Transformations on Graph Databases for Polyglot Persistence with NotaQL

Johannes Schildgen[1], Yannick Krück[2], Stefan Deßloch[3]

Abstract: Polyglot-persistence applications use a combination of many different data stores. Often, one of them is a graph database to model relationships between data items. The data-transformation language NotaQL can be used to define transformations from one NoSQL database to a different one. In this paper, we present a language extension for NotaQL to allow graph transformations, graph analysis, and data migrations on graph databases. NotaQL is schema-flexible, it offers filters and aggregation functions, and it allows for graph traversal and edge creation. Our graph-transformation platform can be used for iterative graph algorithms and bulk processing.

Keywords: Data Transformation Language, Graph Databases, NoSQL

1   Motivation: Graph Databases

NoSQL databases are typically classified in four categories: key-value stores, wide-column stores, document databases, and graph databases. Every NoSQL database has its own data model, a different support for transactions and distribution, and specific benefits and drawbacks. In many enterprises, different NoSQL databases are used in combination to make the best of all. This approach is called *polyglot persistence* [SF12]. As an example, an in-memory key-value store manages frequently-updated page-visit counters, a document database stores user data, and a graph database keeps track of friendship relationships between the users. Storing everything in a graph database would be possible but slow because in distributed environments graph databases typically don't scale as well as other systems and they have higher access costs. Therefore, our document store manages the user data items, and the graph database stores relationships between them. Data-analytic tasks are executed in specific time intervals to extract information from the graph—e.g., the number of friends for each person—and store it into the document database. Alternatively, data from the document store can be analyzed to introduce new edges in the graph database, e.g. to connect people who frequently communicate with each other.

To simplify the development and support the efficient execution of polyglot-persistence applications [Ge14], not just APIs and frameworks are needed, but also languages and platforms that can transform and move data from one data store into another. At the heart of our vision for this data-store interoperability is a language for data transformations that should support a wide range of database systems with all their specific data-model

---

[1] Technische Universität Kaiserslautern, schildgen@cs.uni-kl.de
[2] Technische Universität Kaiserslautern, y_krueck11@cs.uni-kl.de
[3] Technische Universität Kaiserslautern, dessloch@cs.uni-kl.de

concepts. Different from many classical approaches, this language should not unify different data models and cover only their commonalities, but be extensible and allow to introduce individual language constructs to fully support all of their data model specifics. Otherwise, the rapidly evolving landscape of NoSQL databases is difficult to support. Our language NotaQL [SD15, SLD16] addresses the above requirements for data transformations between different NoSQL stores. The language is concise, easy to learn, schema-flexible, and data-model independent. The latter fact is realized by allowing specific language constructs for every data model and by using an internal data structure that is a superset of all other supported models. In our previous work, we described how so-called *aggregate-oriented* NoSQL stores (i.e., key-value stores, wide-column stores, document databases) are supported as sources and targets for NotaQL transformations.

In this paper, we significantly extend NotaQL to support graph databases. The main challenges we have to address stem from data-model differences. Aggregate-oriented stores focus on storing independent items that are typically accessed by an ID and searched or transformed one after the other. In contrast, graph databases connect items, provide graph-traversal languages that include powerful access methods, and support graph APIs for easy access of property values and related items. Item relationships are natively supported by graph databases and optimized for fast navigation.

The data model of a graph database is a graph in the mathematical sense. $G = (V, E)$ defines the graph with its vertices and edges. Vertices in a so-called *property graph* are semi-structured—like JSON documents in a document store. A vertex can have a list of property-value pairs, and optional labels. An edge connects two vertices. It has a label, a direction and also a list of properties. Figure 1 shows a simple property graph with two vertices connected by one edge.
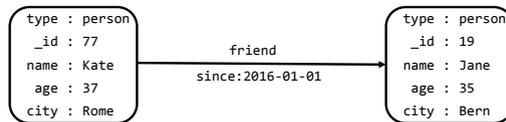
Fig. 1: A simple property graph G_1.

Due to their emphasis on networks of data items, graph databases typically don't scale as well as aggregate-oriented stores. However, they are very important for managing and analyzing complex connected data, which is often found in social networks or recommender systems. Typical operations here are traversing the graph by navigating from one vertex to another and iteratively computing and refining node or edge properties. As a consequence, graph-database languages and platforms differ a lot from classical query languages and computation frameworks. (Please see Section 6 for a more thorough discussion.)

In polyglot-persistence applications that utilize a combination of different systems, frameworks and languages are needed that can handle both aggregate-oriented data models and graphs. Here, one main challenge—in addition to the performance—is how a user can implement algorithms in such a language or framework correctly and efficiently [Ho12]. Lumsdaine et al. [Lu07] state four software design issues, namely flexibility, extensibility, portability, and maintainability. There are many approaches that unify the access to different

stores [Ge14, SGR15, OPV14], but they do not fully support all data-model concepts. Other systems like ArangoDB [Ar16] use a combination of different data models and introduce a query language that supports all of them. In this paper, we present a platform for graph transformations with the language NotaQL. Our platform supports all data-model concepts without introducing a new database system. The NotaQL platform connects to arbitrary NoSQL databases and executes user-defined transformation scripts to perform migrations from one system into another. In graph databases, NotaQL can easily access and modify properties as well as traverse and create edges. Other use cases for NotaQL are data migration and integration tasks. If some database stores relationships as arrays, sub-documents, or foreign keys, and another system uses a graph database, NotaQL can be used to convert the data from the first schema to the second one, and vice versa.

The following list shows the main contributions of our paper:

- We present an easy-to-learn, concise and powerful language for data transformations on property graphs (based on NotaQL),
- a syntax to access and create properties and edges, and to traverse to neighbors,
- a solution for defining iterative graph algorithms,
- a transformations platform to execute NotaQL scripts on different graph database systems,
- an approach for cross-system transformations between graph databases and other kinds of databases or file formats.

In the next section, we provide a brief overview of NotaQL based on previous work. We then present our new extension for graph databases in Section 3. In Section 4, we provide details on implementing graph database support in our transformation platform. After reporting on an initial performance validation in Section 5, we present related work on existing graph languages and frameworks in Section 6 and conclude the paper in Section 7.

2   The Data-Transformation Language NotaQL

NotaQL is a language to transform a set of input items into a set of output items. These sets can be tables in a wide-column store, collections in a document database, or a map in a key-value store. In these cases, the items are rows, documents, or key-value pairs.

The main part of a NotaQL script are a *filter specification* that defines which items to transform, and *attribute mappings*. A NotaQL script is output oriented. This means that the attribute mappings define how the output items should look like, i.e. which attributes they have and how the values of these attributes are computed. Figure 2 shows how a NotaQL script is logically executed. First, the items are filtered by the predicate given in the filter specification in the NotaQL script. Afterwards, intermediate items are created by mapping each item as defined in the attribute mappings. The mapped items are then decomposed into output fragments, and fragments which belong to the same output *cell* are combined with reference to the aggregation functions used in the transformation script.

