hohe Zahl an Verbundoperationen nötig sind. Die Verbund-Geschwindigkeit leidet unter der verteilten Speicherung enorm.

**Mächtigkeit** Bei der Realisierung jedes hier vorgestellten Ansatzes wurde darauf geachtet, dass die Sprache NotaQL möglichst in ihrer vollen Mächtigkeit unterstützt wird. Bei den meisten Ansätzen ist dies möglich, auch wenn die Performanz manchmal darunter leidet. Graph-Frameworks wie Pregel sind dagegen lediglich in der Lage, solche Transformationen auszuführen, die einen Graphen auf der Stelle bearbeiten.

**Engine-Unterstützung** Der Spark-basierte Ansatz ist der einzige, welcher neben klassichen Engines auch Streaming-Engines unterstützt. Bei MapReduce, Pregel, SQL und virtuellen Transformationen sind keine Datenstromtransformationen möglich. Die wenigsten Engines werden im EAV-Modell unterstützt, da die Daten entweder als EAV-Tabelle vorliegen müssen oder es eine Abbildung auf eine solche geben muss. Der Ansatz für virtuelle Transformationen erhält eine leichte Abwertung, da für jede Ausgabe-Engine ein eigenes Zugriffsmodul entwickelt werden muss, bevor sie eingesetzt werden kann.

**Inkrementelle Berechnungen** Theoretisch sind inkrementelle Ausführungsweisen unabhängig von der verwendeten Ausführungsplattform. Jedoch werden bei manchen Ansätzen nicht alle inkrementellen Ansätze unterstützt. Bei MapReduce gibt es keine Trigger-Unterstützung und SQL verwaltet in der Regel keine Zeitstempel für Spaltendaten. Im nächsten Kapitel gehen wir darauf detaillierter ein. Bei virtuellen Transformationen sind inkrementelle Ansätze nicht anwendbar, da Ergebnisse, auf die bei diesen Ansätzen zurückgegriffen werden, nicht persistent gespeichert sind.

# 6 Inkrementelle Wartbarkeit

Die in dieser Arbeit vorgestellte Sprache NotaQL ist deklarativer Natur. Dies bedeutet, dass der Nutzer mit einem NotaQL-Skript definiert, *was* eine Transformation durchführen soll und nicht *wie* der Transformationsprozess genau ablaufen soll. Ein einmal definiertes Skript kann auf verschiedenartigen Plattformen, die im vorherigen Kapitel vorgestellt wurden, ausgeführt werden. Unabhängig von der gewählten Ausführungsart ist bei gleichen Eingaben die Ausgabe eines gegebenen NotaQL-Skriptes stets gleich. Dies macht es möglich, zwischen den Ausführungsarten zu wechseln, um deren verschiedenen Vorteile zu kombinieren.

In diesem Kapitel stellen wir eine weitere Dimension der Ausführung vor, nämlich die der *inkrementellen Wartbarkeit* von NotaQL-Transformationen. Die vorgestellten Ansätze basieren auf Ansätzen zu selbstwartbaren Sichten in relationalen Datenbanken [GJM96]. Genau wie die unterschiedlichen Ausführungsarten im vorherigen Kapitel, beschreiben die nun folgenden Ansätze, *wie* eine Transformation intern abläuft. Die inkrementellen Ansätze sind zum einen *optional* und zum anderen *orthogonal* zu den Ausführungsarten. Optional bedeutet, dass man auf eine inkrementelle Ausführung verzichten und stattdessen auch immer auf eine vollständige Neuberechnung ausweichen kann. Orthogonal bedeutet, dass die inkrementellen Ansätze prinzipiell in Kombination mit jeder Ausführungsart zum Einsatz kommen können. Einige inkrementellen Methoden erfordern jedoch Voraussetzungen, die nicht bei allen Arten gegeben sind. Auf diese weisen wir an gegebener Stelle hin.

Die Idee der inkrementellen Berechnung basiert auf der Tatsache, dass viele Datentransformationen in der Regel in periodischen Zeitintervallen erneut durchgeführt werden. Nun gilt es, bei einer wiederholten Ausführung ein erneutes vollständiges Einlesen der Basisdaten zu vermeiden. Bei NotaQL-Transformationen, die lediglich einmal ausgeführt werden, beispielsweise zur Schema-Migration oder Schema-Evolution, sind die Ansätze nicht anwendbar. Auch ergeben sie in Verbindung mit virtuellen Transformationen keinen Sinn, da diese keine Ergebnisse persistent ablegen und

stets die aktuellen Daten liefern. Die materialisierten Ausführungsarten hingegen sind Stapel-basiert. Sie erzeugen beim Einreichen eines NotaQL-Skriptes eine Ausgabe, welche auf dem aktuellen Stand der Basisdaten basiert. Ändern sich die Basisdaten, veralten Ergebnisse und müssen erneut ausgeführt werden. Je häufiger diese Neuberechnung erfolgt, desto weniger *stale Data*, also veraltete Ergebnisse, liegen vor.

Eine NotaQL-Transformation ist *inkrementell* (oder *selbstwartbar*), wenn das Transformationsergebnis lediglich mit den Änderungen seit einer vormaligen Ausführung sowie mit dem vorherigen Ergebnis berechnet werden kann. Das Gegenteil von einer inkrementellen Ausführung ist eine vollständige Neuberechnung. Oberstes Ziel eines jeden hier vorgestellten Ansatzes ist es, dass das von ihm erzeugte Ergebnis äquivalent mit dem einer vollständigen Berechnung ist. Der Nutzen aller vorgestellten Arten soll eine kürzere Transformationsdauer sein. Dies folgt aus dem Fakt, dass nur die Änderungen seit einer vorherigen Berechnung und nicht die kompletten Basisdaten gelesen werden müssen. Da die Dauer jedoch von der Komplexität eines NotaQL-Skriptes, den Basisdaten, dem Anteil an geänderten Daten und anderen Faktoren abhängt, kann keine pauschale Aussage über den tatsächlichen Zeitgewinn getroffen werden. Zudem ist es durchaus möglich, dass manche inkrementelle Berechnungen in einigen Fällen länger dauern können als volle Neuberechnungen. In solchen Fällen ist eine inkrementelle Ausführung nicht sinnvoll.

Nachdem in der ersten Hälfte dieses Kapitels die verschiedenen Arten zur Änderungserfassung aus Kapitel 2.3 behandelt werden, wird in der zweiten Hälfte eine Plattform für inkrementelle NotaQL-Transformationen vorgestellt, die die verschiedenen Ausführungsarten und inkrementellen Ansätze kombiniert. Diese Plattform beinhaltet eine Advisor-Komponente, welche dem Nutzer basierend auf Heuristiken und automatisierten Lernmethoden die Entscheidung für die geeignetste Ausführungsart abnimmt.

## 6.1 Änderungserfassung in NotaQL-Transformationen

Inkrementelle Berechnungen erfolgen in zwei Schritten: Im ersten Schritt geht es darum, Änderungen in den Basisdaten seit einer vorherigen Berechnung zu ermitteln. Wir bezeichnen diese Änderungen $\Delta$ (Delta) und $\nabla$ (Nabla). Ersteres ist die Menge der Datensätze, die seit der vorherigen Berechnung neu hinzugekommen sind, zweiteres die Datensätze, die seitdem entfernt wurden. Werteänderungen können als Löschung mit anschließendem Neueinfügen angesehen werden (siehe Abbildung 6.1). Die in diesem Kapitel vorgestellten Ansätze verfolgen alle das Ziel, ebendiese Mengen $\Delta$ und $\nabla$ zu bestimmen.

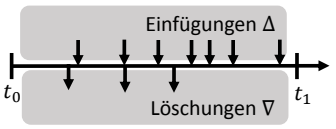



**Abbildung 6.1:** Änderungserfassung

Im zweiten Schritt der inkrementellen Berechnung geht es darum, die Änderungen korrekt in das bereits vorliegende vorherige Transformationsergebnis einfließen zu lassen. Dazu verfolgen wir in diesem Abschnitt hauptsächlich die Idee der *Overwrite-Installation* [Jör+11a], in welcher das vorherige Ergebnis gelesen und nach der Kombination mit den Änderungen das neue Ergebnis berechnet wird. Dieses neue Ergebisnis überschreibt das bisherige. Der Ansatz eignet sich für NotaQL-Transformation besser als die

alternative *Increment-Installation*. In dieser werden nämlich Transformationsauswirkungen direkt auf der Stelle mit dem vorhergen Ergebnis kombiniert. Die dazu erforderliche Inkrement-Operation ist nicht für alle Aggregatfunktionen möglich und sie ist in der Regel langsamer als ein sequenzielles Schreiben – was bei der Overwrite-Installation erfolgt. Zudem erfordert dieser Ansatz Änderungen im Programmcode von Ausgabe-Engines und es ist zu betonen, dass nicht alle den Engines zugrundeliegenden Systeme inkrementelle Operationen unterstützen.

Der folgende Ablauf beschreibt im Groben, wie NotaQL-Transformationen inkrementell ausgeführt werden. $B(t_0)$ steht für den Zustand der Basisdaten bei einer vorherigen Ausführung zum Zeitpunkt $t_0$ und $B(t_1)$ zum Zeitpunkt $t_1$ der inkrementellen Neuberechnung. Das Ergebnis der vorherigen Berechnung ist $E(t_0)$. Nun gilt es, das neue Ergebnis $E(t_1)$ zu bestimmen.

1. $\Delta = B(t_1) \setminus B(t_0)$ (eingefügte Datensätze)
2. $\nabla = B(t_0) \setminus B(t_1)$ (gelöschte Datensätze)
3. $\delta = NotaQL(\Delta)$ (positive Ausgabefragmente)
4. $\wp = NotaQL(\nabla)$ (negative Ausgabefragmente)
5. $E(t_1) = E(t_0) \circ \delta \circ \wp^{-1}$

Die Berechnung der positiven sowie negativen Ausgabefragmente erfolgt weitestgehend nach der gewohnten NotaQL-Ausführungsart. Das heißt, dass für die Datensätze in $\Delta$ bzw. $\nabla$ die Eingabefilter und die Attribut-Mappings angewendet werden. Lediglich die Auswertung der Aggregation erfolgt erst später in der hier angegebenen Operation $\circ$. Diese Operation nimmt eine Menge von Ausgabefragmenten entgegen und führt eine Aggregation gemäß eines vorliegenden NotaQL-Skriptes für alle Werte durch, die in ein und die selbe Zelle fallen, also beispielsweise die Werte für ein bestimmtes Feld in einem bestimmten MongoDB-Ausgabedokument. Erfolgt im NotaQL-Skript keine Aggregation, fällt nie mehr als ein Wert in eine Zelle, da ansonsten die Transformation ungültig wäre. In diesem Fall sorgt die $\circ$-Operation für ein gegenseitiges Auslöschen von gelöschten Datensätzen aus $E(t_0)$ mit negativen Ausgabefragementen $\wp$. Es ist von hoher Relevanz, dass $\Delta$ und $\nabla$ ganze Datensätze beinhalten und nicht nur geänderte Zellen. Andernfalls wäre es unmöglich, Filter-Bedingungen, Attribut-Mappings und Aggregationen auf unveränderten Zellen des geänderten Datensatzes durchzuführen.

Die in Schritt 4 erzeugten negativen Ausgabefragmente werden vor der Aggregation invertiert. Diese Information über Invertiertheit sorgt beim Ausführen einer Aggregatfunktion mittels $\circ$ für ein umgekehrtes Verhalten als bei nicht invertierten Datenwerten. Für die Summenfunktion erfolgt hier also eine Subtraktion, bei Listenkonstruktoren die Entfernung eines Elements aus einer Liste, und so weiter.

Abbildung 6.2 zeigt, wie sich die NotaQL-Ausführung bei inkrementellen Berechnungen verändert. Der Filter-Prozess erfolgt sowohl für eingefügte als auch für gelöschte Werte. Erfüllen neue Datensätze die Filter-Bedingung, werden die Änderungen in das Ergebnis eingebracht. Erfüllen

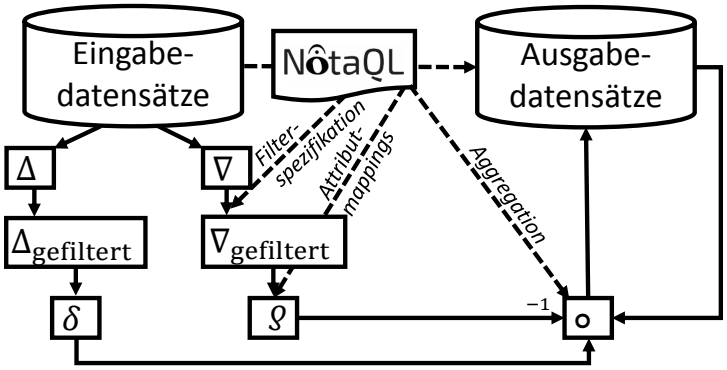



**Abbildung 6.2:** Inkrementelle Ausführung eines NotaQL-Skripts

gelöschte Datenwerte das Filterkriterium, heißt dies, dass zuvor vorgenommene Auswirkungen auf das Endergebnis kompensiert werden müssen. Die in der $\circ$-Operation ausgeführte Aggregation kombiniert positive und invertierte negative Ausgabefragment mit dem vorherigen Ergebnis.

Es fällt auf, dass das gezeigte Vorgehen nur möglich ist, wenn eine Aggregatfunktion mit invertierten Datenwerten umgehen kann und wenn die Aggregatfunktion assoziativ und kommutativ ist. Ersteres Kriterium ist bei Minimum- und Maximumfunktionen nicht gegeben. Es ist nicht möglich, basierend auf einem vorherigem Maximum und dem Entfernen ebendieses Maximalwertes das neue Maximum zu berechnen. Die Assoziativität ist bei Durchschnittsberechnungen nicht gegeben. Daher sind hier Hilfsstrukturen vonnöten. Listen-Konstruktoren und String-Konkatenationen erfüllen das Kommutativitätskriterium nicht. Das sorgt dafür, dass eine Ergebnisliste nach einer inkrementellen Berechnung möglicherweise anders geordnet sein kann, als es bei einer vollständigen Neuberechnung der Fall wäre. Da bei NotaQL-Transformationen die Ordnung in der Regel ohnehin als nicht-deterministisch angenommen wird, können wir die Resultate inkrementeller und vollständiger Berechnungen in diesem Fall dennoch als äquivalent bezeichnen.

Wenn wir als Basis eine Ausführungsplattform hernehmen, die auf einer der vorgestellten Ausführungsarten aus dem vorherigen Kapitel basiert, kann diese für die gezeigten Schritte 3 und 4 zum Einsatz kommen, um die Ausgabefragmente basierend auf den Änderungen in den Basisdaten zu berechnen. Dies erfolgt gemäß dem vorliegenden NotaQL-Skript. Auch Schritt 5 kann mit den existierenden Methoden zur Aggregation übernommen werden. Dazu müssen lediglich die Aggregatfunktionen um eine Fallunterscheidung erweitert werden, um invertierte Werte korrekt zu behandeln.

Es zeigt sich, dass sich NotaQL-Skripte leicht inkrementell ausführen lassen, sofern eine Möglichkeit angewandt werden kann, um diejenigen Datensätze zu bestimmen, die sich seit einer vorherigen Berechnung geändert haben. Im Folgenden diskutieren wir verschiedene solche Möglichkeiten zur Änderungserfassung [KR11].

### Zeitstempel-basiert

Die Zeitstempel-basierte Änderungserfassung setzt voraus, dass bei der Datenanalyse Informationen über den Zeitpunkt von Einfügungen, Änderungen und Löschungen von Datenwerten vorliegen. Im Folgenden diskutieren wir drei Faktoren, die für inkrementelle Transformation mittels Zeitstempel-basierter Änderungserfassung von hoher Relevanz sind:

1. Zeitstempelgranularität
2. System-basierte / Anwendungs-basierte Zeiterfassung
3. Verfügbarkeit gelöschter und überschriebener Datenwerte

Unter der im ersten Punkt genannten *Zeitstempelgranularität* versteht man die Dateneinheit, für die ein gegebener Zeitstempel gilt: Eine Zelle, einen Datensatz oder die vollständige Datenquelle. Zur inkrementellen Ausführung von NotaQL-Transformationen sind Datensatz-basierte Zeitstempel am besten geeignet. Im folgenden Beispiel ist ebendiese Zeitstempelgranularität gegeben. Es zeigt ein MongoDB-Dokument, welches den Zeitpunkt seiner letzten Änderung im Feld `geaendert` beinhaltet.

```
{ _id: 1, name: "Anita", firma: "IBM", gehalt: 50000,
  geaendert: ISODate("2013-10-02T01:11:18.965Z") }
```

Im gegebenen Dokument ist nicht zu erkennen, welche Feldwerte zum angegebenen Zeitpunkt geändert wurden. Zur Änderungserfassung ist dies jedoch auch nicht nötig. Mittels des gespeicherten Zeitpunktes der letzten Änderung kann zumindest die Menge $\Delta$ bestimmt werden, indem diejenigen Dokument gesucht werden, deren `geaendert`-Wert zwischen $t_0$ und $t_1$ liegt. Es ist immer empfehlenswert, die obere Grenze auf den Beginn der Berechnung $t_1$ und nicht auf Unendlich zu setzen, da die Berechnung selbst einige Zeit in Anspruch nimmt und somit Dokumente ignoriert werden, welche erst nach diesem Zeitpunkt eingefügt oder modifiziert wurden. Dennoch kann dieses Verfahren zu Problemen führen. Wird beispielsweise ein Dokument zweimal bearbeitet, einmal vor und einmal nach $t_1$, geht es fälschlicherweise nicht in die Berechnung mit ein. Da ohnehin dieses Verfahren aufgrund der Tatsache, dass überschriebene und gelöschte Werte nicht mehr zugreifbar sind, nur Einfügungen und keine Änderungen und Löschungen von Dokumenten unterstützt, gehen wir auf diese Fälle erst später in diesem Kapitel separat ein.

Ist die Zeitstempelgranularität eine Zelle, bedeutet dies, dass zu jeder Zelle, also beispielsweise jedem Attributwert in einem Dokument, ein Zeitstempel annotiert wird. Zur Änderungserfassung ist eine solche feingranulare Unterscheidung nicht nötig, aber auch nicht hinderlich. Es lässt sich nämlich leicht die letzte Änderung eines Datensatzes als das Maximum aus allen seinen Zellen-Zeitstempeln berechnen.

Liegt lediglich ein Zeitstempel über die letzte Änderung einer kompletten Datenquelle vor, kann man an diesem nur erkennen, ob sich seit der letzten Berechnung etwas geändert hat, aber nicht was. Daher ist diese Zeitstempelgranularität nicht ausreichend. Ist die Eingabe einer NotaQL-Transformation eine CSV-Datei und ist der Zeitpunkt der letzten Änderung dieser Datei, welcher durch das Dateisystem verwaltet wird, kleiner als $t_0$,

folgt daraus: $\Delta = \nabla = \emptyset$. Eine erneute Berechnung ist also nicht nötig. Ist der Zeitpunkt der Änderung jedoch größer als $t_0$, muss eine vollständige Neuberechnung erfolgen, da keine Information über die geänderten Datensätze vorliegt.

Beim zweiten Punkt gilt es zu unterscheiden, wer dafür verantwortlich ist, die Zeitstempel an die Daten zu annotieren. Beim soeben genannten CSV-Datei-Beispiel erfolgt die Zeiterfassung durch das Dateisystem. Eine solche *System-basierte Zeiterfassung* findet ebenfalls beim Wide-Column-Store HBase statt. Das Datenbanksystem annotiert automatisch bei jeder Spaltenwertänderung den aktuellen Zeitstempel an die betreffenden Spalten. Beim oben genannten MongoDB-Beispiel handelt es sich um eine *Anwendungs-basierte Zeiterfassung*. Die Anwendung ist dafür verantwortlich, bei jeder Änderung an den Daten den Zeitstempel der letzten Änderung korrekt zu aktualisieren. Dazu muss mit hoher Sorgfalt darauf geachtet werden, dass keine Schreiboperation erfolgt, ohne dass auch der Zeitstempelwert geändert wird. Dabei können spezielle API-Erweiterungen [Gra13] oder die Installation von Triggern hilfreich sein.

Die in MongoDB in jedem Dokument präsente Dokument-ID (`_id`) kann wahlweise vom Benutzer oder vom System generiert werden. Im letzteren Fall ist sie ein 12-Byte-großer Wert, bei dem die ersten vier Byte den Unix-Zeitstempel zum Zeitpunkt der Generierung dieser ID darstellen. Aus der in Hexadezimalschreibweise notierte ID `507f1f77bcf86cd799439011` lässt sich der Zeitstempel 17.10.2012, 21:13:27 Uhr ableiten. Es ist jedoch zu beachten, dass dieser Zeitstempel den Zeitpunkt der ID- und damit der Dokument-Erstellung darstellt und nicht den der letzten Änderung. Daher ist er für inkrementelle NotaQL-Berechnungen nur dann sinnvoll, wenn keine Änderungen an Dokumenten vorgenommen werden, sondern lediglich neue Dokumente eingefügt werden.

Der dritte oben genannte Punkt, der für die Zeitstempel-basierte Änderungserfassung entscheidend ist, ist die *Verfügbarkeit gelöschter und überschriebener Datenwerte*. Eine inkrementelle Berechnung ist nur dann korrekt durchführbar, wenn gelöschte und überschriebene Datenwerte weiterhin verfügbar bleiben. Betrachten wir die drei genannten Beispiele MongoDB, CSV-Datei und HBase, so ist dieses Kriterium nur für HBase erfüllt. Der Wide-Column-Store HBase sowie andere Datenbanksysteme, die ebenfalls die Versionierung von Daten unterstützen und somit Zugriff auf die Historie von Datenwerten liefern, eigenen sich am besten für die Zeitstempelbasierte Änderungserfassung. Andere Systeme sind nicht oder nur begrenzt dazu geeignet. Daher fahren wir in diesem Abschnitt exemplarisch mit einem Ansatz fort, welcher auf dem Versionierungsmodell und dem Datenmodell von HBase basiert. Er lässt sich jedoch auf andere Datenbanksysteme übertragen, die ebenfalls eine Versionierung unterstützen.