In a wide-column store, the input items are the rows of a table. The ones that fulfill the filter are split into cells. A cell $z = (\_r, \_c, \_v)$ is defined by the row-id $\_r$ of the row where it
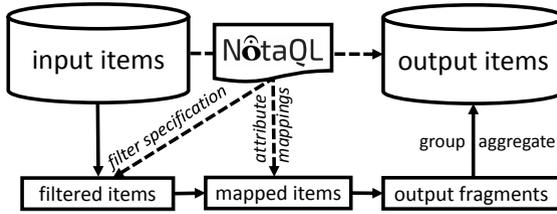
Fig. 2: Logical Execution of a NotaQL Script

originates, a column name $\_c$, and a value $\_v$. The attribute mappings define how to create output cells based on the input cells. As a first example, the following script counts all people older than 17 in each city:

```
IN-FILTER: IN.age > 17,
OUT._r <- IN.city,
OUT.numPeople <- COUNT()
```

The NotaQL script uses a filter to process only rows that have a column called `age` and a value greater than 17 in this column. The filtered rows are split into their cells, and here, only the cell with the column name `city` is of interest. Each city cell is used to produce an intermediate cell $z' = (\_r', \_c', \_v')$ where $\_r'$ is the value of the city cell in the input, $\_c'$ is the string literal `numPeople`, and $\_v'$ in each output fragment a 1 because the aggregation function `COUNT()` is defined as `SUM(1)`. All fragments that belong to the same row (i.e., the same city) are grouped together and their values are summed up to the final value.

The NotaQL platform presented in [SLD16] extends the possibilities of NotaQL by not only supporting wide-column stores, but any kind of aggregate-oriented database. The platform reads from one database system, filters, transforms and aggregates data, and writes its result to another system. The output system can be a different kind of NoSQL database, or it can be the same data store. In the latter case, the results are written into a different collection or table, or it can be the same one as the input to modify data in place. A cross-system NotaQL script starts with the definition of an *input* and an *output engine*. Each engine has a name and specific parameters. As an example, the CSV engine takes the path to a CSV file in a local or distributed file system. After the engine definition, an optional *input-filter* clause can be used to make a selection. The rest of the NotaQL script are *attribute mappings*. They define output attributes and how their values are computed based on the input data. Usually, the first attribute mapping defines the value of an object ID. In key-value stores, the ID is called a *key*; in wide-column stores, it is the *row-id*. They have to be set in a NotaQL script. If an ID value is already present in the database, an in-place update of the existing item is performed, otherwise a new item with the given ID is inserted. Document stores and other system can automatically generate IDs, so there is no need to define an attribute mapping for them in that case. Figure 3 shows the syntax of a cross-system NotaQL script.

notaql:
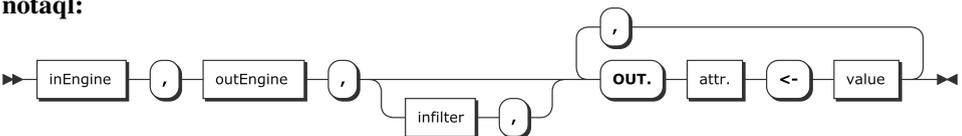
Fig. 3: NotaQL Syntax

The grammar symbols *attr.* and *value* depend on the input and output engine. When—like in the example above—a wide-column store is used as the input, the value can be `IN._r` (row-id), `IN._c` (column name), `IN._v` (column value), or `IN.x` (value of a given column `x`). Analogue for the output attributes. Key-value stores do not have named columns but only keys and values which can be accessed with `IN._k` and `IN._v`. Document stores use `IN._id` for a document ID, dot notation (`IN.x.y`) for accessing sub-attributes, and a function `LIST` to create arrays.

The following example reads all key-value pairs of a Redis database that represent user items and stores them into a MongoDB collection:

```
IN-ENGINE: redis(database <- 0),
OUT-ENGINE: mongodb(database <- 'test', collection <- 'users'),
IN-FILTER: _k LIKE 'user/%',
OUT.username <- IN._k,
OUT.email <- IN._v
```

The access to the key and value using `IN._k` and `IN._v` is specific for key-value databases. For the MongoDB output documents, arbitrary attributes can be set, here `username` and `email`. As no `OUT._id` is set, each document will have an automatically generated document identifier. No-


Fig. 4: Cross-System Execution of a NotaQL Script

taQL is a schema-flexible language. One can access the values of all attributes without knowing their names using `IN.*`. An attribute name can be returned with `IN.*.name()`. Accessing only specific attributes is possible with *attribute filters*: `IN.?(name()!='age')` finds all attributes except the age. On the output side, the *indirection operator* `$` can be used to create new attributes using a derived attribute name, e.g., `OUT.$(IN.*.name())` creates the same attributes in output items as the ones found in the input items. While iterating over a set of attributes using the `*` or `?`, the attribute of the current iteration can be accessed with `IN.@`. This way, all attributes together with their values can be copied using `OUT.$(IN.*.name()) <- IN.@`.

Figure 4 shows the execution of a NotaQL script on our Apache Spark-based [Za10] platform. The internal data model of NotaQL is based on the JSON data model because it is the most powerful data model of the aggregate-oriented stores. All other supported models and also CSV or JSON files can be mapped to this model. But it comes to its limits when we want to support graph databases. That is why we present a new approach in this paper that extends NotaQL by supporting relationships between items, and performing iterative transformations.

3   NotaQL for Graph Databases

A graph database can be seen as a document store with documents being the vertices and connections between documents being the edges. If we leave out the edges, the data model

is the same as for document databases. So, we can reuse the existing NotaQL language for vertices and properties.

3.1   Item-to-Item (Vertex-to-Vertex) Transformation

We first focus solely on how vertex information can be access and mapped. So edge connections are ignored here and will be covered later in Section 3.2. A NotaQL script defines how to create an output item based on the input. Given a property graph without edges $G_1 = (V, \emptyset)$ stored in a graph database, NotaQL connects to the input graph G_1 and writes its output to an initially empty graph G_2. As described in Section 2, an IN-FILTER clause can be used to filter input items by a given predicate. All vertices that fulfill the predicate are transformed with respect to *attribute mappings*. Every attribute mapping has the form OUT.p <- value, where p is the name of a property of the output vertices and value an arbitrary definition or computation of a numeric, string or differently typed value. Here, properties of input vertices can be accessed, literals can be used, and functions can be called.