#### Zeitstempel-basierte Änderungserfassung in HBase

Der Wide-Column-Store HBase [Apab] verwaltet Zeitstempel für jede Zelle, die Erfassung erfolgt durch das System – also automatisch – und die Historie an vorherigen Werten pro Zelle sind samt ihrer Zeitstempel abrufbar. Letzteres setzt voraus, dass beim Erstellen der Tabelle, in der die Basisdaten für Transformation gespeichert werden, der Konfigurationsparameter `VERSIONS` für die benötigten Spaltenfamilien auf einem ausreichend hohen Wert gesetzt ist. Der Parameter bestimmt die Anzahl der gespeicherten Versionen. Ist der Wert zu klein gewählt, könnten veraltete Datenwerte verloren gehen. Ein sehr hoch gewählter Wert sorgt jedoch für einen enormen Anstieg der gespeicherten Datenmenge. Um jedoch inkrementelle Transformationen in jedem Falle korrekt ausführen zu können, sollte die Anzahl der Versionen eher zu hoch als zu niedrig gewählt werden. Ein regelmäßiges Löschen nicht mehr benötigter alter Versionen kann die Datenmenge dann wieder reduzieren.

Genau wie beim Erstellen einer Tabelle wird in HBase auch beim Durchsuchen einer Tabelle die Anzahl der Versionen konfiguriert. Standardmäßig ist diese `1`, das heißt man erhält zu jeder Zelle lediglich den aktuellen Wert. Da es nicht bekannt ist, wie viele Änderungen seit einer vorherigen Berechnung zum Zeitpunkt $t_0$ an den einzelnen Zellen stattgefunden haben, muss die Anzahl der Versionen beim Durchsuchen auf unendlich gestellt werden. Davon sind jedoch nur zwei Datenwerte pro Zelle von Relevanz: Der aktuelle Wert sowie der Wert, der zum Zeitpunkt $t_0$ gültig war. Da es in HBase nicht möglich ist, lediglich diese beiden Werte für jene Datensätze abzufragen, die sich zwischen $t_0$ und $t_1$ geändert haben, muss diese Suche manuell geschehen. Dazu untersuchen wir zwei Ansätze:

```
scan.setTimeRange(t0, t1)
```

Bei diesem Ansatz werden zwar alle seit dem Zeitpunkt $t_0$ geänderten Datensätze gefunden, jedoch treten zwei unerwünschte Effekte auf: Zum einen beinhalten diese Datensätze lediglich die geänderten Spalten – benötigt werden jedoch alle Spalten. Zum anderen kann in den gelesenen Spalten nicht auf den Wert zugegriffen werden, der zum Zeitpunkt $t_0$ gültig war, da dieser einen Zeitstempel kleiner als $t_0$ besitzt. Der Ansatz kann aber trotzdem zum Erfolg führen, indem man ihn nur dazu nutzt, um die Row-IDs der geänderten Zeilen zu suchen. In einem separaten Schritt werden die vollständigen Datensätze samt aller ihrer Versionen geladen. Dieses Vorgehen ist bei einer großen Anzahl geänderter Zeilen langsamer als der nun folgende alternative Ansatz:

```
scan.setTimeRange(0, t1)
```

Dieses Mal werden alle Datensätze samt aller ihrer Versionen geladen. Während über diese Sätze iteriert wird, werden diejenigen aussortiert, bei denen seit $t_0$ keine Änderung stattgefunden hat. Ist die Anzahl solcher unveränderter Datensätze sehr gering, ist der vorherige Ansatz besser geeignet. Vorteil nun ist jedoch, dass aus den vorliegenden Daten alle benötigten Informationen extrahiert werden können. Nach dem Aussortieren unveränderter Datensätze liegen vollständige Zeilen samt aller Spalten und allen Werteversionen vor. Von Relevanz sind je Spalte zum einen der aktuelle Wert $v_1$, also der Wert mit dem höchsten Zeitstempel $z_1 \leq t_1$. Zum anderen wird

der Wert $v_0$ benötigt, der den höchsten Zeitstempel $z_0 \leq t_0$ besitzt. Für die Zeitstempel gilt stets folgendes, $\perp$ steht dabei für einen undefinierten Wert:

$$0 \leq z_1 \leq t_1$$
$$z_0 = \perp \vee (z_0 \leq t_0 \wedge z_0 \leq z_1)$$
$$z_0 = z_1 \Leftrightarrow z_1 \leq t_0$$

Interpretieren wir die verschiedenen Möglichkeiten für die vorliegenden Zeitstempel weiter:

$z_1 \leq t_0 \Rightarrow$ Spaltenwert $v_1$ seit vorheriger Ausführung unverändert
$z_1 > t_0 \wedge z_0 = \perp \Rightarrow$ Spaltenwert $v_1$ neu eingefügt
$z_1 > t_0 \wedge z_0 \neq \perp \Rightarrow$ Spaltenwert $v_0$ wurde auf $v_1$ geändert

Es fällt auf, dass nicht festgestellt werden kann, ob und wann ein Wert gelöscht wurde. Das liegt daran, dass Löschungen keine Spuren hinterlassen, die beim Lesen der Daten auffallen. Nur wenn wir voraussetzen, dass keine Löschoperationen auf den Daten passieren, ist die Zeitstempel-basierte Änderungserfassung für HBase nutzbar. Dazu muss dafür gesorgt werden, dass Anwendungen Spalten mit leeren Werten $[\,]$ – also leeren Byte-Arrays – überschreiben statt die Werte zu löschen. Nun sind auch Löschungen detektierbar:

$$z_1 \geq t_0 \wedge z_0 \neq \perp \wedge v_1 = [\,] \Rightarrow \text{Spaltenwert } v_0 \text{ gelöscht}$$

Da nun für jede Zelle bekannt ist, ob sie unverändert, neu, geändert oder gelöscht ist, ist der nächste Schritt, alle zu einem Datensatz gehörenden Zellen zusammenzufassen und aus ihnen einen *Delta-Datensatz* und einen *Nabla-Datensatz* zu bilden. Ersterer gehört zu denjenigen Datensätzen, die zum Abschluss der Änderungserfassung die Menge der Einfügungen $\Delta$ bilden, letzterer zu denjenigen, die die Menge der Löschungen $\nabla$ bilden. Wie in Abbildung 6.3 zu sehen, besteht der Delta-Datensatz aus allen unveränderten und eingefügten Zellen sowie den neuen Werten von geänderten Zellen. Der Nabla-Datensatz besteht ebenfalls aus den unveränderten Zellen, und zudem den gelöschten Zellen und den alten Werten von geänderten Zellen. Die ID eines Satzes ist immer im Datensatz enthalten und dient zur Gruppierung der zu einem Datensatz gehörenden Zellen. Besteht ein Datensatz nur aus neu eingefügten oder nur aus gelöschten Zellen, wird kein Nabla- bzw. kein Delta-Datensatz erzeugt. Der Fall, dass ein Datensatz nur aus unveränderten Zellen besteht, kann nicht eintreten, da diese bereits in vorherigen Verarbeitungsschritten aussortiert wurden.

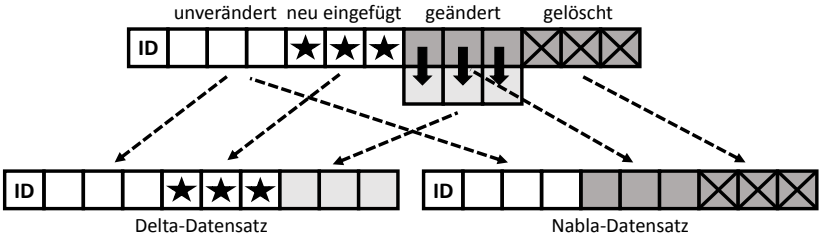



**Abbildung 6.3:** Bildung von Delta- und Nabla-Datensätzen

Als Resultat liegt ein Nabla-Datensatz vor, der den Zustand des Datensatzes zum Zeitpunkt $t_0$ wiederspiegelt. Er entspricht also dem Datensatz, auf dem das vorliegende Ergebnis des vorherigen Transformationsdurchlaufs basiert. Des Weiteren liegt ein Delta-Datensatz vor, welcher den aktuellen Zustand des Datensatzes darstellt. Diesen würde man auch erhalten, wenn man eine klassische Leseoperation auf die Datenbank stellt.

Fasst man alle Delta- bzw. Nabla-Datensätze zusammen, bilden diese die Mengen $\Delta$ und $\nabla$. Sie dienen, wie zu Beginn dieses Kapitels beschrieben, als Eingabe für inkrementelle NotaQL-Transformationen.

#### Andere temporale Datenbanksysteme

Bei anderen temporalen Datenbanken ist ein ähnliches wie das hier beschriebene Vorgehen denkbar. Einige Datenbanken bieten aber auch eine benutzerfreundlichere Methode, um historische Datensätze zu laden. Der *SQL-2011-Standard* erlaubt in der `FOR SYSTEM_TIME`-Klausel die Angabe eines Zeitpunktes in der Vergangenheit [KM12]. Das Datenbanksystem berechnet das Ergebnis einer Anfrage auf Basis des Datenbankzustands, der zu ebendiesem Zeitpunkt gültig war. Des Weiteren können die Zeiträume, von wann bis wann ein Datensatz gültig war, in den Spalten `sys_start` und `sys_end` abgerufen werden. Mit solchen Zugriffsmethoden ist es deutlich einfacher, die Mengen $\Delta$ und $\nabla$ zu bestimmen:

```
Delta = SELECT * FROM basistabelle
        WHERE sys_start > t0
```

Das Prädikat `sys_start > t0` sorgt dafür, dass keine Datensätze gefunden werden, die sich seit der vorherigen Berechnung nicht geändert haben.

```
Nabla = SELECT * FROM basistabelle
        FOR SYSTEM_TIME AS OF TIMESTAMP t0
        WHERE sys_end <= t1
```

Auch in diesem Fall sorgt das Prädikat `sys_end <= t1` dafür, Datensätze auszulassen, die sich nicht mehr verändert haben. Somit besteht $\nabla$ nur aus alten Datensätzen, die mittlerweile gelöscht oder geändert wurden.

### Audit-Columns

Eine Audit-Column ist ein zur Änderungserfassung nutzbares Attribut. Viele Zeitstempel-basierte Ansätze sind Spezialfälle von Audit-Columns, beispielsweise das im vorherigen Abschnitt gezeigte MongoDB-Dokument, welches ein `geaendert`-Feld mit dem Zeitstempel der letzten Änderung beinhaltet. Auch die in relationalen Datenbanken im SQL-2011-Standard [KM12] verwendeten `sys_start`- und `sys_end`-Spalten sind Audit-Columns. Die Versionierung von HBase fällt nicht in diese Kategorie, da dort die Zeitstempel Teil des Datenmodells sind. Audit-Columns sind jedoch klassische Metadaten. In vielen Fällen ist die Anwendung für das Setzen von Audit-Column-Werten verantwortlich. Spezielle APIs, temporale Datenbankerweiterungen oder Trigger können diese Aufgabe jedoch auch übernehmen.

In Audit-Columns müssen jedoch nicht zwangsweise Zeitstempel gespeichert werden, sondern beispielsweise auch die Information, welcher

Benutzer einen Wert geschrieben hat. Auch ist es denkbar, dass eine Audit-Column lediglich einen booleschen Wert beinhaltet, welcher anzeigt, ob ein Datensatz seit einer vorherigen Transformation geändert wurde oder nicht. In diesem Falle ist beim Durchführen einer Transformation darauf zu achten, diese Werte im Anschluss wieder zurückzusetzen. Bei NotaQL-Transformationen kann diese Information hilfreich sein, um die Menge $\Delta$ zu bestimmen, welche die Datensätze enthält, die sich seit der vorherigen Transformation geändert haben. Jedoch ist es unmöglich, die vorherigen Zustände von geänderten und gelöschten Datensätzen abzufragen. Daher betrachten wir diesen Ansatz nicht weiter.

### Snapshots

Liegt bei der Ausführung eines NotaQL-Skriptes ein Schnappschuss der Datenbank vor, welcher zum Zeitpunkt einer vorherigen NotaQL-Transformation angelegt wurde, kann dieser Schnappschuss zusammen mit dem aktuellen Datenbankzustand dazu dienen, die Änderungen in den Basisdaten seit der vorherigen Ausführung zu ermitteln. Im Gegensatz zu vielen anderen Änderungserfassungstechniken muss bei diesem Ansatz das System, aus welchem mit der Eingabe-Engine Daten gelesen werden, keine speziellen Anforderungen erfüllen. Dafür kommt bei diesem Ansatz die Eingabe-Engine doppelt zum Einsatz, zum einen zum Lesen der aktuellen Daten $B(t_1)$, zum anderen zum Lesen des vorherigen Basisdatenzustands $B(t_0)$. Die größten Zeitersparnisse gegenüber einer vollen Neuberechnung sind dann möglich – genau wie bei jeden anderem Ansatz der Änderungserfassung auch –, wenn sich möglichst wenig in den Basisdaten verändert hat. Im ersten Schritt der Snapshot-basierten Änderungserfassung geht es nämlich darum, die Menge derjenigen Datensätze herauszufinden, welche sich seit der vorherigen Berechnung nicht verändert haben. Diese Menge ist in Abbildung 6.4 in der Mitte dargestellt. Es ist die Schnittmenge aus den aktuellen Daten und den Snapshot-Daten. Erst im nächsten Schritt findet basierend auf der verbleibenden Datenmenge die eigentliche Änderungserfassung mittels Mengensubtraktionen statt.

Es ist von großer Wichtigkeit, dass die Schnittmengenbildung auf ganzen Datensätzen basiert und nicht lediglich auf den IDs der Datensätze. Eine Werteänderung innerhalb eines zum Zeitpunkt $t_0$ bereits vorhandenen Datensatzes sorgt bereits dafür, dass dieser nicht mehr Teil der unveränderten Datensätze ist. Stattdessen ist der alte Zustand Element der Menge $\nabla$ und der neue Zustand ein Element von $\Delta$.

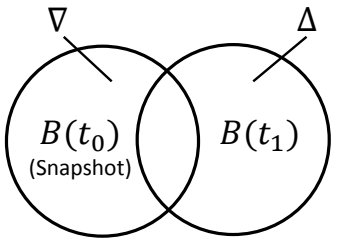



**Abbildung 6.4:** Snapshot

Bei der Verwendung eines primitiven Mengensubtraktionsalgorithmus zur Berechnung von $\nabla$ wird die Datenmenge $B(t_0)$ gelesen und für jeden Datensatz überprüft, ob dieser Datensatz exakt in dieser Form samt aller seiner Attributwerte auch in $B(t_1)$ vorliegt. Ist dies nicht der Fall, ist ebendieser Datensatz teil des Subtraktionsergebnisses. Analog wird in der Gegenrichtung mit der Menge $\Delta$ vorgegangen. Lägen die beiden Eingabemengen auf die gleiche Art und Weise geordnet vor, ließe sich die Subtraktion deutlich effizienter ausführen. Dies ist aber im Allgemeinen nicht der

Fall. Wir können uns jedoch als einfache Optimierung die Tatsache zunutze machen, dass jeder Datensatz eine eindeutige ID besitzt. Bestimmt man zunächst nur die Mengen der IDs von $B(t_0)$ und $B(t_1)$, kann die symmetrische Differenz dieser Mengen für den Subtraktionsalgorithmus nützlich sein. Ist nämlich eine ID Teil ebendieser symmetrischen Differenz, steht fest, dass der vorliegende Datensatz zur Menge $\nabla$ bzw. $\Delta$ gehören muss. Ein Durchsuchen der jeweils anderen Menge ist daher nicht vonnöten. Da jedoch bei inkrementellen Berechnungen typischerweise ein Großteil der Basisdaten unverändert bleibt, ist diese Menge so klein, dass diese Optimierung die Berechnung nur minimal beschleunigt. Andere Ansätze, welche auf Sortierungen, Kompression [LGM96] oder Bloom-Filtern [Blo70] basieren, können jedoch bei der effizienten Berechnung einer Snapshot-Differenz hilfreich sein.

Der Vorteil der Snapshot-basierten Änderungserfassung ist die universale Anwendbarkeit. Viele Datenbanksysteme bieten eingebaute Funktionalitäten zur Erstellung von Snapshots. Oft basieren diese auf den pyhsischen Daten, sodass ein Klonen sehr effizeint vonstatten gehen kann. Sollte ein System keine eingebaute Snapshot- oder Kopierfunktion besitzen, kann der Schnappschuss auch mittels eines simplen Programms oder einer NotaQL-Transformation angelegt werden. Der Nachteil des Snapshot-Ansatzes ist die enorme Größe, die ein Schnappschuss annehmen kann. Dies bedeutet zum einen, dass das Anlegen eines Schnappschusses einige Minuten in Anspruch nehmen kann. Um einen konsistenten Schnappschuss sicherzustellen, ist es in manchen Systemen nötig, Schreibzugriffe auf die Basisdaten während der Erstellung zu verbieten. Das heißt, dass Anwendungen möglicherweise für einige Zeit unverfügbar sind. Zum anderen bedeuten große Schnappschussdaten, dass der Vorgang, den Schnappschuss mit dem aktuellen Datenbankzustand zu vergleichen, ebenfalls mehrere Minuten dauern kann. Darunter leidet zwar nicht die Verfügbarkeit, jedoch verlangsamt es die Dauer einer inkrementellen NotaQL-Transformation.

Die Kosten einer inkrementellen Berechnung mittels Schnappschüssen teilen sich in zwei Teile: Zum einen die Kosten, einen Schnappschuss zu erzeugen und zum anderen die inkrementelle Transformation selbst. Letztere kann nur dann ausgeführt werden, wenn bei einer vorherigen Berechnung ein Schnappschuss erzeugt wurde. Dabei ist es nicht von Relevanz, auf welche Ausführungsart und auf welche inkrementelle oder nicht-inkrementelle Art die vorherige Transformation abgelaufen ist. Lediglich von Relevanz ist, dass die Datenmenge, welche ihr zugrunde lag, als Schnappschuss gespeichert wurde.

Ein Spezialfall der Schnappschuss-basierten Änderungserfassung lässt sich in Verbindung mit der Replikation einer Datenbank einsetzen. Dazu ist es nötig, dass der Replikationsmechanismus der Datenbank ein sogenanntes *Slave-Delay* unterstützt. Bei Systemen wie MongoDB [Mana] oder MySQL [Manb] kann beim Hinzufügen eines Slave-Rechners zu einem Replika-Set ein Slave-Delay von $N$ Sekunden konfiguriert werden. Auf diesen Slave-Rechner können lediglich Lesezugriffe erfolgen. Er hält stets die Daten in dem Zustand, welcher der Replikations-Master vor $N$ Sekunden hatte. Bei inkrementellen NotaQL-Berechnungen wird $N$ auf $t_1 - t_0$ gesetzt. Nun ist es wichtig, dass erneute Transformationen immer in genau diesem

Zeitabstand stattfinden. Der verzögernde Slave-Rechner liefert zum Zeitpunkt $t_1$ den Schnappschuss, also die Daten, die zum Zeitpunkt $t_0$ gültig waren. Dies ist zugleich der Zeitpunkt der vorherigen Berechnung, die das vorliegende Transformationsergebnis erzeugt hat. Mit den oben beschriebenen Mengensubtraktionen lässt sich die Menge der eingefügten, geänderten und gelöschten Datensätze bestimmen. Um bei diesem Verfahren ein stets korrektes Ergebnis zu erhalten, ist es vonnöten, dass sich weder der aktuelle Datenbestand noch der Schnappschuss ändern, während die Differenz zwischen diesen beiden Datenmengen bestimmt wird. Ersteres ist leicht realisierbar. Dazu muss der verzögernde Rechner lediglich temporär aus dem Replika-Set entfernt werden. Diese Trennung verhindert das Einbringen der nach $t_0$ stattgefundenen Änderungen. Beim Wiederverbinden werden diese nachgeholt. Änderungen auf dem aktuellen Datenbestand können jedoch nur durch einen Verbot von Schreibaktionen verhindert werden. Das Verfahren sorgt also für eine regelmäßige Nichtverfügbarkeit.

### Log-basierte Änderungserfassung

Datenbanksysteme legen aus verschiedenen Gründen Log-Dateien an, in welcher die in der Datenbank durchgeführten Operationen protokolliert werden. Sie beinhalten Informationen, welche Datensatze zu welchem Zeitpunkt wie geändert wurden. Diese Information ist für die Wiederausführung von erfolgreich abgeschlossenen Transaktionen nach einem Systemabsturz hilfreich, wenn die Auswirkungen vor dem Ausfall noch nicht auf ein persistentes Speichermedium gespeichert wurden. Zum anderen können die Log-Einträge zu Replikationszwecken genutzt werden. Ein Replika besorgt sich dazu einfach die noch zu erledigenden Änderungen von einem anderen Replika und führt sie anschließend lokal aus, um seine Datenbank auf den aktuellen Stand zu bringen. Sowohl bei Transaktionslogs als auch bei solchen, die für die Replikation angelegt werden, beinhaltet jeder Log-Eintrag eine *Redo-Anweisung*. Zur vollständigen und effizienten Änderungserfassung und damit zur inkrementellen Ausführung von NotaQL-Transformationen sind jedoch auch *Undo-Anweisungen* vonnöten. Diese beschreiben, wie eine ausgeführte Änderung rückgängig gemacht werden kann. Ohne die Undo-Anweisung geht der vorherige Zustand eines Wertes nach einer Änderung verloren. Das gleiche gilt für gelöschte Datensätze. Für Datenquellen, in denen jedoch nur Einfügungen passieren, ist die Redo-Information ausreichend. Treten jedoch auch Änderungen und Löschungen in den Basisdaten auf, müssen veraltete Datenwerte abrufbar sein. Dies geschieht entweder über Undo-Informationen, über einen vorliegenden Schnappschuss oder über möglicherweise sehr alte Redo-Informationen. Da keines der prominenten NoSQL-Datenbanksysteme Undo-Informationen zur Verfügung stellt, sondern ein Rückgängigmachen typischerweise mittels Schnappschüssen passiert [MC16], stellen wir im Folgenden einen Ansatz vor, der ohne Undo-Informationen auskommt. Wir präsentieren exemplarisch anhand von MongoDB einen Ansatz, mit dem sich aus Log-Daten Änderungen erfassen lassen, um daraus die Mengen $\Delta$ und $\nabla$ zu bestimmen.

**Log-Daten-Analyse in MongoDB**

Dient eine MongoDB-Kollektion als Eingabe für eine NotaQL-Transformation, kann statt die Kollektion direkt zu lesen, die sogenannte *Oplog* als Eingabe dienen [Dro16]. Die Oplog wird von MongoDB automatisch angelegt und gepflegt, wenn die Replikation aktiviert ist. Sie beinhaltet Log-Dokumente, die die folgende Gestalt haben:

```
{ "ts": Timestamp(1481210733, 2), "op": "i",
  "ns": "test.personen", "o": { "_id" : 1,
  "name": "Anita", "firma": "IBM", "gehalt": 50000 }}

{ "ts": Timestamp(1481210776, 1), "op": "u",
  "ns": "test.personen", "o2": { "_id": 1 },
  "o": { "$set": { "gehalt" : 50100 } } }

{ "ts": Timestamp(1481210823, 1), "op" : "d",
  "ns": "test.personen", "o": { "_id": 1 }}
```

Im gezeigten Beispiel wurde zuerst ein Personen-Dokument mit der ID 1 angelegt, danach das Gehalt dieser Person geändert und später wurde das Dokument gelöscht. `ts` ist der Zeitstempel, `op` die Art oder Operation (Insert, Update oder Delete) und `ns` der Namensraum, also der Name der Datenbank und der Kollektion. Die Felder `o` und `o2` beinhalten die Informationen, welches Dokument wie geändert oder gelöscht wird bzw. wie das einzufügende Dokument aussehen soll. Diese Informationen sind nicht zwangsläufig die exakten Parameter, die eine Anwendung einer Insert-, Update oder Delete-Operation mitgeliefert hat. Wird beim Einfügen aufgrund des Weglassens des `_id`-Feldes eine ID automatisch erzeugt, beinhaltet das `o`-Objekt ebendiesen erzeugten ID-Wert. Wird beim Ändern eine inkrementelle Operation, wie zum Beispiel `$inc:{gehalt:100}` ausgeführt, ersetzt MongoDB diese in der Log durch eine `$set`-Operation, die den neuen Wert enthält. Auch Listenoperationen wie `$push` und `$pull` tauchen als `$set`-Operation in der Oplog auf, die die vollständige neue Liste enthält. Die Kriterien, die beschreiben, welche Dokumente geändert bzw. gelöscht werden, werden durch MongoDB in `_id`-basierte Kriterien umgewandelt. All diese Umwandlungen verhindern bei der Replikation unerwünschte Effekte. Auch bei der inkrementellen Berechnung sind sie von Vorteil, da bei Änderungen lediglich `$set` und `$unset` unterstützt werden müssen.

Betrachten wir zunächst einen einfachen Fall, in dem nur Einfügungen und keine Änderungen und Löschungen passieren. Dann lässt sich die Menge $\Delta$ der Einfügungen in eine Kollektion `coll` seit einer vorherigen Transformation zum Zeitpunkt $t_0$ leicht bestimmen:

```
Delta = db.oplog.rs.find( { op: "i", ns: "db.coll",
                            ts: {$gt: t0, $lte: t1}})
```

Es ist zu beachten, dass die ermittelte Menge $\Delta$ noch unvollständig ist und nicht die aktuellen Zustände geänderter Datensätze beinhaltet. Das Vorgehen bei Änderungen und Löschungen ist nämlich deutlich aufwändiger, da

die gelöschten und überschriebenen Werte nicht in den Oplog-Dokumenten zu finden sind. Unveränderte Felder sind ebenfalls nicht Teil der Oplog-Dokumente. Während sich letztere aus dem aktuellen Datenbankzustand herleiten lassen, müssen historische Datenwerte aus alten Log-Einträgen und einem eventuell vorliegenden Schnappschuss rekonstruiert werden. Die MongoDB-Oplog ist in der Regel eine sogenannte *capped Collection*, also eine Kollektion mit einer fest konfigurierten Maximalgröße. Ist diese Größe erreicht, werden beim Einfügen alte Einträge in FIFO-Manier gelöscht, um Platz für die neuen zu schaffen. Damit inkrementelle Transformationen jedoch korrekt ausgeführt werden können, muss die Oplog alle seit dem absoluten Beginn ausgeführten Operationen beinhalten. Liegt ein Datenbank-Schnappschuss vor, ist es ausreichend, die Log-Einträge seit der Erstellung des letzten Snapshots zu besitzen.

Betrachten wir einen Oplog-Eintrag, welcher die Löschung eines Dokumentes mit einer bestimmten ID $x$ kennzeichnet. Für die in diesem Kapitel vorgestellte Strategie zu inkrementellen NotaQL-Transformationen ist es nötig, die Menge $\nabla$ zu bestimmen, welche das gelöschte Dokument enthalten soll. Um den Zustand des gelöschten Dokuments zum Zeitpunkt der vorherigen Transformation $t_0$ zu rekonstruieren, müssen die Log-Einträge, die bis zu diesem Zeitpunkt vorliegen, gefunden und kombiniert werden:

```
db.oplog.rs.find( { ns: "db.coll", ts: {$lte: t0},
           $or: [{op:"u", "o2._id":x}, {"o._id":x}]})
```

Die Fallunterscheidung in der zweiten Zeile ist nötig, da die ID des Dokuments, auf welches sich der Log-Eintrag bezieht, bei Updates im `o2`-Feld und andernfalls im `o`-Feld zu finden ist. Falls zum Zeitpunkt $t_S < t_0$ ein Schnappschuss erstellt wurde, kann das Zeitstempel-Kriterium mittels `$gt: tS` weiter eingeschränkt werden. Mit den gefundenen Log-Einträgen sowie mit dem eventuell vorliegenden Schnappschuss gilt es, den Zustand des Dokumentes zum Zeitpunkt $t_0$ zu rekonstruieren. Dieses Vorgehen muss für alle Dokumente wiederholt werden, für die es Löschungs- oder Änderungseinträge in der Oplog gibt. Das Resultat ist die benötigte Menge $\nabla$.

Es ist zu sehen, dass sich die Log-basierte Änderungserfassung für MongoDB nur lohnt, wenn es überwiegend Einfügeoperationen gibt. Änderungen und Löschungen werden zwar unterstützt, sie verlangsamen den Prozess jedoch. Ein regelmäßiges Anlegen eines Schnappschusses kann dem allerdings entgegenwirken.

### Trigger-basiert

Ein *Datenbank-Trigger* sorgt für die Ausführung einer *Aktion*, sobald ein gewisses *Ereignis* stattfindet. Die Ereignisse von Interesse sind im Falle von inkrementellen Berechnungen Einfügungen, Änderungen und Löschungen auf den Basisdaten. Die auszuführende Aktion kann vielfacher Natur sein. Wir betrachten die folgenden vier Möglichkeiten:

1. Die Trigger-Aktion ist eine Zeitstempel-Annotation in den Daten,
2. die Trigger-Aktion erzeugt einen Log-Eintrag,
3. die Trigger-Aktion pflegt die Mengen $\Delta$ und $\nabla$ direkt,

4. die Trigger-Aktion erzeugt einen Datenstrom.

Es fällt auf, dass die ersten beiden Möglichkeiten die Anwendbarkeit der in den vorherigen Abschnitten präsentierten Zeitstempel- und Log-basierten Änderungserfassung aktivieren. Dies ist für Systeme hilfreich, die keine eigene Versionierung und kein Logging anbieten, aber Trigger unterstützen. In diesem Abschnitt beschäftigen wir uns mit den zwei komplett anderen Arten der Änderungserfassung, nämlich den Möglichkeiten drei und vier.

Gegeben sei ein NotaQL-Skript, eine zum Zeitpunkt $t_0$ gültige Basisdatenmenge $B(t_0)$ und ein dazugehöriges Transformationsergebnis $E(t_0)$. Dieses Ergebnis wurde entweder mittels einer vollständigen Neuberechnung berechnet oder über eine inkrementelle Berechnungsart. Der einfachste Fall ist jedoch, wenn $B(t_0)$ und damit auch $E(t_0)$ leere Datenmengen sind. Tritt nun in der Basisdatenmenge eine Einfügung, eine Änderung oder eine Löschung auf, wird diese von einem Trigger wahrgenommen. Die einfachste Implementierung besteht nun daraus, dass der Trigger den eingefügten bzw. gelöschten Datensatz in die Menge $\Delta$ und $\nabla$ einfügt. Bei Änderungen muss der Trigger sowohl den alten als auch den neuen Zustand des geänderten Satzes ablegen. Die beiden Mengen $\Delta$ und $\nabla$ können nun entweder direkt oder in gewissen Zeitabständen verarbeitet und im Anschluss wieder geleert werden, um die inkrementelle Berechnung auszuführen. Im ersten Fall besteht die von einem Trigger erzeugte Menge $\Delta$ bei Einfügungen und Änderungen jeweils aus einem einzigen Datensatz, bei Löschungen ist die Menge leer. Analog dazu beinhaltet die Menge $\nabla$ bei Löschungen und Änderungen einen Datensatz und ist bei Einfügungen leer. Der Vorteil der direkten Ausführung ist, dass das Transformationsergebnis nicht auf möglicherweise veralteten Basisdaten basiert. Genau wie bei einer virtuellen Transformation ist es stets aktuell. Der zu Beginn dieses Kapitels präsentierte Ansatz, um die Mengen $\Delta$ und $\nabla$ auf das vorherige Ergebnis anzuwenden, basiert jedoch auf der Overwrite-Installation und erfordert daher ein vollständiges Einlesen des vorliegenden Ergebnisses. Ist dies sehr groß, ist eine direkte Trigger-basierte Ausführung ineffizient. Bei kleinen Ergebnismengen sowie bei einer verzögerten und stapelbasierten Ausführung kann diese Strategie jedoch effizient sein.

Ein dazu alternativer Ansatz basiert auf der Unterstützung von Datenströmen in NotaQL (siehe Kapitel 4.9). Bei diesem Ansatz ist die Aktion des Triggers eine direkte Weitergabe des eingefügten bzw. des gelöschten Datensatzes an die NotaQL-Plattform. Die NotaQL-Plattform behandelt diese eintreffenden Daten als *Datenstrom*. Während klassische Datenströme jedoch nur neue Daten liefern, sind die hier erzeugten Ströme auch auslöschender Natur.

Zur Implementierung dieses Ansatzes muss die Eingabe-Engine, welche um die Trigger-basierten Änderungserfassung erweitert werden soll, die Schnittstelle `StreamingEngine` implementieren (siehe Abschnitt 4.9). In der dort benötigten Methode `run` wird eine Kommunikationsschnittstelle eröffnet, über die der Datenbanktrigger der Engine die Datensätze liefert [Emd16]. Gelöschte Datensätze sowie alte Zustände geänderter Datensätze werden zuvor in der Trigger-Aktion mit einem Statusindikator annotiert, sodass sie von der NotaQL-Platform zur Kompensation vorangegangener Änderungen behandelt werden. Anders als bei den bisher in diesem Kapitel präsentierten Ansätzen liegt bei diesem Datenstrom-basierten

Ansatz das Ergebnis einer vorherigen Transformation nicht als zusätzliche Eingabe vor. Die Anwendung von Änderungen erfolgt also als Inkrement-Installation und nicht als Overwrite-Installation. Am besten eignet sich der Datenstrom-basierte Ansatz für Skripte ohne Aggregatfunktionen, da hier Transformationsergebnisse lediglich in Einfügungen oder Löschungen resultieren. Bei allen anderen Skripten muss vor der Aggregation ein Lesen der aktuellen Attributwerte in den betreffenden Zellen gelesen werden. Wie zu Beginn des Kapitels beschrieben, bietet sich dies an, wenn es wenige Änderungen seit der vorherigen Transformation gibt. Dies ist beim Datenstrom-basierten Ansatz stets der Fall, da immer nur eine Änderung nach der anderen bearbeitet wird. Dennoch können häufige Änderungen in den Basisdaten dazu führen, dass sie nicht schnell genug verarbeitet werden können und im schlimmsten Fall verloren gehen.

## 6.2 Plattform für inkrementelle Transformationen

In den vorherigen Abschnitten wurden eine Vielzahl von Ausführungsarten und Techniken zur Änderungserfassung präsentiert. Nicht jede Technik eignet sich für jedes NotaQL-Skript und jede Eingabe-Engine, zudem gibt es nicht die eine beste Ausführungsart. Weitere Faktoren sind die Beschaffenheit der zu verarbeitenden Daten, die Anzahl an seit einer eventuellen vormaligen Transformationen geänderten Datensätzen und das Vorhandensein und das Alter eines Schnappschusses. Erfolgen in den Basisdaten lediglich Einfügungen und keine Änderungen oder Löschungen, sind wiederum manche Ansätze besser geeignet als andere.

Aufgrund der Tatsache, dass es viele Ansätze gibt und keiner davon in jeder Situation der beste ist, wird in diesem Abschnitt eine Plattform vorgestellt, in welcher der Benutzer die Wahl hat, auf welche Art ein NotaQL-Skript ausgeführt werden soll. Ein später in diesem Absatz vorgestellter Advisor kann dem Benutzer zumindest teilweise diese Entscheidung abnehmen. Die präsentierte Plattform beinhaltet einige Ausführungsarten aus Kapitel 5 sowie die Techniken zur Änderungserfassung für inkrementelle Transformationen, die in diesem Kapitel vorgestellt wurden. Die Plattform lässt sich jedoch auch auf zusätzliche Ausführungsarten und inkrementelle Techniken erweitern.

### Gesamtarchitektur

Abbildung 6.5 zeigt die Gesamtarchitektur der NotaQL-Plattform. Deren Grundlage bilden die hellgrau dargestellten Ausführungsarten aus Kapitel 5. Die darüber abgebildete Schicht stellt die Dimension der inkrementellen Ausführung dar. Die in der Grafik dunkelgrau hinterlegten inkrementellen Ansätze sowie die virtuellen Transformationen sind jeweils Alternativen zu einer vollen Neuberechnung. Über allen diesen Ausführungskomponenten steht die Sprache NotaQL sowie die Advisor-Komponente, deren Aufgabe es ist, die geeignetste Ausführungsart vorzuschlagen. Neben der effizienten und korrekten Ausführung der in der Sprache NotaQL formulierten Transformationsskripte gilt es zudem, ebendiese Skripte

in einem *Transformationen-Katalog* abzulegen. Dies ist zum einen für Datenstrom-Transformationen und virtuelle Transformationen zwingend vonnöten, da diese kontinuierlich ablaufen. Zum anderen dient der Transformationen-Katalog zum Protokollieren von in der Vergangenheit ausgeführten Transformationen sowie deren Metadaten, wie den Zeitpunkt der letzten Ausführung, Informationen über vorhandene Schnappschüsse sowie Ausführungsstatistiken. Die oberste Schicht stellt die Schnittstelle zum Benutzer dar. Dieser kann Skripte entweder über eine grafische Oberfläche definieren, abschicken und verwalten, oder alternativ über eine Kommandozeilenschnittstelle.

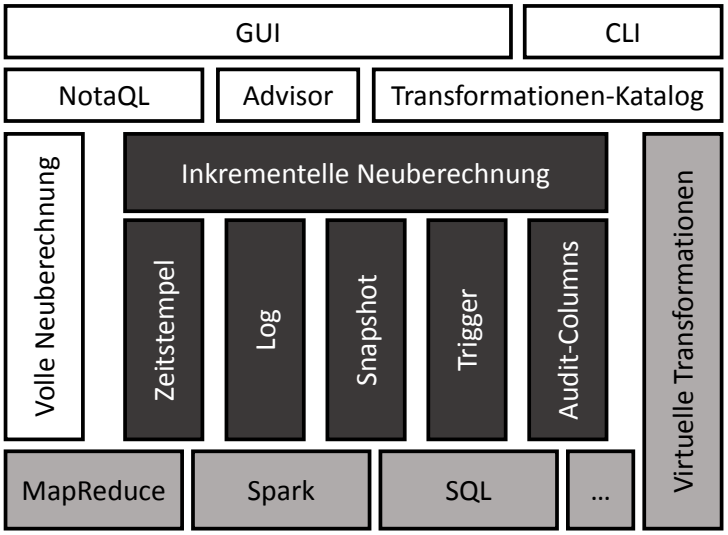



**Abbildung 6.5:** NotaQL-Plattform

Zur Interaktion über eine *Kommandozeilenschnittstelle* (Command Line Interface; *CLI*) benötigt diese im Wesentliche folgende Kommando-Unterstützungen:

- `execute "NOTAQL"`
- `define NAME "NOTAQL"`
- `drop NAME`

Der `execute`-Befehl führt eine gegebene NotaQL-Transformationsvorschrift aus. Da die Spezifikationen der Eingabe- und Ausgabe-Engines im NotaQL-Skript zu finden sind, sind keine weiteren Parameter Pflicht. Allgemeine Konfigurationen für die Plattform, die Ausführungsarten und einzelne Engines werden aus einer NotaQL-Konfigurationsdatei bezogen [Lot15]. Diese enthält beispielsweise die IP-Adressen sowie Ports zu Spark, MongoDB, u. s. w. Über zusätzliche optionale Parameter können die Ausführungsart sowie die Methode zur Änderungserfassung festgelegt werden. Entfallen diese Parameter, wird eine geeignete Weise automatisch durch die Advisor-Komponente gewählt (siehe Kapitel 6.2.3). Auch der Wunsch zum Anlegen eines Schnappschusses kann mittels eines Parameters geäußert werden. Beim Abschicken des `execute`-Befehls wird die NotaQL-Transformation wie angegeben ausgeführt. Nach erfolgreicher Beendigung liegt das Ergebnis in der gewünschten Ausgabedatenbank vor und der Transformationen-Katalog wird mit Metadaten und Ausführungsstatistiken gefüllt.

Im folgenden Beispiel sollen die Anzahl der Datensätze einer HBase-Tabelle in eine Redis-Datenbank geschrieben werden. Dazu soll Apache Spark sowie eine Zeitstempel-basierte Änderungserfassung zum Einsatz kommen. Der Zeitstempel der vorherigen Ausführung wird von der NotaQL-Plattform aus dem Transformationen-Katalog entnommen. Dazu ist es notwendig, dass die Transformationsvorschrift identisch mit der der vormaligen Ausführung ist. Andernfalls wäre eine korrekte Durchführung der inkrementellen Berechnung nicht garantiert, da diese nicht nur auf den geänderten Daten sondern auch auf dem vorherigen Transformationsergebnis basiert. Über den `--snapshot`-Parameter wird ein Schnappschuss des aktuellen Datenbankzustands angelegt. Dieser ermöglicht den Einsatz der Schnappschuss-basierten Änderungserfassung für beliebige in der Zukunft auf der gegeben Eingabetabelle ausgeführten Transformationen. Wie weiter oben in diesem Kapitel beschrieben, ist ein Schnappschuss auch für die Log-basierte Änderungserfassung hilfreich.

```
execute --mode=spark --inc=timestamp --snapshot
  "IN-ENGINE:hbase(table<-'personen'),
   OUT-ENGINE:redis(database<-'0'),
   OUT._k <- 'anzahl', OUT._v <- COUNT()"
```

Das `define`-Kommando dient zur Definition von virtuellen sowie von Datenstrom-Transformationen. Diese verarbeiten die Daten nämlich nicht Stapel-basiert sondern kontinuierlich. Da auch bei der Trigger-basierten Änderungserfassung Basisdatenänderungen der Transformationsplattform als Datenströme vorliegen, sind auch solche Transformationen kontinuierlicher Natur, bei denen ebendiese Art der Änderungserfassung zum Einsatz kommt. Der `define`-Befehl wird immer dann verwendet, wenn das Transformationsergebnis durchgehend von den Basisdaten abhängig ist. Während der Transformationen-Katalog beim `execute`-Befehl im Wesentlichen nur zur Anwendbarkeit verschiedener Änderungserfassungsmethoden und zur Verbesserung der Qualität des Advisors dient – für volle Neuberechnungen ist der Katalog theoretisch gar nicht zwingend notwendig –, ist er beim `define`-Kommando von hoher Relevanz. Eine mit diesem Befehl definierte Transformation führt in der Regel zunächst einmal keine tatsächliche Datentransformation aus, sondern speichert nur die Information ab, was wann transformiert werden soll. Bei virtuellen Transformationen erfolgt die Transformation erst beim Zugriff auf das Ergebnis. Dazu wird sich bei gegebener Zeit des NotaQL-Skriptes aus dem Transformationen-Katalog bedient. Bei Trigger- und Datenstrom-Transformationen erfolgen die eigentlichen Berechnungen, wenn sich Basisdaten ändern bzw. sobald Elemente im Eingabestrom auftauchen. Eine einmal definierte NotaQL-Transformation bleibt so lange aktiv, bis sie mit dem `drop`-Befehl wieder gelöscht wird. Zur einfachen Benutzbarkeit erfolgt das Anlegen und Löschen unter Angabe eines Transformationsnamens. Im folgenden Beispiel wird die weiter oben stehende Transformation Trigger-basiert definiert. Sie erhält den Namen `ANZ_PERSONEN`.

```
define ANZ_PERSONEN --mode=spark --inc=trigger
  "IN-ENGINE:hbase(table<-'personen'),
   OUT-ENGINE:redis(database<-'0'),
```

```
    OUT._k <- 'anzahl', OUT._v <- COUNT()"
```

Es fällt auf, dass das NotaQL-Skript exakt das gleiche wie im vorherigen Beispiel ist. Hier wird jedoch bei jeder Änderung in der Personen-Tabelle in HBase der Wert, welcher die Anzahl speichert, in der Redis-Datenbank aktualisiert. Nach dem Löschen mit dem folgenden Befehl steht das Ergebnis zwar weiterhin in der Ausgabedatenbank zur Verfügung, jedoch erfolgt keine automatische Aktualisierung mehr:

```
drop ANZ_PERSONEN
```

Als Alternative zur Kommandozeile können die Kommandos `execute`, `define` und `drop` auch unter Verwendung einer einer API direkt aus Anwendungen heraus abgesendet werden [Bha15]. Weitere Befehle wie `list` kommen zum Einsatz, um zuvor definierte Transformationen aufzulisten und Informationen sowie Statistiken über diese abzufragen.

### Graphische Oberfläche

Graphische Benutzeroberflächen haben im Gegensatz zur Kommandozeilenschnittstelle mehrere Vorteile. Zum einen bieten sie eine bessere Übersicht und eine leichtere Bedienung – NotaQL-Transformationen können klar strukturiert dargestellt werden und das Vornehmen von Änderungen sowie das Löschen von Transformationen ist mit wenigen Mausklicks möglich. Zum anderen sind Bedienhilfen vorteilhaft, um die Syntax von NotaQL-Skripen direkt bei der Eingabe zu überprüfen [Emd14] oder um Ausführungsarten, Engines und Methoden zur Änderungserfassung mittels Auswahlfelder anzubieten, damit der Benutzer direkt sieht, welche Optionen ihm zur Verfügung stehen [Tha16]. Auf diese Art können nicht unterstütze Kombinationen direkt ausgeblendet werden, um Fehler zu vermeiden. Besonders für unerfahrene Benutzer ist die Funktionalität sinnvoll, NotaQL-Transformationen vollständig mittels Mausklicks in graphischer statt in textueller Form erstellen zu können. Abbildung 6.6 zeigt einen Ansatz, um einfache NotaQL-Transformationen mittels Pfeilen zu definieren [Emd14]. Dazu klickt der Benutzer zuerst auf ein Feld auf der linken und danach auf eines auf der rechten Seite, um Pfeile zu erstellen. Das in der Abbildung gezeigte Beispiel führt eine horizontale Aggregation über alle in einer HBase-Zeile befindlichen Spaltenwerte aus und speichert die Resultate in einem Ausgabeattribut `Summe`. Über die graphische Darstellung können existierende Transformationsskripte leichter verstanden und angepasst werden und neue Transformationen schnell und ohne Einarbeitung in die Sprachsyntax kreiert werden.