The following NotaQL script transforms the graph G_1 (the one shown in Figure 1) that is stored in a Neo4J [Ne16a] database into G_2 by performing some selections and projections:

```
IN-ENGINE: neo4j(path <- '/data/G1'),
OUT-ENGINE: neo4j(path <- '/data/G2'),
IN-FILTER: type='person' && age > 0,
OUT.name <- IN.name,
OUT.year_of_birth <- 2016 - IN.age,
OUT.type <- 'person'
```

type : person	type : person
_id : ...	_id : ...
name : Kate	name : Jane
year-of : 1979 -birth	year-of : 1981 -birth
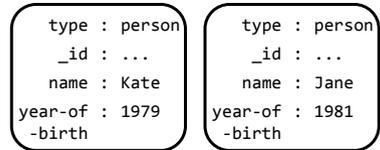
Fig. 5: Selection and Projection of Vertices; Graph G_2

The execution of this transformation works as follows: For every vertex having the label 'person' and an age greater than zero, an output vertex is created with the same name and a new property year_of_birth. All other properties are discarded. As this transformation writes its output to an initially empty graph, the concept of a vertex ID is not needed here. In Neo4J, vertices do have identifiers, so in this case, they are automatically generated. When a NotaQL transformation is used to update an existing graph in place, an ID has to be set to specify whether to change an existing vertex (the one with the given ID) or to insert a new vertex (if no ID matches).

In the following example transformation, the age value in every person vertex with an age greater than zero is incremented by one.

```
IN-ENGINE: neo4j(path <- '/data/G1'),
OUT-ENGINE: neo4j(path <- '/data/G1'),
IN-FILTER: type='person' && age > 0,
OUT._id <- IN._id,
OUT.age <- IN.age+1
```

type : person		type : person
_id : 77		_id : 19
name : Kate	...	name : Jane
age : 38		age : 36
city : Rome		city : Bern
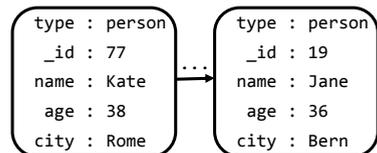
Fig. 6: In-Place Updates on a Graph

The OUT._id <- IN._id mapping indicates that the current vertex should be updated. In case the script might produce multiple output vertices with the same ID, the property

mappings have to either guarantee the equality of all values within each ID group, or they have to contain aggregate functions. The aggregation function SUM, COUNT, MIN, MAX, and AVG are used to generate atomic output values for every property. All property values of output vertices that have the same ID are reduced to a final value using the given function.

The following transformation produces a new graph G_3 that consists of city vertices having the average age of all people living in this city as a property:

```
IN-ENGINE: neo4j(path <- '/data/G1'),
OUT-ENGINE: neo4j(path <- '/data/G3'),
IN-FILTER: type='person' && age > 0,
OUT._id <- IN.city,
OUT.city <- IN.city
OUT.avg_age <- AVG(IN.age),
OUT.type <- 'city'
```

type : city		type : city
_id : Rome		_id : Bern
city : Rome		city : Bern
avg_age : 37		avg_age : 35
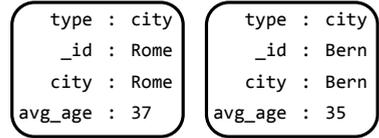
Fig. 7: Average Age per City; Graph G_3

Because of the trivial functional dependency IN.city → IN.city and the given property mapping, the functional dependency OUT._id → OUT.city holds. This is why the city property values are equal within each ID group. For the property IN.age, the aggregation function AVG is used to calculate an atomic output value for the output property OUT.avg_age.

3.2   Traversing Edges in the Input Graph

As shown above, IN. is used to access property values of input vertices. We added the following three steps to access its edges: IN._e traverses all edges, IN._>e all outgoing, and IN._<e all incoming ones. With ?-predicates, which were already used in non-graph NotaQL (see Section 2), the kinds of edges can be further specified; based on their labels and properties. After an edge step, one can access the edge properties with the simple dot notation, e.g. IN._e.since, and one can follow the edge to its neighbor vertex using one more _ symbol. For selecting only specific neighbors, additional ?-predicates can be used. The neighbor vertex's properties and edges can be accessed as usual. E.g., IN._e_.name returns the neighbor's name. When iterating over multiple edges, the edge within the current iteration can be accessed using IN._e[@]. To avoid long lists of edge steps, NotaQL uses the [min,max] option to follow an edge over multiple hops. The default edge traversal is exactly one hop: [1,1]. For the transitive closure up to an unbounded number of hops, [0,*] is used. In that case, termination in cyclic graphs has to be achieved by using proper predicates. Figure 8 shows the grammar for edge traversals in NotaQL.

Aggregation functions in NotaQL take a list of values and compute one output value. In the examples in Section 3.1, the lists where created due to the cell groupings. All values that belong to the same output cell—i.e., in a graph database, the same property for the same output vertex—, are collected in a list, and an aggregation function combines the values in this list. NotaQL also allows calling aggregation functions on lists that are not created after grouping but already existed in the input data. These lists are automatically created when using ambiguous expressions, e.g. IN.a[*] contains all elements of an array, IN._e_._id are the IDs of all neighbor vertices.

edge:

**vertexAccess:**

minmax:
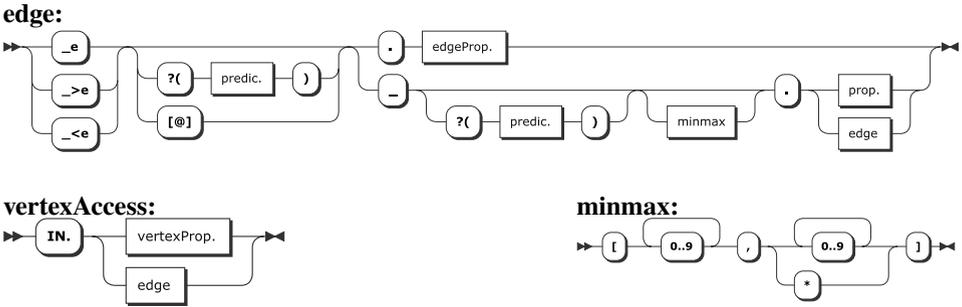
Fig. 8: Syntax for Traversing Edges and Access to Properties

The following NotaQL script shows an example with some edge accesses and aggregations.

```
IN-ENGINE: neo4j(path <- '/data/G1'),
OUT-ENGINE: neo4j(path <- '/data/G1'),
IN-FILTER: type='person',
OUT._id <- IN._id,
OUT.num_neighbors <- COUNT(IN._e_._id),
OUT.num_old_friends <- COUNT(IN._e?('friend')_?(age>=80)._id),
OUT.num_friends_of_friends <- COUNT(IN._e?('friend')_[1,2]._id),
OUT.oldest_friendship_date <- MIN(IN._e?('friend').since),
OUT.mother_name <- IN._>e?('mother')_.name
```

The edge predicate `?('friend')` is a short form for `?(_l = 'friend')`, where `_l` designates the edge label. Most edge traversals in this example use the `_e` path to traverse an edge independently of its direction. However, to navigate to the mother's vertex, the `_>e` path selects only outgoing edges. The aggregation functions used in this example are `COUNT` and `MIN`. They are called on lists of property values of neighbor vertices, respectively on edge properties. When a NotaQL transformation groups multiple vertices, it is possible that values are lists before grouping. This results in lists of lists after grouping. These are resolved by using two aggregation functions in combination. The following example shows how to compute the average number of friends people have per city:

```
IN-ENGINE: neo4j(path <- '/data/G1'),
OUT-ENGINE: neo4j(path <- '/data/G3'),
OUT._id <- IN.city,
OUT.avg_num_friends <- AVG(COUNT(IN._e?('friend')_._id))
```

The expression `IN._e?('friend')_._id` is a list of friend IDs. After grouping by city, every city vertex contains a list of lists of these IDs. The cardinalities of the inner lists are computed with the `COUNT` function. The `AVG` function computes the average of these count values.

3.3 Creating Edges

For traversing edges or accessing their properties, we handle edges almost like vertex properties, as seen above. However, creating an edge is not as trivial as setting a vertex

property. This is because an edge has a target vertex, a label, a direction, and a list of properties. We decided to define the target and direction of an edge on the left-hand side of the mapping arrow `<-`, and the edge data as parameters of an `EDGE` constructor on the right-hand side. This approach is consistent with NotaQL's support for other data models and data types, where the construction of complex objects or list values is specified in a similar way. The target of an edge is determined by a predicate that matches the target vertex. If multiple vertices match the predicate, then multiple edges are created. Figure 9 shows the syntax for edge creation.

outEdge:
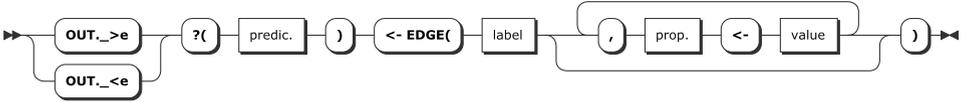
Fig. 9: Syntax for Edge Creation

From the current vertex's view, `OUT._>e` creates an outgoing and `OUT._<e` an incoming edge to respectively from the vertices that fulfill the given predicate. The `EDGE` constructor function requires a label and a (possibly empty) list of edge properties.

In the following example, two edges are created for every vertex: The first one says that everybody is a friend of Sam, and the second one is an edge to each person's grandmother[4]:

```
OUT._id <- IN._id,
OUT._>e?(name = 'Sam') <- EDGE('friend'),
OUT._>e?(_id = IN._>e?('mother' || 'father')_._>e?('mother')_._id)
    <- EDGE('grandmother', via <- IN._>e[@]._l)
```

Within the latter ?-predicate, the ID of the grandmother vertices are searched, and edges are created to the vertices having this ID. The edge is an outgoing one; it has the label `grandmother` and one property `via`. The term `IN._e[@]._l` navigates to the current edge that is bound in the edge-target predicate and reads its label. So the value of the `via` property is either `'mother'` or `'father'`. If a vertex does not have a mother or father edge, or if the neighbor does not have a mother edge, the predicate is evaluated to false for every vertex. In this case, no grandmother edge is created. Multiple paths to the same vertex do not result in the creation of multiple edges. But if a person has multiple mother or father edges to different vertices, the equality predicate checks for list containment so that multiple edges would be created. Same if more than one person is named Sam. Then friendship edges are created to all of them.

3.4 Iterative Computations

Many graph algorithms that change the graph in place have to be executed multiple times to produce the final result. In NotaQL, we therefore introduce the `REPEAT: n` clause, which can be used to run a transformation `n` times. The repeat iteration also stops if the graph has not changed since the previous iteration. For `REPEAT: -1`, the computation runs as long as

---

[4] In this and the following examples, we omit the `IN-ENGINE` and `OUT-ENGINE` clause when the output graph is the same as the input graph, i.e., for in-place updates. We also omit the `IN-FILTER`, which should be used to only apply the transformation on vertices with a specific label.

the graph changes. Another possibility is to monitor the change of a single given vertex property (`property(p%)`), and to stop as soon as the change of this property value is below a certain percentage `p` for all vertices. Figure 10 shows the syntax of a REPEAT clause.

repeat:
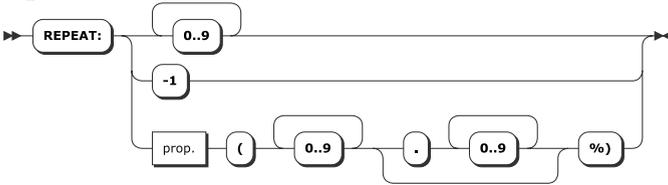
Fig. 10: Repeat Clause

The following example shows the PageRank [Pa99] algorithm in NotaQL. It uses a global function `NumVertices()`, which returns the number of vertices in the graph.

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

REPEAT: pagerank(0.0005%),          # main iteration
OUT._id <- IN._>e._id,
OUT.pagerank <- SUM(IN.pagerank/count(IN._>e._id))
```

$$PR(p) = \frac{1}{N}$$

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

Fig. 11: PageRank

There are two transformations. After having initialized the graph with pagerank = $1/N$ for all vertices—with N being the number of vertices—the iterative part of this NotaQL script is then executed until all PageRank values change less than 0.0005% within one iteration. For a given input vertex p, the PageRank values of all neighbors $q \in out(p)$ via the outgoing edges are influenced. The final value of q's PageRank is the sum of all PageRank values from vertices like p divided by their out-degree. It can be seen in Figure 11 that the NotaQL definition is very similar to the original PageRank formula. Improvements like a damping factor [Pa99] can easily be added into the NotaQL script.

3.5 Modeling Relationships in Non-Graph Databases

Not all applications use graph databases to store relationships between data items. We want to present three popular schema approaches that are often found in relational and NoSQL databases to model graphs using data-model concepts like tables, lists or nested objects. During this discussion we show (non-graph) NotaQL scripts that work with these schema approaches and transfer one representation into another. Later, we present cross-system transformations between a graph database and a non-graph database that uses one of the three presented schemes.

Vertex and Edge Table This schema is often found in relational databases to model graphs. A vertex table has a primary-key column for a vertex ID, a type column for the label, and one column for each property. The edge table consists of two foreign-key columns for the source and target vertex IDs, also one column for the label, and one for each property. For querying and transforming the data, the two tables need to be accessed multiple times

and joined to traverse the graph, once for every hop. As NoSQL databases try to avoid joins, this approach is mostly used in relational databases. Furthermore, the properties have to be defined in advance, and every vertex can only have one label. This can be solved by introducing separate label and property tables [Su15]. Tables 1 and 2 show the schema of the graph shown in Figure 1.

id	type	name	age	city
77	person	Kate	37	Rome
19	person	Jane	35	Bern

Tab. 1: Vertex Table

source	target	label	since
77	19	friend	2016-01-01

Tab. 2: Edge Table

Adjacency Lists   Adjacency lists and adjacency matrices are the most common ways to represent graphs for mathematical computations. While adjacency matrices are rarely used in databases or files, the lists are often found to store graphs, e.g., in CSV files. Every line in this file represents a vertex. The first value in one row contains a vertex ID, the other values are IDs of neighbors via outgoing edges:

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

The same graph can be represented in a wide-column store (see Table 3). Here, multiple column families can be used to both store vertex properties and neighbors in one table [Ch08]. Instead of leaving the column values empty, they can also hold edge properties.

row-id	friends				info		
77	19:-	28:-	39:-	21:-	name:Kate	age:37	city:Rome
19	40:-	28:-			name:Jane	age:35	city:Bern

Tab. 3: Adjacency Lists in a Wide-Column Store

Adjacency lists are also often found in document databases in the form of arrays. Again, vertex properties and outgoing edges can be stored within one single JSON document. We want to show a NotaQL transformation to convert a vertex and edge table into this schema. We do this by taking the edge table (see Table 2) as the input, construct lists of neighbor-vertex ids, and write these lists in the original vertex table (see Table 1).