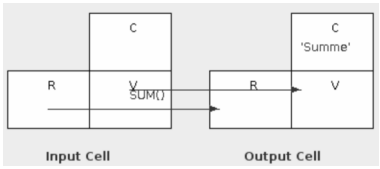



**Abbildung 6.6:**
Graphische NotaQL-Notation

Ein weiterer großer Vorteil einer graphischen Oberfläche wird bei der Definition von Transformationsketten deutlich. Liest eine Transformation die Ausgabe einer anderen, können beide als eine Kette graphisch dargestellt werden, sodass die Abhängigkeit sichtbar wird. Da mittels virtueller Engines (siehe Kapitel 4.10) die Ausgabe einer Transformation mehrfach von anderen wiederverwendet werden kann, und da es auch möglich ist, dass einer Transformation als Eingabe die Vereinigung oder der Verbund

aus mehreren Quellen dient, bildet die Menge der im Transformations-Katalog gespeicherten NotaQL-Skripte einen gerichteten Graphen. Auch wenn keine virtuellen Engines zum Einsatz kommen, können diejenigen Skripte, mit denen Daten gelesen werden, die eine andere Transformation erzeugt und in eine Datenbank persistent gespeichert hat, als eine zusammenhängende Komponente graphisch dargestellt werden. Die Menge der NotaQL-Skripte innerhalb einer zusammenhängenden Komponente heißt *Transformationsgruppe* [Emd16]. Diese Skripte sind stark voneinander abhängig. Beim Löschen eines Skriptes einer Gruppe werden in vielen Fällen die verbleibenden Skripte unbrauchbar, da ihnen entweder Eingabedaten fehlen oder da ihre Ausgabe unbehandelt bleibt. Da auf die Ein- und Ausgabe aber auch von extern zugegriffen werden kann, sollte das Löschen einzelner Skripte einer Gruppe jedoch nicht generell verboten sein.

In gewissen Fällen kann ein temporäres *Deaktivieren* kontinulierlicher NotaQL-Transformationen – also Datenstrom-, virtueller oder Trigger-basierter Transformationen – hilfreich sein. In diesem Falle werden auch alle vorangegangenen NotaQL-Skripte deaktiviert, es sei denn, von ihr lesen noch weitere Transformationen. Wird die Ausgabe der zu deaktivierenden Transformation nur von einer Folgetransformation verwendet, wird diese ebenfalls deaktiviert. Beim Wiederaktivieren werden dementsprechend auch die restlichen Gruppenskripte wieder aktiviert [Emd16].


**Abbildung 6.7:** Graphische Oberfläche

### Advisor

Die Aufgabe des Advisors ist es, die Ausführungsart und die Methode der Änderungserfassung automatisch zu wählen, mit dem Ziel, sich stets für die schnellste Art zu entscheiden. Wir beschränken uns in diesem Abschnitt auf Stapel-basierte NotaQL-Transformationen. Bei kontinuierlichen Transformationen – also virtuellen, Datenströmen und Trigger-basierten – ist das vorgestellte Vorgehen nicht anwendbar. Das liegt daran, dass kontinuierliche Transformationen keine messbare Gesamtlaufzeit haben, weil das Ergebnis stets aktuell gehalten wird. Damit ein Advisor auch kontinuierliche Transformationen unterstützen kann, müssen ihm Statistiken vorliegen, wie und wie oft auf Transformationsergebnisse von extern zugegriffen wird. Erfolgen nur sehr selten und nur gezielte Zugriffe auf das Ergebnis, kann eine virtuelle Transformation sinnvoll sein, eine Trigger-basierte jedoch eher nicht. Die dazu benötigten Zugriffsstatistiken liegen dem Advisor jedoch im Allgemeinen nicht vor, da viele Datenbanksysteme Lesezugriffe üblicherweise nicht protokollieren.

In einem ersten Schritt ermittelt der Advisor die Menge der *Ausführungskandidaten*. Dies sind die anwendbaren Konfigurationen. Sie werden ermittelt, indem für die im vorliegenden NotaQL-Skript spezifizierten Engines nicht unterstützten Ausführungsarten gestrichen werden. Außerdem werden Änderungserfassungsmethoden für inkompatible Ausführungsarten gestrichen. Die Zeitstempel-basierte Änderungserfassung ist nur möglich, falls im Transformationen-Katalog das vorliegende NotaQL-Skript präsent ist und der Zeitstempel einer vorherigen Ausführung vorliegt. Die Snapshot-basierte Änderungserfassung erfordert einen Schnappschuss auf der Eingabequelle, die Log-basierte Änderungserfassung eine ausreichend große Log-Datei. Die volle Neuberechnung ist in der Menge der Ausführungskandidaten stets vorhanden.

Im zweiten Schritt beginnt die eigentliche Arbeit des Advisors. Er wählt aus den Ausführungskandidaten diejenige Konfiguration aus, von der eine schnellstmögliche Transformationsdurchführung erwartet wird. Während der erste Schritt mittels Verträglichkeitsmatrizen und Fallunterscheidungen eindeutig erledigt werden kann, wird für den zweiten Schritt ein *Expertensystem* [Jac86] eingesetzt.

Ein Advisor kann entweder regelbasiert, modellbasiert oder fallbasiert arbeiten [Ber+03]. Der *Regel-basierte Advisor* entscheidet sich anhand hart codierter Regeln, welche Ausführungsmethode in welchen Fällen gewählt werden soll. Da NotaQL-Skripte komplex sind und da Engine-Spezifikationen und Datenänderungsraten in die Entscheidung mit einfließen, lassen sich solche Regeln nicht einfach formulieren. Des Weiteren würde die Einfühung neuer Engines und Ausführungsarten die Überarbeitung der Regeln erfordern. Ein *Modell-basierter Advisor* ist in der Lage, die Regeln selbstständig zu pflegen. Dazu fließen Statistiken über Erfolge und Fehlentscheidungen aus der Vergangenheit in die Regeldefinition mit ein. Über ein neuronales Netzwerk werden in einer Trainingsphase Schwellwerte für verschiedene Regeln automatisch trainiert. Dazu müssen jedoch ausreichend viele Testszenarien vorliegen. Diese sollten sich in den NotaQL-Skripten, den verwendeten Engines sowie der Beschaffenheit der Basisdaten unterscheiden und sie sollten eine Ground Truth bilden, indem sie die Antwort

auf die Frage beinhalten, welches die am besten geeignete Ausführungsart für das jeweilige Szenario ist. Letzteres ist im Allgemeinen nicht realisierbar, da diese Antwort oft weder automatisch noch manuell von einem Benutzer eindeutig und korrekt beantwortet werden kann. Ein *Fallbasierter Advisor* fällt seine Entscheidung nicht anhand von Regeln und Modellen, sondern anhand von Fällen aus der Vergangenheit. Da die Historie an Fällen im Transformationen-Katalog ohnehin vorliegt, eignet sich diese Arbeitsweise gut für den Advisor. Zudem ist das Fall-basierte Schließen nachvollziehbar und es passt sich automatisch an unbekannte Bedingungen an. Das bedeutet, dass beispielsweise beim Einführen einer neuen Engine am Anfang höchstwahrscheinlich einige Fehlentscheidungen getroffen werden, danach aber die *Case-Base* ausreichend gefüllt ist, um aus diesen Fehlern zu lernen und gute Ergebnisse zu liefern.

Führt der Benutzer ein NotaQL-Skript unter Verwendung des Advisors aus, sucht dieser im Transformationen-Katalog diejenigen NotaQL-Transformationen aus der Vergangenheit, welche ähnlich zu der gegebenen NotaQL-Vorschrift sind. Weitere Faktoren, die die Ähnlichkeit zweier Transformationen beeinflussen, sind die verwendeten Engines, die Datenbankgröße, der Anteil an Änderungen seit einer vormaligen Ausführung, die Selektivität eines Eingabefilters und mehr. Auch das Alter des Falls spielt eine Rolle. Ist eine Transformation eine lange Zeit nicht ausgeführt worden, spricht dies für hohe Änderungen in den Eingabedaten und weiteren geänderten Bedingungen. Die Liste der hier genannten Kriterien, die die Ähnlichkeit beeinflussen, ist erweiterbar. Alle Kriterien fließen in eine Distanzfunktion $d(notaql_1, notaql_2)$ ein. Die Distanzfunktion ist $1$, wenn die beiden zu vergleichenden Transformationen vollkommen verschieden sind. Ihr Resultat ist $0$, wenn sie sich vollkommen gleichen. In jedem Falle liegt ihr Wert im Intervall $[0, 1]$.

Welche Kriterien mit welcher Gewichtung in die Distanzfunktion einfließen, kann im Laufe der Zeit basierend auf der Qualität von Entscheidungen geändert werden. Auch das Training der Funktion mittels eines neuronalen Netzwerkes ist denkbar, um automatisch diejenigen Kriterien höher zu gewichten, die die höchste Relevanz auf die vom Advisor gefällte Entscheidung haben.

Aus der Menge zu einer gegebenen NotaQL-Transformation gefundenen ähnlichen Transformationen – das sind diejenigen, deren Distanz unter einem bestimmten Schwellwert liegt – wird diejenige Ausführungsart gewählt, die in der Vergangenheit die kürzeste Laufzeit hatte. Hier kann eine leichte Bevorzugung bei besonders ähnlichen Transformationen erfolgen, selbst wenn diese eine etwas längere Laufzeit besitzen. Auch ist es sinnvoll, dass sich mit einer gewissen Wahrscheinlichkeit für eine komplett andere Ausführungsart entschieden wird. Dies dient zum Ausprobieren und Lernen neuer Methoden sowie zur Bestätigung oder Falsifikation von Entscheidungen. Die Wahrscheinlichkeit für diese zufälligen Sprünge sollte für solche Transformationen höher sein, für die es noch wenige Erfahrungswerte gibt.

# 7 Anwendungen

Die Sprache NotaQL kann für Datentransformationen vielerlei Art verwendet werden und sie eignet sich besonders für ETL-Zwecke in Datenwarenhäusern. Das flexible Datenmodell und die Unterstützung systemübergreifender Transformationen machen es zudem möglich, materialisierte Analysen in Polyglott-Persistence-Umgebungen durchzuführen. In diesem Kapitel werden weitere NotaQL-Anwendungsszenarien dargestellt, die über einfache Datentransformationen hinausgehen. Klassische Anfragesprachen und Datenverarbeitungsframeworks sind für diese Zwecke oft ungeeignet.

## 7.1 Informationsintegration / Datenmigration

Unter Informationsintegration wird der Vorgang verstanden, eine Repräsentation von Informationen zu erzeugen, die zunächst auf mehreren separaten Systemen vorliegen [LN07]. Dabei gilt es, die in Kapitel 2.4 vorgestellten Formen der Heterogenität zu überwinden, da die zugrundeliegenden Systeme typischerweise unterschiedliche Zugriffsmethoden, Datenmodelle und Schemata zu bieten haben. In der Literatur wird zwischen der Integration auf Schema-Ebene und Daten-Ebene unterschieden. Für ersteres werden Algorithmen zum Schema-Matching [RB01; MBR01] und Schema-Mapping [HMH01; MHH00] eingesetzt, um Korrespondenzen zwischen verschiedenen vorlegenden Schemata automatisch zu finden und um daraus ein integriertes Schema zu erzeugen. Zweiteres, die Datenintegration, kann im im Anschluss daran erfolgen. In dieser Phase werden die eigentlichen Daten wahlweise virtuell oder materialisert in das integrierte Schema geladen, sodass über ebendieses auf die Gesamtheit der Daten zugegriffen werden kann.

Die von NotaQL gebotene Schema-Flexibilität bricht die strikte Trennung zwischen Schema- und Datenintegration. Das bedeutet, dass Daten und Schema als eine Einheit integriert werden können. Die bloße Erstellung eines Schemas ohne Dateninstanzen ist in den meisten NoSQL-Datenbanken ohnehin nicht notwendig, teilweise sogar gar nicht erst möglich. Das folgende NotaQL-Skript zeigt ein einfaches Beispiel, um Daten aus einem Quellsystem in ein Zielsystem zu übertragen:

```
IN-ENGINE: hbase(table <- 'quelle1'),
OUT-ENGINE: mongodb(db<-'test', collection<-'ziel'),
OUT._id <- IN._r,
OUT.$(IN._c) <- IN._v
```

Zunächst einmal kann man beim gezeigten Skript noch nicht von einer Integration reden, sondern eher von einer Datenmigration. In [STD16] wird der Begriff der Datenmigration als die Erzeugung und Übertragung eines Schnappschusses von einer Quell- in eine Zieldatenbank definiert. Im gezeigten Fall ist die Quelle eine HBase-Tabelle und das Ziel eine MongoDB-Kollektion. Die zwischen den Systemen vorliegende Datenmodellheterogenität wird mittels NotaQL überwunden. Erfolgt diese Transformation virtuell, können Anwendungen wie gehabt mit der HBase-Tabelle arbeiten, neue Anwendungen können aber auch lesend auf das Zielschema in MongoDB zugreifen. Bei einer materialisierten Transformation ist es möglich, Anwendungen dementsprechend zu modifizieren, um nur noch auf dem neuen MongoDB-Schema zu arbeiten [Ros13]. Alternativ können aber auch hier die Daten im Quellsystem existent bleiben und die MongoDB-Kollektion lediglich Analysezwecken dienen. In diesem Fall sollte eine regelmäßige inkrementelle Neuberechnung erfolgen, um die Kollektion aktuell zu halten.

Eine Datenintegration findet dann statt, wenn nun auf gleiche Art und Weise weitere Transformationen von anderen Quellsystemen in die gewählte Ziel-Kollektion erfolgen. Diese Kollektion enthält schließlich alle integrierten Datensätze. Ist ein Datensatz in mehreren Quellen vorhanden – dies wird anhand einer gleichen ID erkannt –, besteht der Zieldatensatz aus der Vereinigung aller gesetzten Attribute. Bei Attributkonflikten erfolgt ein Überschreiben, das heißt, das jeweilige Attribut hat als Wert denjenigen der zuletzt ausgeführten Transformation. Über die Wahl der Transformationsreihenfolge ist also eine Priorisierung der Datenquellen möglich.

Möchte man lediglich das Schema einer Eingabequelle analysieren und keine Daten migrieren, kann folgendes NotaQL-Skript zum Zwecke einer *Schema-Discovery* [Coh+01; WL97] eingesetzt werden:

```
IN-ENGINE: hbase(table <- 'quelle1'),
OUT-ENGINE: redis(db<-0),
OUT._k <- IN._c,
OUT._v <- COUNT()
```

Im gezeigten Fall werden Schlüssel-Wert-Paare in eine Redis-Datenbank geschrieben, die das Schema der HBase-Tabelle aufzeigen. Dabei trägt der Schlüssel den Namen einer Spalte und der Wert die Anzahl der Zeilen, in denen diese Spalte präsent ist (siehe Abbildung 7.1). Diese Informationen lassen sich dazu nutzen, um ein Schema zu verstehen, um korrekte Anfragen und Anwendungen zu erstellen, um manuell ein integriertes Schema aus mehreren Basisschemata zu entwickeln oder um

| Schlüssel | Wert |
|---|---|
| vorname | 511 |
| nachname | 499 |
| ort | 155 |
| email | 1511 |
| name | 982 |

**Tabelle 7.1:** Schema-Discovery

Korrespondenzen zwischen verschiedenen Quellen automatisch zu ermitteln. Des Weiteren lässt sich aus diesen Schema-Informationen ein Schema-Validator konstruieren. Dieser sorgt für die Einschränkung der Schema-Flexibilität und garantiert, dass alle ab nun eingefügten Datensätze keine Attribute besitzen, die bisher nie verwendet wurden, oder er garantiert, dass gewisse Attribute zwangsweise in den Datensätzen vorhanden sein müssen. Letzteres ist in MongoDB wie folgt über ein boolesches Prädikat möglich:

```
db.runCommand( { collMod: "personen",
   validator: { email: { $exists: true }  } } )
```

Eine weitere Anwendung der oben dargestellten Datenmigration mittels NotaQL ist das Anlegen und Zurückspielen von *Backups*. Zwar bieten die meisten Datenbanksysteme eingebaute Methoden, um einen Datenbank-Dump zu erstellen, jedoch ermöglicht NotaQL darüber hinaus die inkrementelle Wartbarkeit. Des Weiteren erlaubt NotaQL auch Backups für Systeme, die keine eingebaute Backup-Funktion haben, beispielsweise Cloud-Dienste wie Google Apps, Salesforce, Office 365 oder Box. Solche Dienste bieten zwar eine API zum lesenden und schreibenden Zugriff, komplette Sicherungen und Rücksicherungen sind jedoch nur manuell möglich. Obwohl sich der Cloud-Anbieter um die Erstellung von Backups kümmert, kann ein Datenverlust nie vollständig vermieden werden, vor allem nicht bei einer Fehlbedienung. In [Ron15] werden sechs Kriterien genannt, die eine Cloud-to-Cloud-Backup-Lösung zu bieten haben soll:

1. Backups sollen automatisch mehrmals täglich erfolgen,
2. Backups sollen Off-Site – also außerhalb der Cloud – erfolgen,
3. Backups sollen jederzeit wieder in die Cloud zurückgespielt werden können; auch historische Zustände,
4. Multi-App-Unterstützung; nicht nur für eine Cloud-Anwendung,
5. Reporting-Funktionalitäten zur Nachvollziehbarkeit,
6. Suchfunktionen innerhalb der Backups.

Alle diese Kriterien werden von NotaQL erfüllt. Durch die Implementierung von NotaQL-Engines für verschiedene Cloud-Services lässt sich beispielsweise ein Backup von einem Google-Kalender in eine lokale MongoDB-Datenbank ablegen. Das Reporting erfolgt in Log-Dateien durch die NotaQL-Plattform, zur Suche können die MongoDB-Suchfunktionen genutzt werden. Da NotaQL System-übergreifende Transformationen unterstützt, kann die Sprache zudem auch dazu eingesetzt werden, um Daten von einem Cloud-Service zum anderen zu migrieren und um von konventionellen Systemen zu Cloud-System zu wechseln und umgekehrt.

## 7.2 Schema-Evolution

Sind die Quelle und das Ziel einer NotaQL-Transformation gleich, lassen sich Daten an Ort und Stelle modifizieren. Dies ermöglicht Stapel-basierte Änderungen, die wegen mangelnder Funktionalität der NoSQL-Datenbanken oft nicht mit mitgelieferten Zugriffsmethoden durchführbar sind. Lediglich in einigen wenigen Systemen wie MongoDB oder solchen, die eine SQL-Schnittstelle anbieten, ist dies möglich [Phy13]. Meist scheitern diese Methoden jedoch an der Unterstützung flexibler Schemata. Mit NotaQL können auf eine einfache Art Attributwerte geändert werden, auch ohne die Attributsnamen zu kennen, es können Metadaten in Daten gewandelt werden und umgekehrt.

Unter einer *Schema-Evolution* sind laufende Änderungen am Datenbankschema zu verstehen, die aufgrund von sich ändernden Anwendungsanforderungen meist in unregelmäßigen Abständen durchgeführt werden müssen. Aufgrund der Tatsache, dass NoSQL-Datenbanken meist Schema-frei sind und dass Daten in semi-strukturierter Form gespeichert werden, ist eine Schema-Evolution deutlich komplexer durchzuführen als es in relationalen Datenbanken der Fall ist. Denn letztere unterstützten die von SQL bereitgestellten DDL-Befehle wie `ALTER TABLE` zum Bearbeiten von Tabellen, Spalten und Datentypen. Während sich diese Befehle im Wesentlichen lediglich auf den Metadaten-Katalog auswirken, hat eine Schema-Evolution in NoSQL-Datenbanken meist zur Folge, dass alle Datensätze Stapel-basiert geändert werden müssen. Der Grund dafür ist, dass das Schema ein Teil der Daten ist.

Bei der Entwicklung moderner Web-Applikationen kommen häufig *agile Softwareentwicklungsmethoden* zum Einsatz. Dabei ändert sich das Datenbankschema einer Anwendung typischerweise mindestens einmal im Monat [Amb12; Cur+08], oft sogar wöchentlich oder täglich [McH99]. Bei jeder solchen Änderung ist es vonnöten, *Schema-Migrationen* durchzuführen. Darunter versteht man den Prozess, Datensätze aus alten Anwendungsversionen in ein neues Schema zu überführen. Dabei wird zwischen einer *Eager-* und einer *Lazy-Migration* unterschieden [SCA15]. Im ersteren Fall iteriert ein Programm über alle Datensätze des kompletten Datenbestands und migriert diese [SKS13]. Dieser Prozess ist bei großen Datenmengen oft langdauernd und erfordert zur korrekten Durchführung ein Stilllegen von Anwendungen, was eine Nichtverfügbarkeit zur Folge hat. Die *Lazy-Migration* verläuft dagegen im laufenden Betrieb. Dazu müssen jedoch die Anwendungen insofern angepasst werden, dass sie in der Lage sind, sowohl Datensätze des neuen als auch des alten Schemas zu lesen. Letztere werden schließlich in das neue Schema überführt. Solche Änderungen sind vergleichsweise leicht durchzuführen, wenn Objekt-NoSQL-Mapper eingesetzt werden. Nichtsdestoweniger muss mit Sorgfalt darauf geachtet werden, keine Migration zu übersehen, da dies zu Fehlverhalten und Datenverlust führen kann. Werkzeuge wie ControVol [SCA15] warnen den Anwender davor und sorgen für eine korrekte Migration. Es ist jedoch zu beachten, dass Datensätze, auf die nie oder nur lesend zugegriffen wird, in der Datenbank im alten Schema verbleiben. Eine tatsächliche Datenmigration erfolgt erst beim Schreiben der Daten. Möchten andere Anwendungen ebenfalls

auf die Daten zugreifen, müssen diese ebenfalls die Lazy-Migration durchführen oder man überführt schließlich für solche Fälle die noch unmigrierten Datensätze mittels einer Eager-Migration in das neue Schema.