```
OUT._id <- IN.source,
OUT.$(IN.label) <- LIST(IN.target)
```

The indirection operator $ is used to produce individual lists for every edge label. The result documents look like this:

```
{ _id: 77, name: "Kate", age: 37, city: "Rome",
  friend: [ 19, 28, 39, 21 ] }
```

Nested Objects Forming Tree Structures   If the data items form tree structures, relationships between items can be modeled by nesting them. In a document store, we use an array of sub-documents to model edges to these sub-documents. The following example shows a blog post with comments that can again be commented:

```
{ _id: 1420733, user:  "Kate", text: "I'm in Berlin now",
  comments: [{ user: "Carl", text: "Have fun!" },
             { user: "Jim", text: "I'm also in Berlin!",
               comments: [{ user: "Kate", text: "Let's meet!" }]}]}
```

All these variants are often found in databases. But for complex graphs, they are not applicable. This is because writing programs and queries over such schemata is hard, analysis and transformations are slow, and joins are often not supported. The solution are graph databases which offer special query languages and a better performance.

3.6 Cross-System Graph Transformations

In polyglot-persistence environments, it is often necessary to transfer or copy data from one data store to another. As graph databases exhibit worse horizontal scalability than for example document stores, it is a popular approach to use a graph database only for storing relationships between items, while the item's properties are stored in a document store. The grammar in Figure 12 is a modified version of Figure 3 from the beginning of this paper. With the graph-database extension of NotaQL, it is possible to mix the usage of graph and other databases in the `IN-ENGINE` and `OUT-ENGINE` definition. For each engine, data–model-specific language constructs can be used, e.g. `IN._k` and `IN._v` for a key-value store or `IN._e` for a graph database as input. All this is part of the definition of the `value` symbol in Figure 12.

notaql:
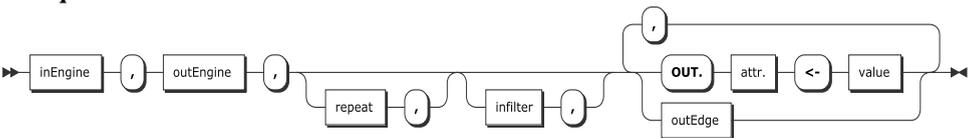
Fig. 12: Syntax for Cross-System NotaQL Transformations with Edge-Creation Option

In the next example, we want to show a typical data transformation, namely a data-migration task from MongoDB to Neo4J. As MongoDB does not support direct relationships between documents, a one-to-n relationship for a person's status updates can be modeled as a list of nested sub-documents, and an n-to-m relationship for friendships can be modeled as an adjacency list of the friends' IDs. Figure 13 shows one input document and the desired output of the documents-to-graph transformation.

Using the following script, the data is transformed into a graph with person and status vertices plus the connecting edges.

```
#1. create status vertices
IN-ENGINE: mongodb(database<-'socialnet', collection<-'people'),
OUT-ENGINE: neo4j(path <- '/data/socialnet'),
OUT._id <- IN.status[*].sid, OUT.ts <- IN.status[@].ts,
OUT.text <- IN.status[@].text, OUT.type <- 'status';
```

```
{ _id: 77, name: "Kate", age: 37,
 status: [
  { sid:714, ts:1448269379,
    text:"I'm in Berlin now!!!"},
  { sid:788, ts:1448372647, text:
    "The currywurst museum is amazing!"}
 ], friends: [ 19, 78, 81, 103 ]
}
```


Fig. 13: Left: Input as JSON Documents; Right: Output as a Graph

```
#2. create person vertices and edges to their statuses and friends
IN-ENGINE: mongodb(database<-'socialnet', collection<-'people'),
OUT-ENGINE: neo4j(path <- '/data/socialnet'),
OUT._id <- IN._id, OUT.name <- IN.name, OUT.age <- IN.age,
OUT._>e?(type='status' && _id=IN.status[*].sid) <- EDGE('posts')
OUT._>e?(type='person' && _id=IN.friends[*]) <- EDGE('friend'),
OUT.type <- 'person';
```

There are two transformations. The first one creates status vertices. A new vertex is created for every element in each person's status list. With `[*]`, we iterate over the list. While iterating, the current element can be accessed with `[@]`. For the example document in Figure 13, two status vertices are created. In the second transformation, the person vertex is created. There, an edge links to every status vertex and other edges to the friends. As we explain in the next section, edges are created at the very end of a transformation. So, it is guaranteed in this case, that the target vertices of friendship relationships exist—assumed that there are no dangling references in the input database. Usually, the list of friends in the input is symmetric, so Kate's friend Jane will have Kate's ID in its friends list. To avoid the creation of edges in both directions, one can simply add `&& _id>IN._id`. Then, the friendship edge points from the vertex with the smaller ID to the one with the larger one. The typical way to model symmetric relationships in graph databases is creating an edge in an arbitrary direction. At query time, an edge is traversed independently of its direction.

A transformation in the opposite direction, i.e. from a graph database to a different database, is possible with the edge-accessing steps shown in Section 3.2. To reverse the transformation in Figure 13, the following NotaQL script can be used:

```
IN-ENGINE: neo4j(path <- '/data/socialnet'),
OUT-ENGINE: mongodb(database<-'socialnet', collection<-'people'),
IN-FILTER: type='person',
OUT._id <- IN._id,
OUT.name <- IN.name, OUT.age <- IN.age,
OUT.status <- LIST(OBJECT(sid <- IN._>e?('posts')_._id,
 ts <- IN._e[@]_.ts, text <- IN._e[@]_.text)),
OUT.friends <- LIST(IN._e?('friend')_._id)
```

The transformation is performed for every person vertex. It creates person documents in which the `status` attribute is a list of objects, filled with property values of neighbor vertices using the outgoing `posts` edges. For friends, only the vertex IDs are retrieved to create a list of foreign keys. In the given example, we used the constructor functions `LIST` and `OBJECT`, which are specific for document stores as an output.

As there are also input and output engines for CSV and JSON files, NotaQL can be used to easily import and export graphs.

4   Realization

While our implementation in [SLD16] is based on Apache Spark [Za10], we developed our graph transformation prototype as a simple Java tool. This is because we found no good framework that supports the different data models of graphs and simple datasets properly. The other reason is that graph databases are usually accessed with their own API and not with frameworks because they are typically not stored in a distributed system with multiple machines. We decided not to use the direct API of a graph database but to use the graph-database gateway *Tinkerpop Blueprints* [Ap16] (see Figure 14). This gateway gives a unified access to different graph database systems like Neo4J, Titan [Ti16], Tinkergraph [Ap16] and many others. This way, we support a magnitude of various stores at the same time. One drawback is that Blueprints does not support vertex labels. As seen in the previous examples, we used a `type` property for storing it.

A graph-to-graph transformation is done in five steps:

1. Parsing
2. Vertex Access and Pre-Filter
3. Input Filter
4. Creation of Vertices
5. Creation of Edges
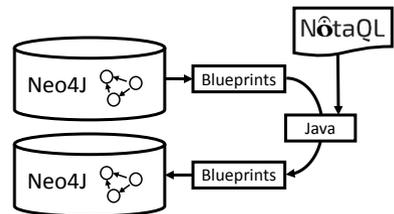
Fig. 14: NotaQL-Script Execution on Graphs

In iterative algorithms, the steps (2) to (4) are repeated until the stop condition is fulfilled. For parsing in step (1), we use the ANTLR [Pa16] parser and lexer. Step (2) uses the Blueprints method `getVertices(String key, Object value)` to find a superset of all vertices of interest. As this method only checks for equality on one single property, complex input-filter predicates have to be converted into the conjunctive normal form (CNF) first. After that, one clause that only consists of one equality literal, can be used for pre-filtering with the `getVertices` method. If there is no such single-equality-literal clause, or if there is no input filter at all, all vertices are received. But in fact, in typical NotaQL transformations there is such a equality literal, namely a type predicate (e.g., `type = 'person'`).

In step (3), our algorithm iterates over all found vertices from step (2) and filters out the ones that violate the input filter. If Blueprints adds a support for complex predicates in the future, or if we change the implementation so that it uses the native graph-database API, step (3) can be dropped.

Within the iteration over the input vertices, new vertices are created and the attribute mappings in the NotaQL script are performed (step (4)). Simple property mappings like `OUT.a <- IN.b` or `OUT.a <- 5` can be easily evaluated to create an output vertex's property values. Traversals over edges in the input graph are converted into method calls of the Blueprints API to navigate to neighbors. Aggregation functions that are applied on lists in the input can also be easily evaluated by retrieving all values from neighbors and computing the aggregated result, e.g. a sum. Aggregation functions after grouping are more complex to evaluate as they combine data from multiple input vertices. One solution would be grouping all output vertices by their ID and afterwards applying a reduce function to produce one single vertex for every group. This behavior is similar to MapReduce [DG04] or Spark [Za10]. But these frameworks split the data into chunks and distribute them over a cluster of machines. In our case, all data has to be kept in memory of our computing node. Our solution is simple and effective: The aggregation functions are ignored first. When the output vertex is written into the target graph database, this write is done in an incremental fashion with reference to the given aggregation function. For example, if the function is `MIN`, the property value is overwritten only if it is smaller than the current one. If it is `SUM`, the current property value is incremented by the new one.

Step (5) starts as soon as all vertex creations are completed. In that step, our algorithm iterates again over all input vertices that fulfill the input filter. Then it evaluates the `OUT._id` definition and all edge creation parts. With this information, it can produce all edges between vertices in the output graph.

For cross-system transformations, we combine our existing Spark-based platform [SLD16] and the graph-to-graph transformation platform presented in this paper. This way, we combine the benefits of both worlds: The first system is optimized for aggregated stores and files and supports distributed storage and computation. The second one works directly on graphs so that connections between data items are supported as native data-model concepts. Both platforms communicate with each other using a common in-memory graph database Tinkergraph [Ap16] as an intermediate format. A transformation from a graph to a non-graph system uses the given NotaQL
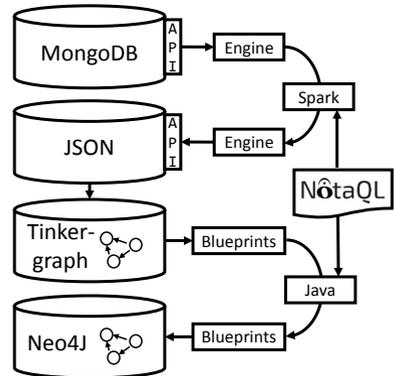

Fig. 15: Cross-System Execution of NotaQL between a Graph and a different NoSQL Database.

script to produce a Tinkergraph without edges. After that, a generic Tinkergraph-to-JSON tool is executed to produce a JSON file that can be read by the Spark-based platform to write the output in the target database. A transformation in the opposite direction, i.e., from a non-graph to a graph database, produces the intermediate JSON file first. This one is converted into a Tinkergraph, and the Tinkergraph is the input for a graph-to-graph transformation that produces the output. Both the intermediate JSON file and Tinkergraph do not contain edges. We perform query rewriting to move the edge access and creation

part of a NotaQL script to the graph platform. The result is a fast execution that involves different kinds of databases and data models.

Figure 15 shows the execution of a transformation from a MongoDB collection to the graph database Neo4J. The NotaQL script is rewritten so that it writes the (filtered) MongoDB documents into a JSON file. After that, the generic JSON-to-Tinkergraph tool is used to create a Tinkergraph. This Tinkergraph acts as an input for the NotaQL graph-transformation platform that writes the result into Neo4J using the rewritten NotaQL script.

5   Initial Validation

We ran some initial experiments to validate our general approach in terms of feasibility and performance, For that, we used a single machine with a double-core Intel i3 M 370, 2.4 GHz and 3 GB of memory. As a test database, we used a subset of the data of the Slovenian social network Pokec [TZ12] in JSON format. Our test dataset consists of 15, 000 vertices and 200, 000 edges. Different from our examples above, friendships are asymmetric, so if A has a friend B, this does not imply that B also has a friend A. The data from [TZ12] consists of one collection of person documents, and one containing the relationships between them. This corresponds to the vertex/edge-table schema described in Section 3.5. In a first step, we combined both collections into person documents with adjacency lists. These documents look like this:

```
{ "_id":"14943", "user-id":"14943", "username":"14943",
  "avatar":"185 cm, 65 kg, ????asi:d",
  "follower": ["8303"], "following": ["8934","544"] }
```

In [Pa15], a complex Java program is presented to import such a graph that is stored in a JSON file into Neo4J. The program iterates over all JSON documents and creates Cypher statements to create vertices for every person and statements to add the edges between them. The list of statements is written into a file, which is then executed on Neo4J. We tested the proposed program on our Pokec database. It took 207 minutes to import the data into Neo4J. With NotaQL, the transformation can be expressed in only four short lines of code and the import needed only 95 seconds. In applications where we have much larger data sets, the solution that generates Cypher statements is not suitable because it would need many days for the import.

In a second experiment, we imported the Pokec dataset into a MongoDB database. We used the MongoDB aggregation pipeline to compute the number of friends plus the number of friends of friends for every person. The algorithm starts with unwinding the `following` array, grouping by the ID, and counting the direct neighbors. Then, the number of indirect neighbors are added to this number by performing a `$lookup` operation and another grouping and counting. This computation took 23 minutes. Next, we formulated the query as an NotaQL script. As neither the input nor the output is a graph database, we cannot apply the NotaQL graph-transformation platform here directly. So, we first load the MongoDB collections into an in-memory Tinkergraph, and apply the friends-of-friends computation

on this, using the MongoDB output engine. All in all, this job took only 40 seconds, so we have a speed-up of more than 3,000%.

6   Related Work: Graph Languages and Frameworks

The three most common ways to work with a graph database are (1) working with the database-specific API to find, create, and modify vertices and edges of a graph, (2) using a graph query language, and (3) utilizing a graph-processing framework. The API methods provide efficient access to individual vertices and support a simple graph traversal. Application developers can directly use these methods to work with graph databases. However, for complex tasks like data analytics or iterative computations, much code needs to be written. A NotaQL script can be written in one minute and executes the same complex graph transformation. As an alternative to APIs, graph query languages like Gremlin and Cypher can be used. *Gremlin* [Ro15] belongs to the *Tinkerpop* [Ap16] stack and uses the common *Blueprints* API for graph databases. Thus, Gremlin is widely supported. A query consists of multiple steps such as filters to select vertices and edges based on their labels or properties, traversal steps to move to neighbor edges or vertices, or aggregation steps. Side-effect steps can be used to store intermediate values in variables so that they can be accessed in later steps. Gremlin is easy-to-use and powerful for reading, but not for graph transformations. For vertex and edge creation, there are only simple methods that are typically only used to modify one single vertex or edge. NotaQL also uses the Blueprints API and thus, it supports all graph databases that are supported by Gremlin. However, NotaQL is used for the tasks Gremlin cannot do, namely complex graph transformations. Another language is *Cypher*, the graph query language for *Neo4J* [Ne16a]. Its syntax is similar to SQL and consists of pattern-matching elements like in *SPARQL* [Pr08]. A `MATCH` clause is used to define a pattern that is searched for in the whole graph. In this pattern, variable names can be introduced. For every match, the rest of the query is executed. This can be a `WHERE` clause for filtering, a `RETURN` clause to return a result for each match, or a writing clause like `UPDATE`, `CREATE` or `DELETE` to modify the graph in place. While this allows for set-oriented graph transformations, Cypher has restrictions with respect to the complexity of these transformations. In writing clauses, no aggregation functions can be used. Furthermore, Cypher does not support iterative algorithms, it cannot properly handle flexible schemata or transform metadata into data and vice versa. NotaQL supports all these features and it allows for cross-system transformations. Cypher queries can only be based on one single graph, and they can only change the graph in place.

There are many graph-processing frameworks that are optimized for distributed graph processing. *Pregel* [Ma10] and its open-source implementation *Apache Giraph* [Av11] are frameworks to define an iterative algorithm as a user-defined function that is called on every vertex. Within this function, the properties and neighbors of a vertex can be accessed, properties can be modified, messages can be sent to neighbors, and incoming messages from the previous iteration can be processed. These frameworks are typically not used for computations on graph databases but on graphs stored in a file, a relational database or a wide-column store. NotaQL supports both graphs in one of these formats and also graph databases. Furthermore, there is no need to have programming skills. One simple writes a

concise script to map input data to the output. The data-processing framework *Apache Spark* [Za10] has a component called *GraphX* [Xi13] which supports iterative graph analysis similar to Pregel. The graph has to be stored in a vertex table and an edge table. However, again, this framework is not used to work on graph databases. There are Spark connectors for graph databases, e.g. for Neo4J, but they offer a flattened look on the graph without the information about vertex neighbors. Edges can be traversed by writing Cypher queries within Spark methods. In our graph extension for NotaQL, edges are first-class citizens. It natively supports the full graph data model and not just a greatest common divisor between graphs and aggregate stores.

Spark can be used to implement a cross-system transformation from a graph database to a different database. However, these kinds of transformations do not work in the opposite direction, and they only work on simple graphs. There are special tools for graph imports and exports, but no generic ones like our NotaQL platform. The Neo4J Doc Manager [Ne16b] is a tool that loads documents from MongoDB [Mo16] into the graph database Neo4J. A vertex is created for every document *d* and one for every of their sub-documents *s* with an edge between *d* and *s*. This is very restricted and requires an extra effort on the source and target side before and after the import process to bring the data in the desired format. With the other languages and frameworks presented above, graph transformations to or from a different kind of data store are not possible.