Eine Eager-Migration kann mittels NotaQL wie folgt durchgeführt werden [Hau+17]:

```
IN-ENGINE: mongodb(db<-'test', collection<-'person'),
OUT-ENGINE: mongodb(db<-'test', collection<-'person'),
IN-FILTER: IN.schemaVersion < 2,
OUT._id <- IN._id,
OUT.schemaVersion <- 2,
OUT.strasse <- DROP(),
OUT.street <- IN.strasse
```

An der Engine-Spezifikation ist zu sehen, dass Datensätze an Ort und Stelle modifiziert werden. Die Änderung erfolgt nur auf Datensätze, mit einer Schema-Version, die niedriger als die aktuelle ist. Im aktuellen Schema wurde nämlich das Attribut `strasse` in `street` umbenannt. Die Transformation sorgt dafür, das Attribut `strasse` aus allen Datensätzen zu entfernen, den Wert dieses Attributs in ein neues namens `street` zu speichern und die Schema-Versionsnummer zu erhöhen. Letzteres macht die Transformation idempotent, sodass eine wiederholte Ausführung auf bereits migrierte Datensätze keine Auswirkung hat.

Da die NotaQL-Skripte bei umfangreichen Schema-Evolutionen schnell komplex werden können, bietet das Programm *ControVol Flex* [Hau+17] die Möglichkeit, NotaQL-Transformationen zur Ausführung von Eager-Migrationen automatisch aus Anwendungscode erzeugen zu lassen. Dabei werden die Änderungen an Klassenattributen überwacht und eine Warnung ausgegeben, wenn es nach einer solchen Änderung Datensätze des alten Schemas in der Datenbank geben kann. Um das Problem zu lösen, zeigt das Programm ein generiertes NotaQL-Skript zur Eager-Migration, welches nach Belieben erweitert werden kann, um komplexere Transformationen durchzuführen. Beispielsweise kann nach der Umbenennung des Attributs `name` in `vorname` mittels einer Split-Funktion (siehe unten im folgenden Abschnitt) nur der Teil des Namens vor einem Leerzeichen übernommen werden, während der Teil danach zu dem Wert eines neuen Attributs `nachname` wird.

ControVol Flex unterstützt neben der Eager- auch die Lazy-Migration mittels Objekt-NoSQL-Mapper-Annotationen sowie eine *hybride Migration*. Dabei handelt es sich um eine Kombination der beiden Formen. Dies ermöglicht Eager-Migrationen ohne Ausfallzeit, da während der laufenden Migration in Anwendungen parallel lazy migriert wird. Dies ist auch möglich, wenn die Lazy-Migration bereits einige Zeit aktiv ist und nicht-migrierte Datensätze schließlich eager migriert werden sollen. Die danach überflüssig werdenden Programmcode-Annotationen werden von ControVol Flex automatisch entfernt [Hau+17].
Abbildung 7.1 zeigt den Dialog zur Lösung von Schemaevolutionsproblemen in ControVol Flex. Nach einer Umbenennung eines Klassenattributs `highscore` in `score` warnt die Entwicklungsumgebung, dass es in der Datenbank noch Datensätze des alten Schemas geben kann. Zur Lazy-Migration wird im Dialog angeboten, dem Programmcode Annotationen anzufügen, zur Eager-Migration kann ein NotaQL-Skript generiert werden.

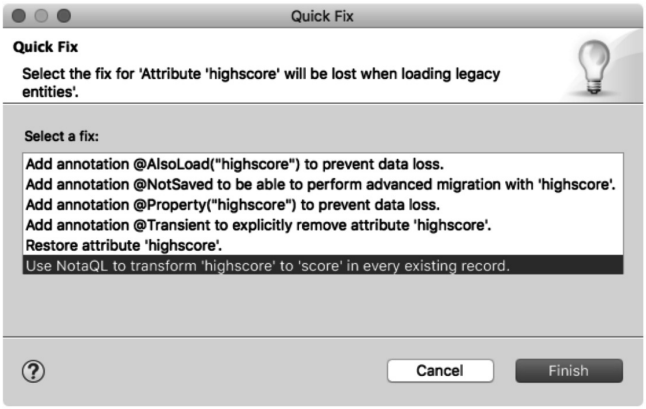



**Abbildung 7.1:** ControVol Flex

Letzteres ist auch dann noch nachträglich möglich, wenn bereits Annotationen hinzugefügt wurden. Dieser Fall entspricht der hybriden Migration.

## 7.3 Multimedia- und Text-Analyse

Das relationale Modell und die Anfragesprache SQL sind darauf optimiert, dass Datenwerte in atomarer Form in Tabellenspalten abgelegt werden. Dies ermöglicht nicht nur eine effiziente Speicherung, sondern auch eine gute Durchsuchbarkeit, Indexierbarkeit und schnelle Berechnungen. Diese Vorteile zeigen sich vor allem bei nummerischen Datenwerten. Der SQL/MM-Standard [ME01] erweitert relationale Datenbanken um die Speicherung und um mächtige Anfragemethoden für Multi-Media-Daten wie Text-, Bild-, Audio- und Video-Dokumente. Auch MapReduce, Spark und andere Datenverarbeitungsframeworks werden häufig zur Analyse von Text- und Multimediadaten eingesetzt [LD10]. Sowohl in SQL/MM als auch in den genannten Frameworks erfolgt die Analyse meist mittels benutzerdefinierten oder vorgefertigten Funktionen. Im Falle von SQL wird also nicht die Sprache selbst erweitert, sondern auf existierende Konzepte wie Routinen zurückgegriffen.

Auch in NotaQL kann die Multimedia-Daten-Analyse mittels benutzerdefinierter Funktionen (siehe Kapitel 4.7) erfolgen. Durch die Auslagerung der Analyselogik können existierende Programmbibliotheken wiederverwendet werden und in NotaQL-Skripten zur Datentransformation und zur Filterung eingesetzt werden. Das folgende Beispiel verwendet eine benutzerdefinierte Funktion zur Suche nach orangefarbigen Bildern:

```
IN-FILTER: img_farbe(IN.rawdata) = 'orange'
```

Auf die gleiche Art und Weise können Metadaten aus Bildern, Videos und Audiodaten extrahiert werden oder neuronale Netzwerke zum Einsatz kommen, die ebendiese Daten anhand der Rohdaten herausfinden und klassifizieren.

Textdokumente unterscheiden sich von anderen Multimediadaten dadurch, dass sie nicht zwangsläufig in ihrer Rohdaten-Form als Byte-Array analysiert werden müssen, sondern dass sie in Form einer Zeichenkette des Datentyps String vorliegen. Da dies sowohl in NotaQL als auch in den für

benutzerdefinierten Funktionen zum Einsatz kommenden Programmiersprachen ein eingebauter Datentyp ist, sind mächtige Textanalysen leicht zu realisieren. Die auf dem Java-Typ String vorhandenen Methoden wie `contains` oder `substring` können durch eine Abbildung als NotaQL-Funktion auch in NotaQL-Transformation zur Volltextsuche und zur Zerlegung von Textdaten eingesetzt werden. Komplexere Textanalysen basieren auf Listen oder Mengen von Textelementen. Um solche Textelemente zu generieren, liefert die NotaQL-Funktion `split` aus einem gegebenen Text und einem Texttrenner ein listenwertiges NotaQL-Objekt. Dies kann sowohl als Eingabe für die bekannten Aggregatfunktionen wie `COUNT` dienen – in dem Fall werden die Anzahl der Textelemente gezählt – als auch zur Weitertransformation mittels benutzerdefinierter Funktionen. Ein Beispiel für letzteres ist die Entfernung von Stoppwörtern aus einer Liste von Wörtern. Zusätzlich können die einzelnen Textelemente als ID und Attributsnamen von Ausgabedatensätzen verwendet werden. Damit lassen sich beispielsweise die Wörter in einer Dokumentensammlung zählen, Log-Analysen durchführen oder invertierte Termindexe erzeugen.

**Wörter zählen** Der Text wird in einzelne Wörter zerlegt, welche als ID-Werte der Ausgabedokumente verwendet werden. In einem Anzahl-Attribut wird gezählt, wie oft jedes Wort vorkommt.

```
OUT._id <- split(IN.text, ' '),
OUT.anzahl <- COUNT()
```

```
life is life                    { _id: "is", anzahl: 1 }
na na na na na          →       { _id: "life", anzahl: 2 }
                                { _id: "na", anzahl: 5 }
```

**Log-Analysen** Einträge in Log-Dateien sind typischerweise mit einem Begrenzungssymbol getrennte Werte. Eine Datei, in der Webseiten-Zugriffe geloggt werden, kann mittels folgendem Skript analysiert werden, um die Browserverteilung je Ressource zu ermitteln.

```
OUT._id <- split(IN.text, ' ')[1],
OUT.$(split(IN.text, ' ')[2]) <- COUNT()
```

```
9:45 main.htm Firefox       →   { _id: "main.htm",
9:47 main.htm Chrome                Firefox: 1, Chrome: 1 }
```

Alternativ lässt sich zur Log-Analyse auch eine CSV-Engine verwenden, in welcher die Text-Zerlegung von der Engine selbst übernommen wird, sodass direkt mittels `IN.1` bzw. `IN.2` auf die Werte zugegriffen werden kann.

**Invertierte Termindexe** Um effiziente Volltextsuchen zu ermöglichen, kommen oft invertierte Termindexe zum Einsatz. Diese verwalten zu jedem Schlagwort (Term) die IDs derjenigen Dokumente, in denen das Schlagwort auftaucht. Zusätzlich können die Anzahl der Vorkommen sowie die Position im Dokument gespeichert werden. Das folgende NotaQL-Skript erzeugt für jedes Wort ein Ausgabedokument. Die Attributsnamen beinhalten die Dokument-IDs, die Werte die Häufigkeit im Dokument.

```
OUT._id <- split(IN.text, ' '),
OUT.$(IN._id) <- COUNT()
```

```
{ _id:1, txt:"hey hey" }  →  { _id:"hey","1":2,"2":1 }
{ _id:2, txt:"hey ya" }      { _id:"ya","2":1 }
```

Die vorgestellten Transformationen lassen sich inkrementell warten und sie sind auch auf Datenströmen anwendbar. Komplexere Textanalysen lassen sich mittels Transformationsketten realisieren.

## 7.4 Sampling und Visualisierung

In [Ma+16] wird eine Einführung in die Problematik der *Big Graph Search* sowie Verweise auf viele Anwendungen gegeben. Die Suche in Graphen wie sozialen Netzwerken sei historisch gesehen die neuartigste Form zu Suchen – nach Dateisuchen, Datenbanken und Web-Suchmaschinen. Ein geeignetes Such- und Analyseprogramm solle die *FAE-Regeln* erfüllen: Friendlyness, Accuracy, Efficiency. Neben der einfachen Bedienung soll also eine hohe Trefferquote und eine hohe Geschwindigkeit angestrebt werden. Um vor allem letzteres immer weiter steigern zu können, können Techniken zur Performanzgewinnung auf Kosten der Trefferquote und damit der Genauigkeit eingesetzt werden. Möglichkeiten dazu sind, eine Anfrage in eine einfacher ausführbare Anfrage umzuwandeln, die ein ungefähres Ergebnis liefert, oder die Eingabedaten werden vereinfacht – niedrige Zahlenwerte könnte man durch Nullen ersetzen –, oder aber es kommen *Sampling-Techniken* zum Einsatz.

Unter dem Begriff Sampling werden Verfahren verstanden, die auf Stichproben arbeiten. Für NotaQL-Transformationen bedeutet das, dass die Eingabe-Engine nur einen gewissen Anteil der Eingabedaten einliest und die Analyse lediglich auf diesem Anteil ausgeführt wird. Durch das Ignorieren eines Großteils der Grunddatenmenge kann durch Sampling-Verfahren eine Transformation deutlich beschleunigt werden. Wie oben beschrieben, leidet darunter allerdings die Genauigkeit des Ergebnisses. Da jedoch für viele Analyseprobleme eine gewisse Ungenauigkeit toleriert wird, gilt es, das richtige Maß zwischen Genauigkeit und Geschwindigkeit zu finden. Im Allgemeinen gilt, dass sich die Genauigkeit erhöht, je größer der Anteil der zu lesenden Grunddatenmenge ist. Eine hundertprozentige Genauigkeit kann meist nur dadurch erreicht werden, indem auch einhundert Prozent der Daten eingelesen werden. In anderen Fällen stellt sich die Ermittlung eines Genauigkeitsmaßes als schwierig dar. Es lässt sich entweder exakt bestimmen, indem Sampling-basierende Analyseergebnisse mit den korrekten Ergebnissen verglichen werden – für diesen Vergleich muss

jedoch eine Gesamtberechnung erfolgen – oder die Genauigkeit wird geschätzt.

Wir teilen die Klasse der NotaQL-Transformationen zunächst in drei Unterklassen ein, welche sich durch deren Anwendbarkeit auf Sampling-Verfahren unterscheiden:

- *Typ A*: NotaQL-Transformationen, die unverändert auf Stichproben ausgeführt werden können,
- *Typ B*: NotaQL-Transformationen, bei denen eine Hochrechnung der Ergebnisse erfolgen muss,
- *Typ C*: NotaQL-Transformationen, die für Sampling-Verfahren ungeeignet sind.

Zum Typ A gehören solche Skripte, in denen die Aggregatfunktionen `AVG`, `MIN` oder `MAX` vorkommen. Die Ergebnisse dieser Funktionen entsprechen dem tatsächlichen Ergebnis einer auf der Gesamtdatenmenge basierenden Transformation, vorausgesetzt, die gesamten Eingabedaten besitzen exakt die gleiche Beschaffenheit wie die Stichprobe.

Ergebnisse von Transformationsskripten, in denen `COUNT`-Funktion zum Einsatz kommen, liefern in der Regel um den Faktor $n/N$ zu kleine Werte, da die Zählungen nur auf einem Teil der Daten basiert. Hierbei ist mit $n$ die Stichprobengröße und mit $N$ die Größe der Gesamtheit gemeint. Um die Werte abzuschätzen, die bei einer Komplettberechnung herauskämen, ist eine Korrektur der Ergebnisse vonnöten. Diese erfolgt durch eine Multiplikation mit dem Faktor $N/n$. In vielen Anwendungen zählen auch Skripte mit der `SUM`-Funktion zum Typ B, vorausgesetzt diese Funktion erfüllt bezüglich der zum summierenden Werte die Monotonieeigenschaft. Genauer gesagt muss die Summe stets (positiv oder negativ) wachsen, je mehr Datenwerte als deren Eingabe dienen. In diesem Fall kann auch hier eine Abschätzung mittels einer Hochrechnung um den Faktor $N/n$ erfolgen. Es ist möglich, dass NotaQL-Skripte sowohl zum Typ A als auch zum Typ B gehören, wenn mehrere Aggregatfunktionen zum Einsatz kommen. In diesem Falle erfolgt bei einigen Ausgabeattributen die Hochrechnung, bei anderen entfällt dieser Schritt.

Skripte ohne Aggregatfunktion oder solche mit Funktionen wie `LIST`, die alle zu aggregierenden Datenwerte in ihrer Ursprungsform belassen und sie nicht auf einen atomaren Wert reduzieren, gehören zum Typ C und können nicht in der Verbindung mit Sampling-Verfahren verwendet werden. Das Analyseergebnis ist unvollständig und es gibt keine Möglichkeit, Abschätzungen über die nicht eingelesenen Datenwerte in das Ergebnis mit einfließen zu lassen.

### Stichprobennahme

Da die wenigsten Datenbankmanagementsysteme eine eingebaute Funktionalität bieten, eine repräsentative Stichprobe der Daten anzufordern, muss dies von der NotaQL-Plattform übernommen werden. Anhand der Ausführungsmethode MapReduce werden in diesem Unterabschnitt verschiedene Ansätze dazu präsentiert. Diese lassen sich auch auf andere Ausführungsmethoden übertragen.

Die *Early Accurate Result Libary* (*EARL*) [LZZ12] ist eine Hadoop-Erweiterung, um MapReduce-Jobs auf einer Stichprobe der Eingabedaten durchzuführen und die Ergebnisse hochzurechnen. Der Sampling-Anteil $n/N$ wird dabei als Prozentsatz $p\%$ angegeben. Dieser Wert bedeutet zum einen, dass $p\%$ der Eingabe eingelesen werden sollen, zum anderen kann dieser Wert als die Wahrscheinlichkeit gedeutet werden, dass ein Eingabedatensatz in die Stichprobe aufgenommen wird. Bei einem *Pre-Map*-Samplingansatz wird vor der Anwendung der Map-Funktion entschieden, ob ein Datensatz diese Funktion passieren soll oder nicht. Bei Map-Funktionen, in denen eine variable Anzahl von Zwischenergebnissen produziert wird, kann ein *Post-Map*-Ansatz die Genauigkeit dadurch erhöhen, dass zunächst alle Datensätze die Map-Funktion passieren, jedoch die Map-Ausgaben mit einer Wahrscheinlichkeit von $(100-p)\%$ wieder verworfen werden. Zudem gibt es bei MapReduce und anderen Verfahren die Alternative, ob einzelne Datensätze oder ganze Splits bzw. Chunks von Datensätzen übersprungen werden sollen [SSD15]. Letztere Alternative wird Cluster-Sampling [Sud76] genannt und erfordert den geringsten Eingabe-Ausgabe-Aufwand. Damit jedoch Sampling für jede mögliche Eingabe-Engine in NotaQL unterstützt wird, eignet sich die erste Alternative, in der jeder einzelne Datensatz, welcher von der Eingabe-Engine geliefert wird, entweder passieren darf oder übersprungen wird.

### Hochrechnung

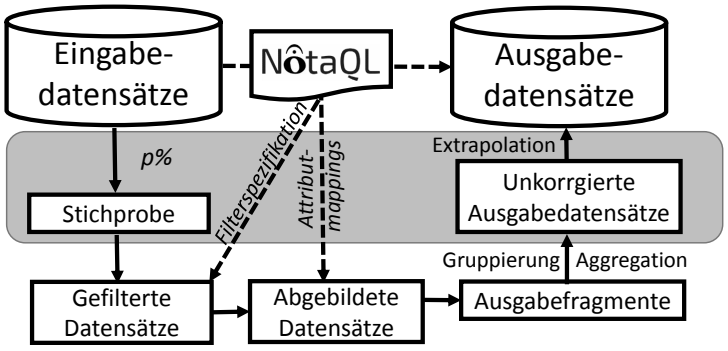



**Abbildung 7.2:** Sampling in NotaQL: Post-Reduce-Extrapolation

Abbildung 7.2 zeigt, wie die NotaQL-Ausführung um Sampling-Verfahren erweitert werden kann. Bei Skripten vom oben beschriebenen Typ A, also solche, in denen keine Hochrechnung vonnöten ist, ist lediglich der linke Teil der Grafik von Relevanz. Dieser Teil beschreibt, dass eine Stichprobe der Größe $p\%$ aus den von der Eingabe-Engine bereitgestellten Eingabedaten gezogen wird, welche die Basis der Transformation bildet. Bei Skripten von Typ B ist zudem eine Hochrechnung notwendig. Wie im rechten

Teil der Grafik zu sehen, erfolgt diese nach der Gruppenbildung und nach der Berechnung der Aggregate für jede einzelne Gruppe. Die ermittelten Zahlenwerte werden um den Faktor $100/p$ nach oben korrigiert. Bei einer Ausführung mittels MapReduce erfolgt diese Hochrechnung innerhalb der Reduce-Funktion bzw. nach der eigentlichen Reduce-Ausführung. Daher sprechen wir hier von einer *Post-Reduce-Extrapolation* [SSD15].
Die sogenannte *Pre-Reduce-Extrapolation* verfolgt einen anderen Ansatz, welcher viel generischer anzuwenden ist und sich daher noch besser für die Ausführung von NotaQL-Skripten eignet. Während beim Post-Reduce-Ansatz die Hochrechnung für jedes einzelne Attribut und in Abhängigkeit von der verwendeten Aggregatfunktion erfolgen muss, kann beim Pre-Reduce-Ansatz die Reduce-Funktion in immer gleicher Manier ausgeführt werden. Die Funktionsweise ist in Abbildung 7.3 dargestellt.

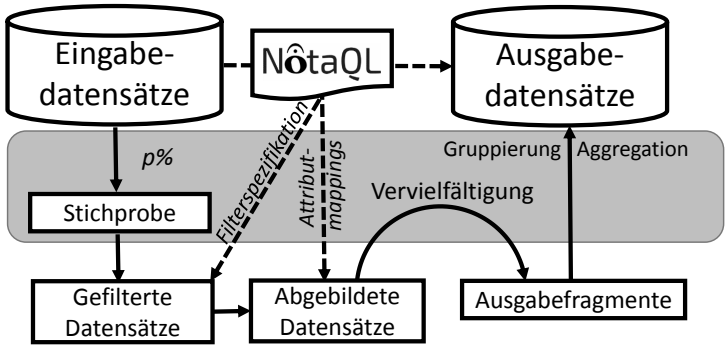



**Abbildung 7.3:** Sampling in NotaQL: Pre-Reduce-Extrapolation

Nachdem die noch nicht gruppierten und aggregierten abgebildeten Datensätze vorliegen – bei der MapReduce-Ausführungsart ist dies nach der Map-Phase der Fall –, werden diese Zwischenergebnisse um den Faktor $100/p$ vervielfältigt. Die Reduce-Funktion sieht also jedes vorliegende Objekt mehrfach. Um den Speicherbedarf und die Netzwerklast dabei nicht künstlich zu erhöhen, kann die Vervielfältigung rein virtuell erfolgen. Ein Zähler $k$ an jedem Wert gibt an, wie oft die Reduce-Funktion diesen Wert als Eingabe erhalten soll. Die zur Aggregation von Zwischenwerten durchlaufende Schleife wird dazu um zusätzliche Durchläufe erweitert. Dies kann durch eine Erweiterung der `Iterator`-Schnittstelle implementiert werden:

```
Interface MultiIterator<T extends Mutli>
          extends Iterator<T extends Mutli> {
 Iterator<T> it; // eigentlicher Iterator
 int counter = 0;
 T current;
 public T next() {
  if(counter == 0) {
   this.current = it.next();
   this.counter = current.k;
  }
  this.counter--;
  return this.current;
 }
```

Im gezeigten Code kommt das Delegate-Pattern zum Einsatz, damit ein Multi-Iterator zum einen nach außen als Iterator fungieren kann, zum anderen bezieht er seine Elemente aus einem existierenden Iterator. In diesem wird aus jedem Element der Zählerwert $k$ entnommen und das Element dementsprechend oft ausgegeben. Da es jedoch keine halben Durchläufe geben kann, sollte $k$ ganzzahlig sein, was für die Prozentsätze $p$, die ein Teiler von $100$ sind, der Fall ist: $1\%, 2\%, 4\%, 5\%, 10\%, 20\%, 25\%, 50\%, 100\%$.

Sei als Beispiel $p = 25\%$ und damit $k = 100/p = 4$. Es werden nur ein Viertel der Eingabedaten eingelesen, indem jeder Eingabedatensatz nur mit einer fünfundzwanzigprozentigen Wahrscheinlichkeit an die Map-Funktion weitergegeben wird. Die nach Filterung und Anwendung der Attribut-Mappings erzeugten abgebildeten Datensätze erhalten die Annotation, dass sie in Aggregationsfunktionen $k = 4$ mal beachtet werden sollen. Bei einer Summenbildung wird beispielsweise der Wert $7$ viermal auf eine Summe addiert, sodass sich diese letztendlich um $28$ erhöht. Ist die Aggregatfunktion stattdessen die Minimum-Funktion, hat die Mehrfachbeachtung keine Auswirkung. Im Allgemeinen wird der Reduce-Funktion der Eindruck vermittelt, dass ein um den Faktor $k$ größeres Zwischenergebnis vorliegt. Daher ist keine weitere Extrapolation der Endergebnisse mehr vonnöten.

Ist $k$ nicht ganzzahlig, muss der Wert gerundet werden. Zur Erhöhung der Genauigkeit kann der nicht-ganzzahlige Anteil als Wahrscheinlichkeit dienen, um entweder auf- oder abzurunden. Sei als Beispiel $p = 16\%$ und damit $k = 6{,}25$. Mit einer Wahrscheinlichkeit von $0{,}25 = 25\%$ durchläuft die Aggregationsschleife für das gegebene Objekt sieben mal, mit fünfundsiebzigprozentiger Wahrscheinlichkeit nur sechs mal.

### Inkrementelles Sampling

Die Pre-Reduce-Extrapolation ist nicht nur generisch durchführbar, sie macht auch *inkrementelles Sampling* möglich. Darunter versteht man das erneute Starten einer Sampling-basierten Transformation, welche auf einer vorherigen Ausführung aufbaut. Mit diesem Verfahren können schnell ungefähre Ergebnisse von Berechnungen ermittelt werden, indem eine sehr kleine Sampling-Größe gewählt wird. Solange einem diese Ergebnisse zu ungenau sind, können diese in eine erneute Berechnung mit höherer Sampling-Größe und damit höherer Genauigkeit einfließen.

Damit hochgerechnete Ergebnisse aus einer vorherigen Iteration $T_0$ in eine inkrementelle Berechnung $T_1$ einfließen können, müssen die im Ergebnis befindlichen Zahlenwerte beim Einlesen zunächst wieder durch $k_0$ geteilt werden. Ansonsten wären sie im Anschluss doppelt hochgerechnet. $k_0$ ist der Faktor $k$ aus Transformation $T_0$. Dienen in $T_1$ nun $p\%$ als neue Stichproben-Eingabemenge, ist $k_1 = 100/p$. Der Faktor $k$ für die gesamte inkrementelle Berechnung berechnet sich daraus wie folgt:

$$k = \frac{1}{\frac{1}{k_0} + \frac{1}{k_1}}$$

Folgt beispielsweise im Anschluss an eine einprozentige Stichprobenberechnung ($k_0 = 100$) eine zweiprozentige ($k_1 = 50$), ist der Faktor $k$ der neuen inkrementellen Berechnung $(100^{-1} + 50^{-1})^{-1} = 33, \overline{3}$. Sie basiert also auf einer Sampling-Größe von drei Prozent. Es ist zu beachten, dass bei der inkrementellen Berechnung manche Eingabedatensätze mehrfach

eingelesen werden können, da es sich um eine Stichprobenberechnung mit Zurücklegen handelt. Trotzdem bietet das Verfahren eine Genauigkeitsverbesserung mit geringeren Geschwindigkeitsverlusten als bei einer nichtinkrementellen Sampling-basierten Tranformation [SSD15].

### Genauigkeitsabschätzung mittels Konfidenzintervallen

Jeder einzelne Ergebniswert einer NotaQL-Transformation, bei der Sampling zum Einsatz kommt, ist aufgrund der unvollständigen Stichprobe sowie aufgrund der Hochrechnung mit einer gewissen Ungenauigkeit behaftet. Je kleiner die Stichprobe gewählt wird, desto schneller ist die Berechnung und desto ungenauer ist das Ergebnis. Damit der Benutzer ein Gefühl dafür bekommt, wie genau oder ungenau ein Ergebniswert ist, kann mittels Abschätzungungen ein Konfidenzintervall hergeleitet werden [Bra15]. In typischen Anwendungen kommt ein 95%-Konfidenzintervall zum Einsatz. Dieses besagt, dass bei immer wieder wiederholender Ausführung der Sampling-basierten NotaQL-Transformation in 95% der Fälle das berechnete Konfidenzintervall den wahren Endwert beinhaltet. Sei als Beispiel eine NotaQL-Transformation gegeben, welche die Durchschnittsgehälter von Personen je Firma berechnet. Eine auf einer kleinen Stichprobe basierende Ausführung liefert als Ergebnis für die Firma *IBM* den Wert $100.000$ sowie ein dazugehöriges 95%-Konfidenzintervall $[70.000, 130.000]$. Das tatsächliche Durchschnittsgehalt ist nun zwar unbekannt, jedoch würden wiederholte Berechnungen in 95% der Fälle Intervalle liefern, die den tatsächlichen Wert beinhalten. Je schmaler das Intervall ist, desto genauer wird der Ergebniswert eingeschätzt.

Sei $n$ die Größe der Stichprobe. Je größer $n$ ist, desto schmaler werden die Intervalle, was sich anhand der Formel zur Berechnung der 95%-Konfidenzintervalle [Kre88, Kapitel 13.2] erkennen lässt:

$$\left[\overline{x} - 2 \cdot \frac{\sigma}{\sqrt{n}}, \overline{x} + 2 \cdot \frac{\sigma}{\sqrt{n}}\right]$$

Neben der Anzahl der verarbeiten Eingabedatensätze $n$ wird die Intevallbreite zudem von der Standardabweichung $\sigma$ der zu aggregierenden Zahlenwerte beeinflusst. Es ist zu beachten, dass sich die gegebene Formel lediglich auf die `AVG`-Funktion anwenden lässt, da Mittelwerte $\overline{x}$ die Zentren der Konfidenzintervalle bilden. Für andere Aggregatfunktionen muss die Formel entsprechend angepasst werden. Des Weiteren ist zu beachten, dass bei der Abschätzung der Konfidenzinvervalle gewisse Annahmen getroffen werden. Zum einen wird eine Normalverteilung der zu aggregierenden Werte angenommen, zum anderen basiert die Formel auf einer entweder unbekannten oder unendlichen Grundmenge. Letzteres ist bei NotaQL-Transformationen zwar eine falsche Annahme, da die Anzahl der Eingabedatensätze zum einen endlich und zum anderen zumindest mittels $N \approx n \cdot 100/p$ ausreichend genau abgeschätzt werden kann. Dennoch sind die ermittelten Konfidenzintervalle durchaus aussagekräftig. Sie geben dem Benutzer einen Eindruck über die Genauigkeit oder Ungenauigkeit der von einer NotaQL-Transformation produzierten Ergebnisse.

### Visualisierung von Transformationsergebnissen

Die Ergebnisse komplexer Datenanalysen können Benutzern in visueller Form präsentiert werden. Die dabei zum Einsatz kommenden Visualiserungstechniken dienen dem Zweck, leicht Aussagen über die analysierten Datenmengen, welche in der Regel enorme Größen annehmen können, machen zu können. In graphischer Form lassen sich Zusammenhänge, Trends und Anomalien besser erkennen, als wenn Daten lediglich in nummerischer oder tabellarischer Form vorliegen. Basieren Analysen auf einer *multidimensionalen Datenmodellierung*, sind zudem interaktive Elemente vonnöten, die es dem Benutzer erlauben, in Diagramme zu zoomen, Filter anzuwenden oder die Gruppierungsgranularität zu variieren [CD97]. Programme wie *Piwik* [KT11] oder *Google Analytics* [Cli12] bieten diese Funktionalitäten speziell für die Zwecke der Web-Analyse. Andere Programme sind generischer Natur und werden als Erweiterungen für Datenbankmanagementsysteme vertrieben. Während es für MySQL, MongoDB und andere Systeme einige spezielle Visualisierungswerkzeuge gibt [Ubi; Cha], lassen sich das *Tableau Visualisierungssystem* [Hee+08] sowie *Periscope* [Per] mit nahezu beliebigen Datenbanken und Diensten verbinden. Das Tableau-System lässt sich hauptsächlich interaktiv mit der Computermaus bedienen. Etwas anders arbeitet Periscope. Es nimmt eine SQL-Anfrage entgegen und liefert neben einer tabellarischen Darstellung von Anfrageergebnissen auch interaktive Diagramme.

Die Erweiterung der NotaQL-Plattform um eine Visualisierungskomponente [BSD15] bietet alle Vorteile wie die soeben genannten Systeme, eröffnet aber gleichzeitig aufgrund ihrer Unterstützung für NoSQL-Datenbanken und flexiblen Schemata neue Möglichkeiten. Des Weiteren macht es die Join-Engine möglich, Daten aus heterogenen Systemen miteinander zu verbinden und die Resultate in graphischer Form zu betrachten. Mittels NotaQL-Streaming-Engines können zudem Datenströme als Eingabe dienen und somit Ströme in Echtzeit visualisiert werden. Abbildung 7.4 zeigt die Einbindung einer Visualisierungskomponente in die NotaQL-Plattform.


**Abbildung 7.4:** Visualisierung von NotaQL-Transformationen

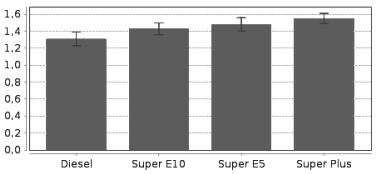



**Abbildung 7.5:** Fehlerbalkendiagramm

Die Grafik zeigt, dass die Visualisierung im Anschluss an eine NotaQL-Transformation folgt. Die Transformationsskripte können dabei alle in dieser Arbeit präsentierten Sprachkonstrukte beinhalten. Dazu zählen beispielsweise Filter, Aggregationen, Transformationsketten, das Aufrufen benutzerdefinierter Funktionen und Verbundoperationen. Dienen die von der NotaQL-Transformation erzeugten Ergebnisse lediglich der Visualisierung und werden nicht anderweitig verwendet, bietet es sich an, die Transformation virtuell auszuführen oder alternativ die Ausgabe-Engine eines In-Memory-Datenbanksystems zu verwenden.

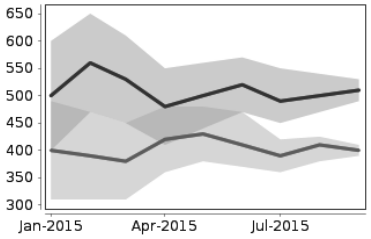



**Abbildung 7.6:** Liniendiagramm

In Diagrammen ist es selten vonnöten, dass die präsentierten Zahlenwerte hundertprozentig genau dargestellt werden. Vor allem bei Linien- und Balkendiagrammen macht eine Abweichung von wenigen Prozent vom wahren Wert meist eine nicht wahrnehmbare Veränderung der Datenpunkte aus. Aus diesem Grund bietet es sich an, bei NotaQL-Transformationen, die zu Visualisierungszwecken ausgeführt werden, Sampling einzusetzen. Die zu den Ergebniswerten bereitgestellten Konfidenzintervalle können in Form von *Whiskers* in *Fehlerbalkendiagrammen* oder als eine gefärbte Fläche in einem Liniendiagramm dargestellt werden (siehe Abbildungen 7.5 und 7.6).

Die Benutzeroberfläche einer um eine Visualisierungskomponente erweiterte NotaQL-Plattform ist in Abbildung 7.7 dargestellt. In der linken Seite des Fensters lassen sich das NotaQL-Skript sowie die Datenquellen und das Transformationsziel eintragen. Auf der rechten Seite wird das Ergebnis in Form eines Diagramms dargestellt. Über diverse Parameter lassen sich Diagrammtyp und Sampling-Eigenschaften verändern. Mittels inkrementellen Samplings ist es möglich, mit einer sehr kleinen Sampling-Rate zu beginnen, um früh ein Ergebnisdiagramm zu sehen. Nach jeder Iteration werden die Diagrammwerte genauer, was sich am Schrumpfen der Fehlerbalken im Diagramm erkennen lässt. Sobald die gewünschte Genauigkeit erreicht ist, kann die Berechnung vom Benutzer unterbrochen und damit beendet werden.

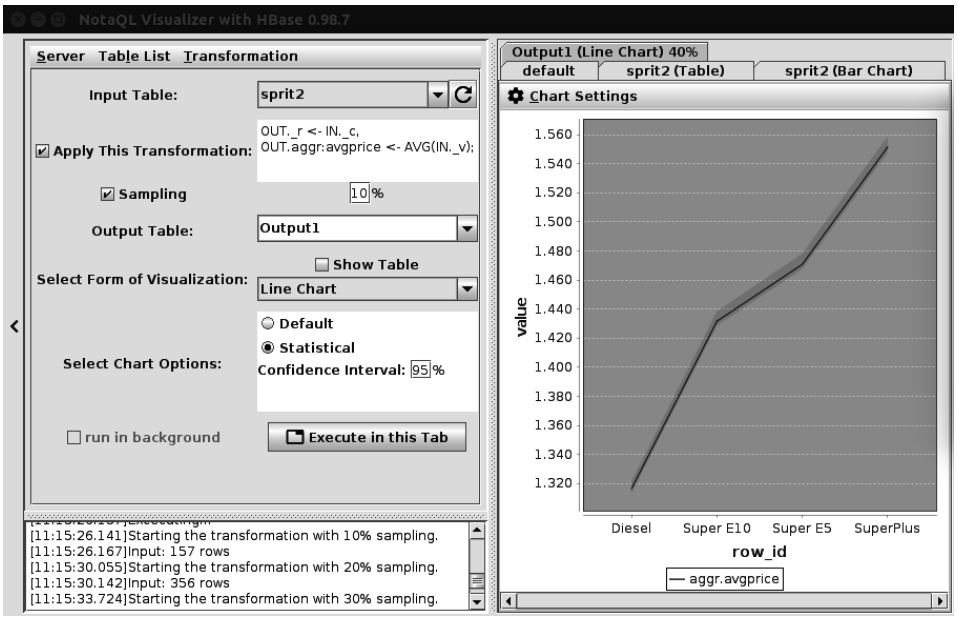



**Abbildung 7.7:** NotaQL-Plattform mit Visualisierungskomponente

# 8 Zusammenfassung

In dieser Arbeit wurde ein Ansatz vorgestellt, wie Datentransformationen auf und zwischen verschiedenartigen NoSQL-Datenbanken definiert und ausgeführt werden können. Andere Sprachen, die zum Teil auf SQL basieren oder aber ebenfalls neuartige Sprachansätze verfolgen, sind oft nur für eine bestimmte Klasse von NoSQL-Datenbanken geeignet und unterstützen keine systemübergreifenden Transformationen. Letztere sind bei System-Migrationen, bei der Informationsintegration und bei ETL-Prozessen vonnöten. Aber auch in Polyglott-Persistence-Umgebungen, in denen eine Kombination mehrerer eigenständiger Datenbanksysteme zum Einsatz kommen, mangelt es an Sprachen, um Analysen durchzuführen, die auf mehreren Systemen basieren, oder um Daten von einem System in ein anderes zu übertragen. Die Umwandlung von Metadaten in Daten und in die Gegenrichtung wird ebenfalls nur von wenigen Sprachen unterstützt. NoSQL-Datenbanken erfordern jedoch diese Art von Transformationen, da dort flexible Schemata vorliegen und häufig keine klare Grenze zwischen Daten und Metadaten gezogen werden kann. Eine strukturelle Heterogenität muss demnäch häufiger überwunden werden als bei Datentransformationen in relationalen Datenbanken.

Die vorgestellte Sprache NotaQL ist nicht für eine bestimmte Klasse von Datenbanken und nicht für ein bestimmtes Datenmodell konzipiert, sondern hat die Absicht, universell einsetzbar zu sein. Mittels Engines und Spracherweiterungen können verschiedenartige Systeme angebunden und komplexe Datentypen unterstützt werden. Eine Engine ist eine Anbindung an ein bestimmtes Datenbanksystem, Dateiformat oder einen Service. Sie dient als eine Art Wrapper zur NotaQL-Plattform und ermöglicht ihr den lesenden und schreibenden Zugriff auf Eingabe- und Ausgabedaten. Eine NotaQL-Transformation liest stets die Eingabe in einem Quellsystem und schreibt die Ausgabe entweder in das selbe oder aber in ein anderes Zielsystem.

Die Hauptelemente der Sprache NotaQL sind Attribut-Mappings. Sie definieren, wie ein Eingabedatensatz in einen Ausgabedatensatz umgewandelt werden soll. Möchte man nicht die gesamte Datenmenge transformieren, kann ein Eingabefilter definiert werden, welcher vor der eigentlichen

Transformation ausgeführt wird. NotaQL ist Zellen-basiert. Das heißt, das kleinste Element einer Transformation ist nicht ein Datensatz, sondern ein Attribut innerhalb eines Datensatzes – eine Zelle. Zunächst wird jeder Eingabedatensatz in seine Zellen zerlegt, anschließend werden diese gemäß den Attribut-Mappings in Ausgabezellen transformiert. Immer wenn mehrere Werte für ein und die selbe Ausgabezelle vorliegen, kommen Aggregatfunktionen zum Einsatz, um ebendiese Werte zu kombinieren und die finalen Ausgabewerte je Zelle zu berechnen. Flexible Schemata werden sowohl seitens der Eingabe als auch der Ausgabe unterstützt. In der Eingabe ist es möglich, über alle Attribute zu iterieren und Metadaten auszulesen. Auf der Ausgabeseite können mit dem Indirektionsoperator dynamisch neue Attribute basierend auf Datenwerten und Eingabeattributen kreiert werden. Diese Sprachkonstrukte ermöglichen mächtige Datentransformationen, ohne dass die einfache Benutzbarkeit darunter leidet. Zur einfachen Benutzbarkeit trägt auch das schlichte Ausführungsmodell bei, welches ähnlich zu dem von MapReduce ist. Für komplexere Analysen kommen Transformationsketten zum Einsatz. Die Zwischenergebnisse solcher Ketten können wahlweise in einer Datenbank persistent gespeichert werden oder aber auch lediglich virtuell vorliegen. Letztere Option ist in der Regel die performantere und bietet zudem Potenzial für Optimierungen.

Anhand von Graphdatenbanken wurde in Kapitel 4.5 gezeigt, dass die Sprache auch bei Datenmodellen Anwendung findet, welche sich von klassischen Aggregat-orientierten Modellen, wie sie in Key-Value-Stores, Wide-Column-Stores, Dokumentendatenbanken sowie auch in relationalen Datenbanken und CSV-Dateien zu finden sind, stark unterscheiden. Dazu wird auf Eingabeseite eine Syntax zur Navigation über die Knoten und Kanten eines Graphen eingesetzt. Während neue Knoten wie einfache Datensätze erstellt werden können, kommt zum Erzeugen von Kanten ein Konstruktor zum Einsatz, der die Eigenschaften der Kante entgegennimmt und ein Prädikat beinhaltet, über welches der Zielknoten ermittelt wird. Für iterative Graph-Berechnungen bietet NotaQL die Möglichkeit, eine Transformation mehrfach auszuführen, bis die berechneten Endwerte für jeden Knoten die gewünschte Genauigkeit erreicht haben. Als Beispiele für solche Berechnungen wurden Algorithmen zur Berechnung des Graphdurchmessers mittels Breitensuche sowie der PageRank-Algorithmus vorgestellt.

Damit nicht nur Datenbanken und Dateien, sondern auch Datenströme in NotaQL unterstützt werden, präsentierten wir in Kapitel 4.9 Datenstromtransformationen. Anders als bei klassischen Transformationen sind diese kontinuierlicher Natur. Sie erhalten laufend Eingabedaten und produzierend kontinuierlich Ausgaben. Über die Definition von Fenstern auf der Eingabeseite können in NotaQL analog wie in anderen Datenstromsprachen Datensätze über einen gewissen Zeitraum gesammelt und schließlich aggregiert werden. Um dabei zum NotaQL-Ausführungsmodell konsistent zu bleiben, wird der Inhalt eines Fensters wie eine persistente Eingabequelle behandelt. Dadurch sind zum einen alle Sprachkonstrukte für klassische Datenbanksysteme auch auf Datenströmen anwendbar, zum anderen lassen sich Joins zwischen Strömen und Datenbanken durchführen oder die Ausgabe von Datenstromtransformationen in eine persistente Datenbank speichern.

Da nur sehr wenige NoSQL-Datenbanken Möglichkeiten zur Datentransformation eingebaut haben, werden in der Praxis für ebendiese Zwecke Datenverarbeitungsframeworks wie MapReduce oder Spark eingesetzt. Die Sprache NotaQL unterstützt die Definition von Transformationen, ohne diese jedoch manuell programmieren zu müssen. Dieser deklarative Ansatz beschleunigt die Erstellung und Wartung von Transformationen und ist weniger anfällig für Fehler. Darüber hinaus können einmal erstellte NotaQL-Skripte auf verschiedene Arten ausgeführt werden. In Kapitel 5 wurden Ansätze präsentiert, die auf MapReduce, Spark oder auf einem Graphframework basieren. Der Spark-basierte Ansatz zeichnete sich als sehr geeignet aus, da hier auch Joins und Transformationsketten effizient unterstützt werden. Bei MapReduce ist bei solchen Transformationen ein Zwischenspeichern auf die Festplatte vonnöten. Ein anderer Ausführungsansatz ist die Übersetzung von NotaQL in SQL. Gewisse Transformationen lassen sich dadurch mit einer hohen Performanz ausführen, andere sind wegen der mangelnden Unterstützung für flexible Schemata in SQL nur ineffizient ausführbar. Die vorgestellten Ausführungsformen ermöglichen materialisierte Transformationen. Als Alternative dazu ist auch eine virtuelle Ausführung denkbar, bei denen ähnlich wie in Datenbanksichten die eigentlichen Berechnungen erst beim Abfragen des Ergebnisses ausgelöst werden. Virtuelle NotaQL-Transformationen bieten sich an, um Analysen, auf deren Ergebnis lediglich selten und gezielt zugegriffen wird, zu beschleunigen. Dadurch, dass die Ergebnisse nicht vorberechnet werden, können die Ergebnisse auch nicht veralten. Bei sich ändernden Anforderungen oder wachsenden Datenmengen kann im Nachhinein auf eine materialisierte Ausführung gewechselt werden. Ein Wechsel zwischen den verschiedenen Ausführungsarten ist jederzeit möglich, ohne ein NotaQL-Skript anpassen zu müssen.