Green-Marl [Ho12] is a language for graph analysis. It can be used to compute scalar values from the graph or a property for every vertex. Users can develop algorithms in a few lines of code which is then optimized and compiled into efficient parallel C++ code. Green-Marl is not a query or transformation language, but a programming language especially for graphs. Thus, users need to write code that defines *how* to compute a result, not *what* the result should be—as in SQL or NotaQL. In the paper, the authors claim that the PageRank algorithm can be expressed with 15 lines of code, in contrast to its native implementation in C++, which has 58 lines. As shown in Figure 11, the PageRank definition in NotaQL has only 5 lines of code.

7   Conclusions and Future Work

As a summary, we presented a language extension for NotaQL to define graph transformations as short transformation scripts. This way, we showed the extendability of NotaQL for complex data models. The language is more powerful than existing graph languages, it supports graph traversal, edge access and creation, and iterative algorithms. NotaQL can be used to modify a graph in place, to produce new graphs, and perform graph analytics using groupings and aggregations. We presented an approach for cross-system graph transformations between a graph database and any kind of NoSQL database or file format. For this, we used system-specific NotaQL engines and a combination of the Apache Spark framework and the Blueprints API for graphs. Our data–model-independant language fully supports the different data models, not the greatest common divisor of those. For that, we use tailored access paths for each model: graphs, documents, key-value pairs, tables, files, and more.

The most complex ones are graphs. For those, we presented an powerful and intuitive syntax to work with edges, neighbor vertices and their properties.

Our results show that our platform is very fast in graph transformations. This is because a NotaQL script is directly executed on a graph database using an API, not an intermediate query language. Cross-system transformations between graph and non-graph databases use the combination of the Blueprints API and the Apache Spark framework for distributed computations.

The NotaQL graph-transformation platform fulfills the four requirements for graph-processing software described in [Lu07]. It is *flexible* regarding data models and the data schema, *extensible* through user-defined engines and functions, *portable* to different database systems, and *maintainable* as a NotaQL script is concise and well understandable.

For future work, we want to improve the performance by integrating our Spark-based and graph-transformation platform closer. Instead of using an intermediate JSON file, we want to build a Tinkergraph engine for the Spark-based platform. This way, the in-memory graph database Tinkergraph can be accessed with Spark like a document store. After that, we plan more performance evaluations and tests. As the NotaQL language is independent of its implementation, more efficient implementations based on systems like Apache Giraph [Av11] or Green-Marl [Ho12] can follow.

References

[Ap16]   Apache Tinkerpop: 2016. `http://tinkerpop.apache.org`.

[Ar16]   ArangoDB: 2016. `https://www.arangodb.com`.

[Av11]   Avery, Ching: Giraph: Large-scale graph processing infrastructure on hadoop. Proceedings of the Hadoop Summit. Santa Clara, 11, 2011.

[Ch08]   Chang, Fay; Dean, Jeffrey; Ghemawat, Sanjay; Hsieh, Wilson C; Wallach, Deborah A; Burrows, Mike; Chandra, Tushar; Fikes, Andrew; Gruber, Robert E: Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 26(2):4, 2008.

[DG04]   Dean, Jeffrey; Ghemawat, Sanjay: MapReduce: Simplified Data Processing on Large Clusters. OSDI, pp. 137–150, 2004.

[Ge14]   Gessert, Felix; Friedrich, Steffen; Wingerath, Wolfram; Schaarschmidt, Michael; Ritter, Norbert: Towards a Scalable and Unified REST API for Cloud Data Stores. In: GI-Jahrestagung. pp. 723–734, 2014.

[Ho12]   Hong, Sungpack; Chafi, Hassan; Sedlar, Edic; Olukotun, Kunle: Green-Marl: a DSL for easy and efficient graph analysis. In: ACM SIGARCH Computer Architecture News. volume 40. ACM, pp. 349–362, 2012.

[Lu07]   Lumsdaine, Andrew; Gregor, Douglas; Hendrickson, Bruce; Berry, Jonathan: Challenges in parallel graph processing. Parallel Processing Letters, 17(01):5–20, 2007.

[Ma10]   Malewicz, Grzegorz; Austern, Matthew H; Bik, Aart JC; Dehnert, James C; Horn, Ilan; Leiser, Naty; Czajkowski, Grzegorz: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, pp. 135–146, 2010.

[Mo16]   MongoDB: 2016. `https://www.mongodb.org`.

[Ne16a]   Neo4J: 2016. `https://www.neo4j.com`.

[Ne16b]   Neo4j Doc Manager: 2016. `https://neo4j.com/developer/neo4j-doc-manager`.

[OPV14]   Ong, Kian Win; Papakonstantinou, Yannis; Vernoux, Romain: The SQL++ Query Language: Configurable, Unifying and Semi-structured. arXiv preprint arXiv:1405.3631, 2014.

[Pa99]   Page, Lawrence; Brin, Sergey; Motwani, Rajeev; Winograd, Terry: The PageRank Citation Ranking: Bringing Order to the Web. (1999-66), November 1999. Previous number = SIDL-WP-1999-0120.

[Pa15]   Paksoy, Volkan: From JSON to Neo4J. 2015. `http://volkanpaksoy.com/archive/2015/09/18/from-json-to-neo4j`.

[Pa16]   Parr, Terence: ANTLR. 2016. `http://www.antlr.org`.

[Pr08]   Prud'Hommeaux, Eric; Seaborne, Andy et al.: SPARQL query language for RDF. W3C recommendation, 15, 2008.

[Ro15]   Rodriguez, Marko A: The Gremlin graph traversal machine and language. In: Proceedings of the 15th Symposium on Database Programming Languages. ACM, pp. 1–10, 2015.

[SD15]   Schildgen, Johannes; Deßloch, Stefan: NotaQL Is Not a Query Language! It's for Data Transformation on Wide-Column Stores. In: British International Conference on Databases – BICOD 2015. 7 2015.

[SF12]   Sadalage, Pramod J.; Fowler, Martin: NoSQL Distilled: A brief guide to the emerging world of polyglot persistence. Addison-Wesley Professional, 1st edition, 2012.

[SGR15]   Schaarschmidt, Michael; Gessert, Felix; Ritter, Norbert: Towards Automated Polyglot Persistence. Datenbanksysteme für Business, Technologie und Web (BTW), 2015.

[SLD16]   Schildgen, Johannes; Lottermann, Thomas; Deßloch, Stefan: Cross-system NoSQL data transformations with NotaQL. In: Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond. ACM, p. 5, 2016.

[Su15]   Sun, Wen; Fokoue, Achille; Srinivas, Kavitha; Kementsietsidis, Anastasios; Hu, Gang; Xie, Guotong: SQLGraph: an efficient relational-based property graph store. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, pp. 1887–1901, 2015.

[Ti16]   Titan: Distributed Graph Database. 2016. `http://titan.thinkaurelius.com`.

[TZ12]   Takac, Lubos; Zabovsky, Michal: Data analysis in public social networks. In: International Scientific Conference and International Workshop Present Day Trends of Innovations. pp. 1–6, 2012.

[Xi13]   Xin, Reynold S; Gonzalez, Joseph E; Franklin, Michael J; Stoica, Ion: Graphx: A resilient distributed graph system on spark. In: First International Workshop on Graph Data Management Experiences and Systems. ACM, p. 2, 2013.

[Za10]   Zaharia, Matei; Chowdhury, Mosharaf; Franklin, Michael J; Shenker, Scott; Stoica, Ion: Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. volume 10, p. 10, 2010.