Als eine weitere Dimension der Ausführung unterstützt NotaQL die inkrementelle Berechnung. Ähnlich wie bei der Selbstwartbarkeit materialisierter Sichten in relationalen Datenbanken ist die NotaQL-Plattform in der Lage, Änderungen in den Basisdaten auf vormals berechnete Transformationsergebnisse einzubringen, ohne die Transformation vollständig neu starten zu müssen. Aufgrund der einfachen Ausführungssemantik von NotaQL-Skripten ist in der Regel jedes NotaQL-Skript inkrementell ausführbar. Es hängt lediglich vom System ab, in dem sich die Eingabedaten befinden, welche Techniken der Änderungserfassung angewandt werden können. Die verschiedenen Techniken wurden in Kapitel 6 vorgestellt. Anhand der Datenbanksysteme HBase und MongoDB wurde die Zeitstempelbasierte Änderungserfassung erläutert. Die in den Eingabedaten zu findenden Zeitstempel der letzten Änderung dienen der NotaQL-Plattform zum Finden der Änderungen seit einer vorherigen Ausführung. Ebenfalls anhand von MongoDB wurden Datenbank-Logs zu Hilfe genommen, um daraus die Änderungen zu ermitteln. Unterstützt die als Eingabe verwendete Datenbank Trigger, können diese zum Einsatz kommen, um aus Einfügungen, Änderungen und Löschungen einen Datenstrom zu erzeugen, welcher das Transformationsergebnis inkrementell wartet. Während diese Verfahren in der Regel spezielle Kriterien an das Eingabedatenbanksystem stellen, ist eine Snapshot-basierte Änderungserfassung stets möglich. Dabei

werden die Änderungen anhand der Differenz des aktuellen Datenbankzustands zu einem bei der vorherigen Ausführung erzeugten Datenbankschnappschuss ermittelt.

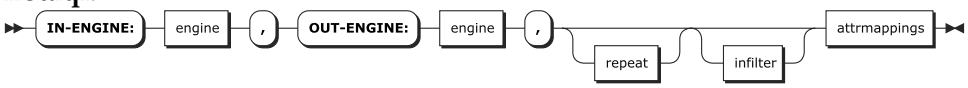
Das Arbeiten mit der Sprache NotaQL teilt sich in zwei Teile auf. Zum einen ist eine Entwicklung einer Engine vonnöten, um neue Systeme anzubinden, damit diese als Ein- und Ausgabe in NotaQL-Skript zum Einsatz kommen können. Dieser Teil ist jedoch generisch und muss nur einmalig erfolgen. Der zweite Teil ist die Definition der NotaQL-Skripte. Dieser Teil erfordert keine Programmierkenntnisse und ist meist mit wenigen Code-Zeilen erledigt. Die graphische Benutzeroberfläche der NotaQL-Plattform unterstützt den Anwender bei der Erstellung und Wartung von Skripten und bietet diverse inkrementelle und nicht-inkrementelle Ausführungsalternativen. Die in Kapitel 6.2 vorgestellte Advisor-Komponente verfolgt den Zweck, unter den zur Verfügung stehenden Ausführungsarten stets die performanteste zu wählen.

Während sich diese Arbeit hauptsächlich mit der Sprache an sich und deren Ausführung beschäftigt hat, sind offene Fragestellungen bezüglich der Optimierung und Konsistenz von Transformationen Themen für die Zukunft. Diese Arbeit bildet die Grundlage für zukünftige Sprachen, um die Herausforderungen zu meistern, welche NoSQL-Datenbanken mitbringen. Zu diesen Herausforderungen zählen die Schema-Flexibilität, die Heterogenität auf Schema-, Daten- und Zugriffsebene sowie die Unterstützung verschiedenartiger Datenmodelle. Des Weiteren können die inkrementellen und nicht-inkrementellen Ausführungsmethoden für die Sprache NotaQL leicht auf andere Sprachen übertragen werden. Bisher ist mit existierenden Ansätzen keine automatische inkrementelle Ausführung möglich. Mit den in dieser Arbeit vorgestellten Ansätzen können existierende und neue Sprachen um eine solche Funktionalität erweitert werden.

# A

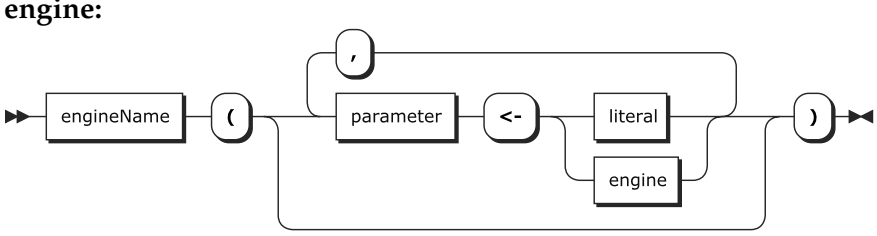
# NotaQL-Grammatik

**notaql:**



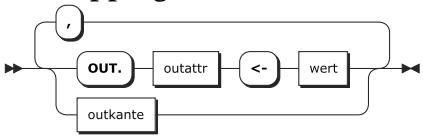

```
notaql ::= 'IN-ENGINE:' engine ','
           'OUT-ENGINE:' engine ','
           repeat? infilter? attrmappings
```

**engine:**



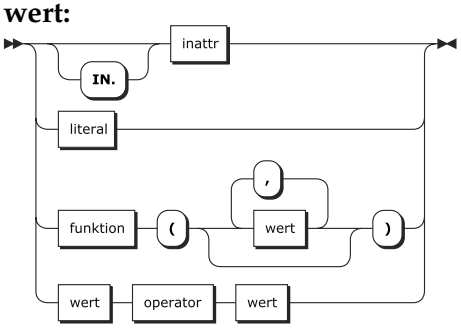

```
engine ::= engineName '('
            ((parameter '<-' (literal|engine))
             (',' parameter '<-' (literal|engine))*)?
           ')'
```

**attrmappings:**



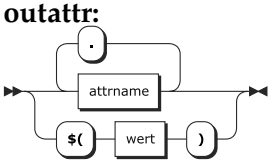

```
attrmappings ::= ( 'OUT.' outattr '<-' wert
                 | outkante )
                 ( ',' ( 'OUT.' outattr '<-' wert
                 | outkante ) )*
```

**wert:**



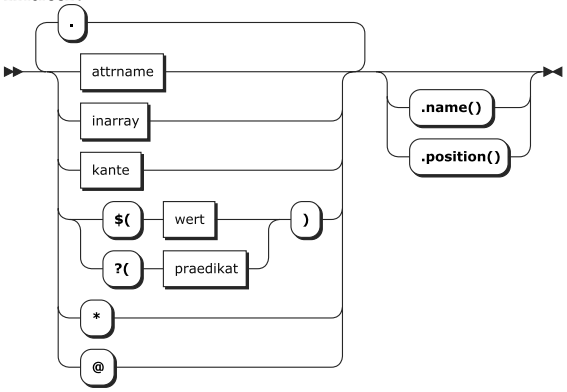

```
wert ::= ( 'IN.'? inattr
         | literal
         | funktion '(' (wert (',' wert)*)? ')'
         | wert operator wert )
```

**outattr:**



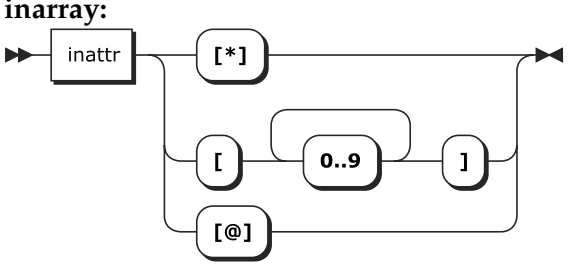

```
outattr ::= ( attrname ( '.' attrname )*
            | '$(' wert ')' )
```

**inattr:**



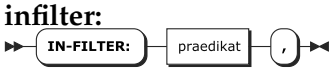

```
inattr ::= ( attrname | inarray | kante
           | ( '$(' wert | '?(' praedikat ) ')'
           | '*' | '@' )
           ( '.' ( attrname | inarray | kante
           | ( '$(' wert | '?(' praedikat ) ')'
           | '*' | '@' ) )*
           ( '.name()' | '.position()' )?
```

**inarray:**



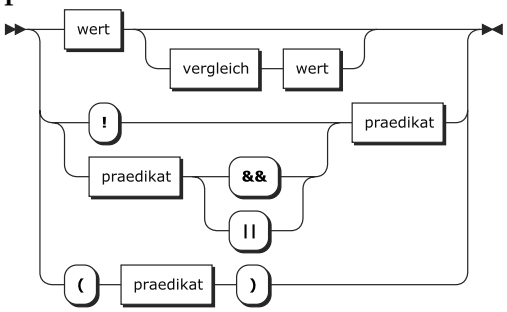

```
inarray  ::= inattr ( '[*]' | '[' '0..9'+ ']' | '[@]' )
```

**infilter:**



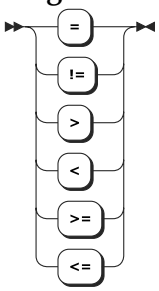

```
infilter ::= 'IN-FILTER: ' praedikat ','
```

**praedikat:**



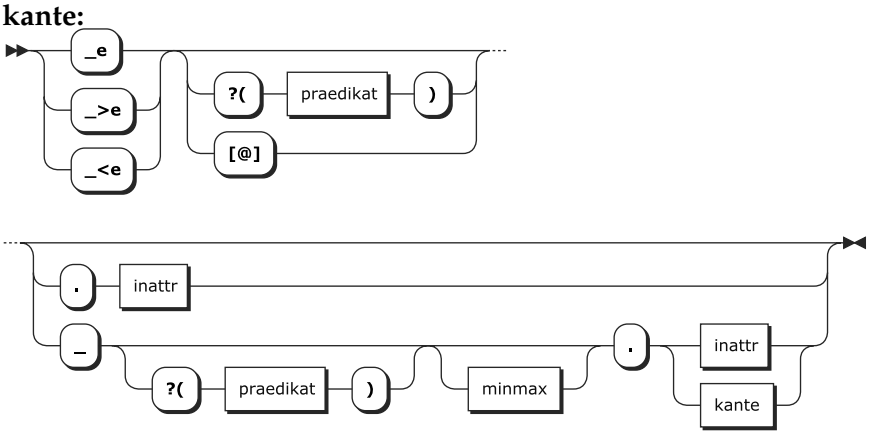

```
praedikat ::= wert (vergleich wert)?
            | '!' praedikat
            | '(' praedikat ')'
            | praedikat ('&&'|'||') praedikat
```

**vergleich:**



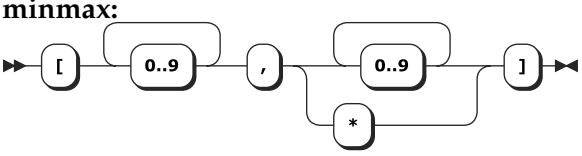

```
vergleich ::= '='|'!='|'>'|'<'|'>='|'<='
```

**kante:**



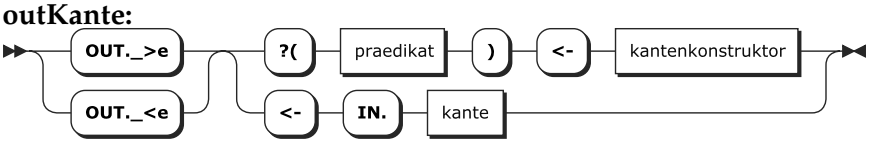

```
kante ::=
  ('_e'|'_>e'|'_<e') ( '?(' praedikat ')' | '[@]')?
   ( '.' inattr
   | '_' ( '?(' praedikat ')' )?
     minmax? '.' (  inattr | kante )
 )?
```

**minmax:**



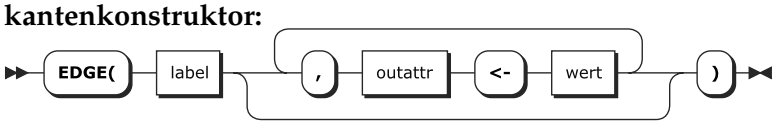

```
minmax ::= '[' '0..9'+ ',' ('0..9'+|'*') ']'
```

**outKante:**



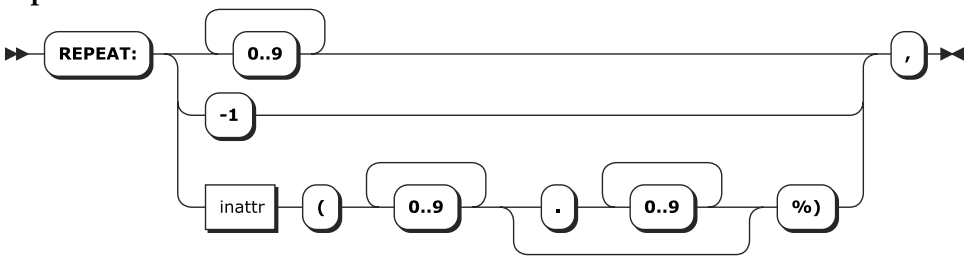

```
outKante ::= ('OUT._>e'|'OUT._<e')
             ( '?(' praedikat ')' '<-'
               kantenkonstruktor
             | '<-' 'IN.' kante )
```

**kantenkonstruktor:**



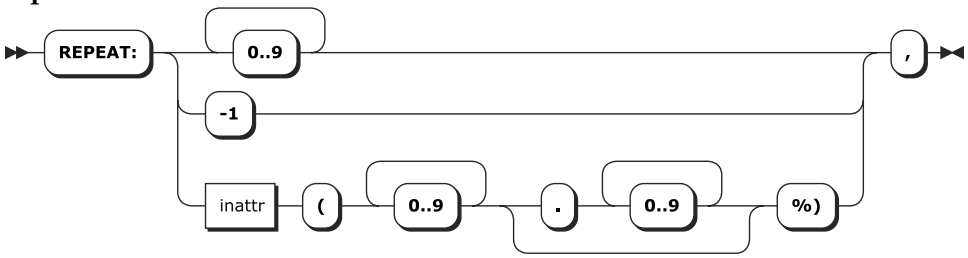

```
kantenkonstruktor ::= 'EDGE(' label
                      (',' prop. '<-' wert)* ')'
```

**repeat:**




```
repeat ::= 'REPEAT:' ( '0..9'+ | '-1'
           | inattr '(' '0..9'+ ( '.' '0..9'+ )? '%)'
           ) ','
```

**repeat:**




```
repeat ::= 'REPEAT:' ( '0..9'+ | '-1'
           | inattr '(' '0..9'+ ( '.' '0..9'+ )? '%)'
           ) ','
```

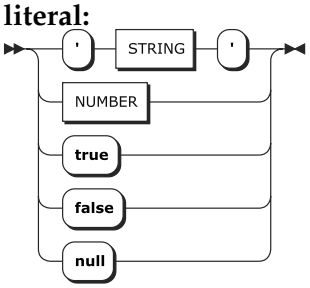
**engineName:**


```
engineName ::= STRING
```

**parameter:**

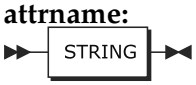

```
parameter ::= STRING
```

**literal:**




```
literal ::= "'" STRING "'" | NUMBER
            | 'true' | 'false' | 'null'
```

**attrname:**

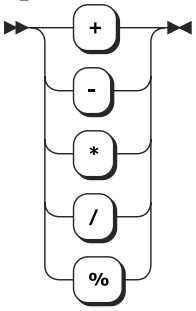



```
attrname ::= STRING
```

**funktion:**


```
funktion ::= STRING
```

**operator:**




```
operator ::= '+' | '-' | '*' | '/' | '%'
```

# Literatur


[ABW06] Arvind Arasu, Shivnath Babu und Jennifer Widom. „The CQL continuous query language: Semantic foundations and query execution“. In: *The VLDB Journal-The International Journal on Very Large Data Bases* 15.2 (2006), S. 121–142.

[Amb12] Scott Ambler. *Agile database techniques: Effective strategies for the agile software developer*. John Wiley & Sons, 2012.

[Ara] ArangoDB. URL: `https://www.arangodb.com`.

[ASL89] AM Alashqur, Stanley YW Su und Herman Lam. „OQL: a query language for manipulating object-oriented databases“. In: *Proceedings of the 15th international conference on Very large data bases*. Morgan Kaufmann Publishers Inc. 1989, S. 433–442.

[ASX01] Rakesh Agrawal, Amit Somani und Yirong Xu. „Storage and querying of e-commerce data“. In: *VLDB*. Bd. 1. 2001, S. 149–158.

[Ave11] Ching Avery. „Giraph: Large-scale graph processing infrastructure on hadoop“. In: *Proceedings of the Hadoop Summit. Santa Clara* 11 (2011).

[Bas14] Kenny Bastani. „Using Apache Spark and Neo4j for Big Data Graph Analytics“. In: (2014). URL: `http://www.kennybastani.com/2014/11/using-apache-spark-and-neo4j-for-big.html`.

[Beh+11] Alexander Behm u. a. „Asterix: towards a scalable, semistructured data platform for evolving-world models“. In: *Distributed and Parallel Databases* 29.3 (2011), S. 185–216.

[Ber+03] Ralph Bergmann u. a. *Developing industrial case-based reasoning applications: The INRECA methodology*. Springer, 2003.

[Bha15] Sougata Bhattacharjee. „Views and Virtual Table Transformations on HBase“. Bachelorarbeit, TU Kaiserslautern. Magisterarb. TU Kaiserslautern, 2015.

[Blo70] Burton H. Bloom. „Space/time trade-offs in hash coding with allowable errors“. In: *Commun. ACM* (1970).

[Bor+15] Vinayak Borkar u. a. „Algebricks: a data model-agnostic compiler backend for Big Data languages“. In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM. 2015, S. 422–433.

[Bra15] Stefan Braun. *Visualisierung von NotaQL-Transformationen unter der Verwendung von Sampling-Techniken*. Bachelorarbeit, TU Kaiserslautern. 2015.

[Bru16] Peter Brucker. *Eine Join-Input-Engine für systemübergreifende NoSQL-Datentransformationen*. Bachelorarbeit, TU Kaiserslautern. 2016.

[BSD15] Stefan Braun, Johannes Schildgen und Stefan Deßloch. „Visualisierung von NoSQL-Transformationen unter der Verwendung von Sampling-Techniken.“ In: *LWA*. 2015, S. 427–438.

[Cal] Apache Calcite. URL: `http://calcite.incubator.apache.org`.

[CD97] Surajit Chaudhuri und Umeshwar Dayal. „An overview of data warehousing and OLAP technology“. In: *ACM Sigmod record* 26.1 (1997), S. 65–74.

[Cha] Cloud9 Charts. „MongoDB Reporting & Analytics“. In: (). URL: `https://www.cloud9charts.com`.

[Cha+03a] Don Chamberlin u. a. „XQuery: A query language for XML“. In: *SIGMOD Conference*. Bd. 682. 2003.

[Cha+03b] Sirish Chandrasekaran u. a. „TelegraphCQ: Continuous Dataflow Processing for an Uncertain World.“ In: *CIDR*. 2003.

[Cha+08a] Don Chamberlin u. a. „XQuery update facility 1.0“. In: *W3C Candidate Recommendation* 1 (2008).

[Cha+08b] Fay Chang u. a. „Bigtable: A distributed storage system for structured data“. In: *ACM Transactions on Computer Systems (TOCS)* 26.2 (2008), S. 4.

[Cha00] Miranda Chan. „Incremental update to aggregated information for data warehouses over internet“. In: *In: 3rd ACM International Workshop on Data Warehousing and OLAP (DOLAP '00). McLean, Virginia, United States*. 2000, S. 57–64.

[CK85] George P Copeland und Setrag N Khoshafian. „A decomposition storage model“. In: *ACM SIGMOD Record*. Bd. 14. 4. ACM. 1985, S. 268–279.

[CL13] Arnaud Castelltort und Anne Laurent. „Representing history in graph-oriented nosql databases: A versioning system“. In: *Digital Information Management (ICDIM), 2013 Eighth International Conference on*. IEEE. 2013, S. 228–234.

[Cla+99] James Clark u. a. „Xsl transformations (xslt)“. In: *World Wide Web Consortium (W3C)* (1999). URL: `http://www.w3.org/TR/xslt`.

[Cli12] Brian Clifton. *Advanced web metrics with Google Analytics*. John Wiley & Sons, 2012.

[Coh+01] Edith Cohen u. a. „Finding interesting associations without support pruning“. In: *Knowledge and Data Engineering, IEEE Transactions on* 13.1 (2001), S. 64–78.

[Cur+08] Carlo A Curino u. a. „Schema evolution in wikipedia: toward a web information system benchmark“. In: *In International Conference on Enterprise Information Systems (ICEIS*. Citeseer. 2008.

[Cur+11] Olivier Curé u. a. „Data Integration over NoSQL Stores Using Access Path Based Mappings.“ In: *DEXA (1)*. Hrsg. von Abdelkader Hameurlain u. a. Bd. 6860. Lecture Notes in Computer Science. Springer, 2011, S. 481–495.

[DG04] Jeffrey Dean und Sanjay Ghemawat. „MapReduce: Simplified Data Processing on Large Clusters“. In: *OSDI* (2004), S. 137–150.

[DG92] David DeWitt und Jim Gray. „Parallel database systems: the future of high performance database systems“. In: *Communications of the ACM* 35.6 (1992), S. 85–98.

[Dro16] Voichita Droanca. „Log-based Change-Data Capture on MongoDB“. Masterarbeit, TU Kaiserslautern. Magisterarb. TU Kaiserslautern, 2016.

[Edl+10] Stefan Edlich u. a. *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. Hanser Fachbuchverlag, 2010.

[EM01] Andrew Eisenberg und Jim Melton. „SQL/XML and the SQLX Informal Group of Companies“. In: *Sigmod Record* 30.3 (2001), S. 105–108.

[Emd14] Michael Emde. *GUI und Testumgebung für die HBase-Schematransformationssprache NotaQL*. Bachelorarbeit, TU Kaiserslautern. 2014.

[Emd16] Michael Emde. „A High-Level Platform for Analytics on Data Streams“. Bachelorarbeit, TU Kaiserslautern. Magisterarb. TU Kaiserslautern, 2016.

[Fel+14] Pascal Felber u. a. „On the Support of Versioning in Distributed Key-Value Stores“. In: *Reliable Distributed Systems (SRDS), 2014 IEEE 33rd International Symposium on*. IEEE. 2014, S. 95–104.

[FLG12] Leonidas Fegaras, Chengkai Li und Upa Gupta. „An optimization framework for map-reduce queries“. In: *Proceedings of the 15th International Conference on Extending Database Technology*. ACM. 2012, S. 26–37.

[Fox16] Dominic Fox. „How Not To Use Cassandra Like An RDBMS“. In: (2016). URL: `https://opencredo.com/how-not-to-use-cassandra-like-an-rdbms-and-what-will-happen-if-you-do`.

[FRP15] Jing Fan, Adalbert Gerald Soosai Raj und Jignesh M Patel. „The Case Against Specialized Graph Analytics Engines.“ In: *CIDR*. 2015.

[GJM96] Ashish Gupta, Hosagrahar V Jagadish und Inderpal Singh Mumick. „Data integration using self-maintainable views“. In: *Advances in Database Technology—EDBT'96*. Springer, 1996.

[GM95] A. Gupta und I. S. Mumick. „Maintenance of Materialized Views: Problems, Techniques, and Applications“. In: *IEEE Data Engineering Bulletin* 18.2 (1995), S. 3–18.

[Gra13] Tugdual Grall. *Implement Document Versioning with Couchbase*. 2013. URL: http://blog.couchbase.com/how-implement-document-versioning-couchbase.

[Gro+13] Katarina Grolinger u. a. „Data management in cloud environments: NoSQL and NewSQL data stores“. In: *Journal of Cloud Computing: Advances, Systems and Applications* 2.1 (2013), S. 1.

[Gro07] W3C XSL/XML Query Working Groups. *The XPath 2.0 standard*. W3C Recommendation. 2007. URL: http://www.network-theory.co.uk/w3c/xpath.

[Hau+17] Florian Haubold u. a. „ControVol Flex: Flexible Schema Evolution for NoSQL Application Development“. In: *BTW*. 2017.

[Hee+08] Jeffrey Heer u. a. „Graphical histories for visualization: Supporting analysis, communication, and evaluation“. In: *IEEE transactions on visualization and computer graphics* 14.6 (2008).

[Hiv] *Hive as a contrib project*. 2008. URL: https://issues.apache.org/jira/browse/HADOOP-3601.

[HMH01] Mauricio A Hernández, Renée J Miller und Laura M Haas. „Clio: A semi-automatic tool for schema mapping“. In: *ACM SIGMOD Record* 30.2 (2001), S. 607.

[Hof13] Steve Hoffman. *Apache Flume: Distributed Log Collection for Hadoop*. Packt Publishing Ltd, 2013.

[Hon+12] Sungpack Hong u. a. „Green-Marl: a DSL for easy and efficient graph analysis“. In: *ACM SIGARCH Computer Architecture News*. Bd. 40. 1. ACM. 2012, S. 349–362.

[HR05] Theo Härder und Erhard Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung, 2. Auflage*. Springer, 26. Sep. 2005. ISBN: 3-540-42133-5.

[HR83] Theo Haerder und Andreas Reuter. „Principles of Transaction-oriented Database Recovery“. In: *ACM Comput. Surv.* 15.4 (Dez. 1983), S. 287–317.

[Jac86] Peter Jackson. „Introduction to expert systems“. In: (1986).

[JMS95] H. V. Jagadish, Inderpal Singh Mumick und Abraham Silberschatz. „View Maintenance Issues for the Chronicle Data Model.“ In: *PODS*. Hrsg. von Mihalis Yannakakis. ACM Press, 1995.

[Jör+11a] Thomas Jörg u. a. „Can MapReduce learn form Materialized Views?“ In: *LADIS 2011*. 2011, S. 1–5.

[Jör+11b] Thomas Jörg u. a. „Incremental Recomputations in MapReduce“. In: *CloudDB 2011*. 2011.

[Kim11] Aaron Kimball. *Real-time Streaming Analysis for Hadoop and Flume*. 2011.

[KM12] Krishna Kulkarni und Jan-Eike Michels. „Temporal features in SQL: 2011“. In: *ACM Sigmod Record* 41.3 (2012), S. 34–43.

[Kol+15] Boyan Kolev u. a. „CloudMdsQL: Querying heterogeneous cloud data stores with a common language“. In: *Distributed and Parallel Databases* (2015), S. 1–41.

[Kor+16] Marcel Kornacker u. a. „Impala: Eine moderne, quellen-offene SQL Engine für Hadoop“. In: *Big Data*. Springer, 2016, S. 159–178.

[KR11] Ralph Kimball und Margy Ross. *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, 2011.

[Kre88] Ulrich Krengel. *Einführung in die Wahrscheinlichkeitstheorie und Statistik*. Bd. 8. Springer, 1988.

[KS09] Jürgen Krämer und Bernhard Seeger. „Semantics and implementation of continuous sliding window queries over data streams.“ In: *ACM Trans. Database Syst.* 34.1 (21. Apr. 2009).

[KT11] Moritz Karg und Sven Thomsen. „Einsatz von Piwik bei der Reichweitenanalyse“. In: *Datenschutz und Datensicherheit-DuD* 35.7 (2011), S. 489.

[Lan01] Doug Laney. „3D data management: Controlling data volume, velocity and variety“. In: *META Group Research Note* 6 (2001), S. 70.

[LD10] Jimmy Lin und Chris Dyer. „Data-intensive text processing with MapReduce“. In: *Synthesis Lectures on Human Language Technologies* 3.1 (2010), S. 1–177.

[Lee+11] Kyong-Ha Lee u. a. „Parallel data processing with MapReduce: a survey.“ In: *SIGMOD Record* (2011).

[LGM96] Wilburt Labio und Hector Garcia-Molina. „Efficient snapshot differential algorithms in data warehousing“. In: (1996).

[Lin+86] Bruce Lindsay u. a. „A Snapshot Differential Refresh Algorithm“. In: *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*. SIGMOD '86. Washington, D.C., USA: ACM, 1986, S. 53–60. ISBN: 0-89791-191-1.

[Lin13] Jimmy Lin. „Mapreduce is good enough? if all you have is a hammer, throw away everything that's not a nail!“ In: *Big Data* (2013).

[Liu+14] Chang Liu u. a. „Automating distributed partial aggregation“. In: *Proceedings of the ACM Symposium on Cloud Computing*. ACM. 2014, S. 1–12.

[LN07] Ulf Leser und Felix Naumann. *Informationsintegration – Architekturen und Methoden zur Integration verteilter und heterogener Datenquellen*. dpunkt, Heidelberg, 2007.

[Lot15] Thomas Lottermann. „Cross-System NoSQL Data Transformations“. Bachelorarbeit, TU Kaiserslautern. Magisterarb. Technische Universität Kaiserslautern, 2015.

[LS99] Ora Lassila und Ralph R Swick. „Resource description framework (RDF) model and syntax specification“. In: (1999).

[LSS96] Laks VS Lakshmanan, Fereidoon Sadri und Iyer N Subramanian. „SchemaSQL-a language for interoperability in relational multi-database systems“. In: *VLDB*. Bd. 96. 1996, S. 239–250.

[LZZ12] Nikolay Laptev, Kai Zeng und Carlo Zaniolo. „Early Accurate Results for Advanced Analytics on MapReduce." In: *PVLDB* 5.10 (2012), S. 1028–1039.

[Ma+16] Shuai Ma u. a. „Big graph search: challenges and techniques". In: *Frontiers of Computer Science* 10.3 (2016), S. 387–398.

[Mal+10] Grzegorz Malewicz u. a. „Pregel: a system for large-scale graph processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, S. 135–146.

[Mana] MongoDB Manual. *Configure a Delayed Replica Set Member*. URL: https://docs.mongodb.com/manual/tutorial/configure-a-delayed-replica-set-member.

[Manb] MySQL 5.6 Refenerence Manual. *Delayed Replication*. URL: https://dev.mysql.com/doc/refman/5.6/en/replication-delayed.html.

[MBR01] Jayant Madhavan, Philip A Bernstein und Erhard Rahm. „Generic schema matching with cupid". In: *VLDB*. Bd. 1. 2001, S. 49–58.

[MC16] David Matos und Miguel Correia. In: *IEEE Data Eng. Bull.* (2016).

[McH99] Peter McHugh. *Making it Big in Software: A guide to success for software vendors with growth ambitions*. Rubic, 1999.

[ME01] Jim Melton und Andrew Eisenberg. „SQL multimedia and application packages (SQL/MM)". In: *ACM Sigmod Record* 30.4 (2001), S. 97–102.

[Mel+02] Jim Melton u. a. „SQL/MED: a status report". In: *ACM SIGMOD Record* 31.3 (2002), S. 81–89.

[Mel+11] Sergey Melnik u. a. „Dremel: interactive analysis of web-scale datasets". In: *Communications of the ACM* 54.6 (2011), S. 114–123.

[MHH00] Renée J Miller, Laura M Haas und Mauricio A Hernández. „Schema Mapping as Query Discovery." In: *VLDB*. Bd. 2000. 2000, S. 77–88.

[Mul] Multicorn. *Unified data access library*. URL: http://multicorn.org.

[Nad+99] Prakash M Nadkarni u. a. „Organization of heterogeneous scientific data using the EAV/CR representation". In: *Journal of the American Medical Informatics Association* 6.6 (1999), S. 478–493.

[Neu13] Peter Neubauer. *Should I learn Cypher or Gremlin for operating a Neo4j database?* 2013. URL: https://www.quora.com/Should-I-learn-Cypher-or-Gremlin-for-operating-a-Neo4j-database.

[OHR15] David Ostrovsky, Mohammed Haji und Yaniv Rodenski. „The N1QL Query Language". In: *Pro Couchbase Server*. Springer, 2015, S. 107–133.

[Ols+08] Christopher Olston u. a. „Pig latin: a not-so-foreign language for data processing". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM. 2008.

[Ols+11] Christopher Olston u. a. „Nova: continuous pig/hadoop workflows“. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM. 2011, S. 1081–1090.

[OPV14] Kian Win Ong, Yannis Papakonstantinou und Romain Vernoux. „The SQL++ Query Language: Configurable, Unifying and Semi-structured“. In: *arXiv preprint arXiv:1405.3631* (2014).

[Orc] *Apache ORC*. URL: `https://orc.apache.org`.

[Owe03] Michael Owens. „Embedding an SQL database with SQLite“. In: *Linux Journal* 2003.110 (2003), S. 2.

[Pag+99] Lawrence Page u. a. *The PageRank citation ranking: Bringing order to the web.* Techn. Ber. Stanford InfoLab, 1999.

[Per] Periscope. *From SQL Query to Analysis in Seconds*. URL: `https://www.periscopedata.com`.

[Pho] *Phoenix: Performance*. URL: `https://phoenix.apache.org/performance.html`.

[Phy13] Physalix. *Datas manipulation in MongoDB: rename field, change type, add sub-document*. 2013. URL: `http://physalix.com/datas-manipulation-in-mongodb-rename-field-change-type-add-sub-document`.

[Pik+05] Rob Pike u. a. „Interpreting the data: Parallel analysis with Sawzall“. In: *Scientific Programming* 13.4 (2005), S. 277–298.

[PS+08] Eric Prud'Hommeaux, Andy Seaborne u. a. „SPARQL query language for RDF“. In: *W3C recommendation* 15 (2008).

[RB01] Erhard Rahm und Philip A Bernstein. „A survey of approaches to automatic schema matching“. In: *the VLDB Journal* 10.4 (2001), S. 334–350.

[RG00] Raghu Ramakrishnan und Johannes Gehrke. *Database management systems*. 2. Aufl. McGraw-Hill, 2000.

[RLMW14] Julian Rith, Philipp S. Lehmayr und Klaus Meyer-Wegener. „Speaking in Tongues: SQL Access to NoSQL Systems“. In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. SAC '14. Gyeongju, Republic of Korea: ACM, 2014, S. 855–857. ISBN: 978-1-4503-2469-4. DOI: `10.1145/2554850.2555099`.

[Rob+12] Jonathan Robie u. a. „JSONiq: XQuery for JSON“. In: *JSON for XQuery* (2012), S. 63–72.

[Rod15] Marko A Rodriguez. „The Gremlin graph traversal machine and language“. In: *Proceedings of the 15th Symposium on Database Programming Languages*. ACM. 2015, S. 1–10.

[Ron15] Trace Ronning. *Fop 6 Features of a Cloud-to-Cloud Backup Solution*. 2015.

[Ros13] Michael Rose. *Migrating from MongoDB to Cassandra*. 2013. URL: `https://www.fullcontact.com/blog/mongo-to-cassandra-migration`.

[RS97] Mary Tork Roth und Peter M Schwarz. „Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources." In: *VLDB*. Bd. 97. 1997, S. 25–29.

[Sah+15] Bikas Saha u. a. „Apache tez: A unifying framework for modeling and building data processing applications". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, S. 1357–1369.

[Sat12] Kazunori Sato. „An inside look at google bigquery". In: *White paper* (2012).

[SBH13] Caetano Sauer, Sebastian Bächle und Theo Härder. „Versatile xquery processing in mapreduce". In: *East European Conference on Advances in Databases and Information Systems*. Springer. 2013, S. 204–217.

[SCA15] Stefanie Scherzinger, Thomas Cerqueus und Eduardo Cunha de Almeida. „ControVol: A framework for controlled schema evolution in NoSQL application development". In: *2015 IEEE 31st International Conference on Data Engineering*. IEEE. 2015, S. 1464–1467.

[Sch+14] Johannes Schildgen u. a. „Marimba: A Framework for Making MapReduce Jobs Incremental". In: *2014 IEEE International Congress on Big Data*. IEEE. 2014.

[Sch15] Nico Schäfer. *An SQL-based Intermediate Layer for Table Transformations on Wide-Column Stores*. Bachelorarbeit, TU Kaiserslautern. 2015.

[SD15] Johannes Schildgen und Stefan Deßloch. „NotaQL Is Not a Query Language! It's for Data Transformation on Wide-Column Stores". In: *British International Conference on Databases - BICOD 2015*. Juli 2015.

[SD16] Johannes Schildgen und Stefan Deßloch. „Heterogenität überwinden mit der Datentransformationssprache NotaQL". In: *Datenbank-Spektrum* 16.1 (2016), S. 5–15.

[Seo+10] Sangwon Seo u. a. „Hama: An efficient matrix computation with the mapreduce framework". In: *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE. 2010, S. 721–726.

[SF12] Pramod J. Sadalage und Martin Fowler. *NoSQL Distilled: A brief guide to the emerging world of polyglot persistence*. 1st. Addison-Wesley Professional, 2012.

[SKD17] Johannes Schildgen, Yannick Krück und Stefan Deßloch. „Transformations on Graph Databases for Polyglot Persistence with NotaQL". In: *BTW*. 2017.

[SKS13] Stefanie Scherzinger, Meike Klettke und Uta Störl. „Managing schema evolution in NoSQL data stores". In: *arXiv preprint arXiv:1308.0514* (2013).

[SLD16] Johannes Schildgen, Thomas Lottermann und Stefan Deßloch. „Cross-system NoSQL data transformations with NotaQL“. In: *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*. ACM. 2016, S. 5.

[SSD15] Marc Schäfer, Johannes Schildgen und Stefan Deßloch. „Sampling with Incremental MapReduce.“ In: Workshop on Big Data in Science (BigDS) @ BTW, 2015.

[STD16] Marco Scavuzzo, Damian A. Tamburri und Elisabetta Di Nitto. „Providing Big Data Applications with Fault-tolerant Data Migration Across Heterogeneous NoSQL Databases“. In: *Proceedings of the 2Nd International Workshop on BIG Data Software Engineering*. BIGDSE '16. Austin, Texas: ACM, 2016.

[Sud76] Seymour Sudman. *Applied sampling*. Academic Press New York, 1976.

[Sun+15] Wen Sun u. a. „SQLGraph: an efficient relational-based property graph store“. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, S. 1887–1901.

[Süß15] Fabian Süß. *A Web-based Sandbox for NotaQL Table Transformations*. Bachelorarbeit, TU Kaiserslautern. 2015.

[Ter+92] Douglas Terry u. a. „Continuous queries over append-only databases“. In: *SIGMOD '92*. San Diego, California, United States: ACM, 1992, S. 321–330. ISBN: 0-89791-521-6. DOI: 10.1145/130283.130333.

[Tha16] Philipp Thau. „Eine Advisor-unterstütze Plattform für inkrementelle Datentransformationen“. Bachelorarbeit, TU Kaiserslautern. Magisterarb. TU Kaiserslautern, 2016.

[Thu+09] Ashish Thusoo u. a. „Hive: a warehousing solution over a map-reduce framework“. In: *Proceedings of the VLDB Endowment* 2.2 (2009), S. 1626–1629.

[Tia+13] Yuanyuan Tian u. a. „From "Think Like a Vertex"to "Think Like a Graph".“ In: *PVLDB* 7.3 (2013), S. 193–204.

[Ubi] Ubiq. *Reporting & Analytics for MySQL,PostgreSQL*. URL: http://ubiq.co.

[Wei12] Claudius Weinberger. *Is UNQL Dead*. 2012. URL: https://www.arangodb.com/2012/04/is_unql_dead.

[Wie15] Lena Wiese. *Advanced Data Management for SQL, NoSQL, Cloud and Distributed Databases*. DeGruyter, 2015.

[WL97] Ke Wang und Huiqing Liu. „Schema Discovery for Semistructured Data.“ In: *KDD*. Bd. 97. 1997, S. 271–274.

[WR05] Catharine M Wyss und Edward L Robertson. „Relational languages for metadata integration“. In: *ACM Transactions on Database Systems (TODS)* 30.2 (2005), S. 624–660.

[Xin+13] Reynold S Xin u. a. „Graphx: A resilient distributed graph system on spark“. In: *First International Workshop on Graph Data Management Experiences and Systems*. ACM. 2013, S. 2.

[Zah+10] Matei Zaharia u. a. „Spark: cluster computing with working sets“. In: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. Bd. 10. 2010, S. 10.

[Zah+12] Matei Zaharia u. a. „Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters“. In: *Presented as part of the*. 2012.

[Zik+12] P. Zikopoulos u. a. *Harness the Power of Big Data – The IBM Big Data Platform*. Mcgraw-Hill, 2012.

[Apaa] Apache Hadoop project. URL: http://hadoop.apache.org/.

[Apab] Apache HBase. URL: http://hbase.apache.org.

[Apac] Apache Kafka. URL: http://kafka.apache.org.

[Apad] Apache Phoenix - We put the SQL back to NoSQL. URL: http://phoenix.incubator.apache.org.

[Apa15a] Apache Software Foundation. *Apache Avro*. 2015. URL: http://avro.apache.org.

[Apa15b] Apache Software Foundation. *Apache Flink*. 2015. URL: https://flink.apache.org.

[Bas15] Basho Technologies, Inc. *Riak*. 2015. URL: http://basho.com/riak.

[Cou] Couchbase Server. URL: http://www.couchbase.com.

[For] Foreign data wrappers - PostgreSQL wiki. URL: https://wiki.postgresql.org/wiki/Foreign_data_wrappers.

[Goo15] Google Inc. *What is BigQuery?* 2015. URL: https://cloud.google.com/bigquery/what-is-bigquery.

[Hiv] Hive HBase Integration. URL: https://cwiki.apache.org/confluence/display/Hive/HBaseIntegration.

[Ind] Indirection in MUMPS. URL: http://mumps.sourceforge.net/docs.html#INDIRECTION.

[JSOa] JSON. URL: http://www.json.org.

[JSOb] JSON to JSON transformation library. URL: https://github.com/bazaarvoice/jolt.

[Mona] MongoDB. URL: https://www.mongodb.org.

[Monb] MongoDB Manual. *Tailable Cursors*. URL: https://docs.mongodb.com/v3.2/core/tailable-cursors.

[Neo] Neo4J. URL: https://www.neo4j.com.

[Pos] PostgreSQL Documentation. *NOTIFY*. URL: https://www.postgresql.org/docs/9.0/static/sql-notify.html.

[Red] Redis. URL: https://www.redis.io.

[Toa11] Toad for Cloud Databases. *Streamline Access to Complex Data Stores and Manage Them with Ease*. 2011. URL: https://software.dell.com/documents/Toad-for-Cloud-Databases_1.pdf.

[Tra] Transformy.io. URL: https://www.transformy.io.

[Uns] Unstructured Data Query Language. URL: http://unql.sqlite.org.

# Lebenslauf

**Berufstätigkeit**

| | |
|---|---|
| seit<br>04/2017 | Trainer & Consultant<br>EXASOL AG, Nürnberg |
| 09/2012 –<br>03/2017 | Wissenschaftlicher Mitarbeiter<br>Arbeitsgruppe Heterogene Informationssysteme<br>Fachbereich Informatik<br>Technische Universität Kaiserslautern |
| 02/2016 –<br>03/2017 | Dozent für Datenbanksysteme<br>Wilhelm Büchner Hochschule Darmstadt |
| 11/2015 –<br>02/2016 | Lehrbeauftragter für NoSQL-Datenbanken<br>Hochschule Niederrhein |
| 04/2009 –<br>08/2012 | Übungsgruppenleiter für Datenbank-Vorlesungen<br>Technische Universität Kaiserslautern |
| 03/2010 –<br>03/2011 | Wissenschaftliche Hilfskraft (Softwareentwicklung)<br>Center for Mathematical and Computational Modelling<br>Technische Universität Kaiserslautern |
| 11/2008 –<br>12/2010 | Wissenschaftliche Hilfskraft (Softwareentwicklung)<br>Universitätsbibliothek Kaiserslautern |

**Studium und Schulbildung**

| | |
|---|---|
| 10/2010 –<br>08/2012 | Studium der Informatik<br>Master of Science<br>Technische Universität Kaiserslautern |
| 04/2007 –<br>09/2010 | Studium der Informatik<br>Bachelor of Science<br>Technische Universität Kaiserslautern |
| 08/1998 –<br>03/2007 | Thomas-Morus-Gymnasium Daun<br>Abitur |
| 08/1994 –<br>07/1998 | Grundschule Mehren |