NotaQL Is Not a Query Language! It's for Data Transformation on Wide-Column Stores

Johannes Schildgen(✉) and Stefan Deßloch

University of Kaiserslautern, Kaiserslautern, Germany
{schildgen,dessloch}@cs.uni-kl.de

Abstract. It is simple to query a relational database because all columns of the tables are known and the language SQL is easily applicable. In NoSQL, there usually is no fixed schema and no query language. In this article, we present NotaQL, a data-transformation language for wide-column stores. NotaQL is easy to use and powerful. Many MapReduce algorithms like filtering, grouping, aggregation and even breadth-first-search, PageRank and other graph and text algorithms can be expressed in two or three short lines of code.

Keywords: NoSQL · Transformation · Language · Wide-column stores

1 Motivation

When we take a look at NoSQL databases[1], they differ from classical relational databases in terms of scalability, their data model and query method. The simplest form of such a database is a key-value store: One can simply write and read values using a key-based access. In this paper, we concentrate on wide-column stores. Such a store consists of tables that have one row-id column and one or more column families. Basically, each column family can be seen as a separate key-value store where column names function as keys. The three most popular wide-column stores are Google's Big Table [2], its open-source implementation Apache HBase[2], and Cassandra[3]. Figure 1 shows an example table with two column families.

ROW_ID	information			children	
	born	cmpny	salary	Susi	John
Peter	1967	IBM	€50000	€5	€10
	born	cmpny	salary	Susi	John
Kate	1968	IBM	€60000	€20	€0
	born	school			
Susi	1989	Eton			
	born	school			
John	1991	Eton			

Fig. 1. Person table with a children graph and amounts of pocket money

At first sight, the table looks similar to a relational table. This is because both consist of columns and these columns hold atomic values. In relational

[1] http://nosql-database.org.
[2] http://hbase.apache.org.
[3] http://cassandra.apache.org.

© Springer International Publishing Switzerland 2015
S. Maneth (Ed.): BICOD 2015, LNCS 9147, pp. 139–151, 2015.
DOI: 10.1007/978-3-319-20424-6_14

databases, however, the database schema is static, i.e., all columns of a table are known, before values are inserted or modified. In contrast, in a wide-column store, at each insertion, one is able to set and create arbitrary columns. In other words, the database schema does not exist, or is dynamically evolving. The first column family *information* contains attributes of people. Note that different rows can have different columns which are not predefined at table-creation time. The second column family *children* models a graph structure. The names in the columns are references to row-ids of children and the values are the amounts of pocket money the children get from their parents. We will later use this table as an example for all of our NotaQL transformations. Web graphs are very akin to this example: The first column family comprises information about a web site, while the second contains links to other web sites.

If the table in Fig. 1 was stored in HBase, one could use a *Get* operation in the *HBase Shell* or the Java API to fetch a row with all its columns by its row-id. In HBase, there always is an index on the row-id. Other secondary indexes are not supported. To execute more complex queries, programmers can utilize a framework that allows access via an SQL-like query language. The most prominent system for that is Hive [21]; others are presented in the next section.

As an alternative, one may consider generating redundant data which then can be accessed via simple *Get* operations. This approach shows similarities with materialized views in traditional relational DBMS [7]. In [13], ideas are presented to do selections, joins, groupings and sorts by defining transformations over the data. The authors advocate that one does not need a query language like SQL when the data is stored in the way it is needed at query time. If finding all people with a specific year of birth is a frequent query, the application which modifies data should maintain a second table whose row-id is a year of birth and columns are foreign keys to the original row-id in the main table. As a drawback, applications have to be modified carefully to maintain all the tables, so every change in the base data immediately leads to many changes in different tables. In [5], similar approaches are presented to maintain secondary indexes on HBase, either with a dual-write strategy or by letting a MapReduce [3] job periodically update an index table.

In this paper, we present NotaQL, a data-transformation language for wide-column stores. Like SQL, it is easy to learn and powerful. NotaQL is made for schema-flexible databases, there is a support for horizontal aggregations, and metadata can be transformed to data and vice versa. Complex transformations with filters, groupings and aggregations, as well as graph and text algorithms can be expressed with minimal effort. The materialized output of a transformation can be efficiently read by applications with the simple Get API.

In the following section, we present some related work. In Sect. 3, NotaQL is introduced as a data-transformation language. We present a MapReduce-based transformation platform in Sect. 4 and the last section concludes the article.

2 Related Work

Transformations and queries on NoSQL, relational and graph databases can be done by using different frameworks and languages. With Clio [8], one can perform a schema mapping from different source schemata into a target schema using a graphical interface. Clio creates views in a semi-automatic way which can be used to access data from all sources. This virtual integration differs from our approach because NotaQL creates materialized views. Clio can only map metadata to metadata and data to data. There is no possibility to translate attribute names into values and vice versa. In [1], a copy-and-paste model is presented to load data from different sources into a curated database. Curated databases are similar to data warehouses, but here it is allowed to modify data in the target system. A tree-based model is used to support operations from SQL and XQuery as well as copying whole subtrees. The language presented in that paper also contains provenance functions to find out by which transaction a node was created, modified or copied. Although the language is very powerful, it does not support aggregations, unions and duplicate elimination because in these cases, the origin of a value is not uniquely defined.

There are many approaches to query wide-column stores using SQL, e.g. Hive, Phoenix[4] or Presto[5]. On the one hand, one does not need to learn a new query language and applications which are based on relational databases can be reused without many modifications. On the other hand, SQL is not well-suited for wide-column stores, so the expressiveness is limited. Figure 16 at the end of this paper shows the weak points of SQL: Transformations between metadata and data, horizontal aggregations and much more can not be expressed with an SQL query. Furthermore, many frameworks do not support the schema flexibility of HBase. Before an HBase table can be queried by *Hive*, one has to create a new Hive table and define how its columns are mapped to an existing HBase table[6]. With *Phoenix*, an HBase table can be queried with SQL after defining the columns and their types of a table with a `CREATE TABLE` command. *Presto* is an SQL query engine by Facebook. The presto coordinator creates an execution plan for a given query and a scheduler distributes the tasks to the nodes that are close to the data. Usually, Presto directly accesses data that is stored in the Hadoop distributed file system but connectors for other systems, e.g. HBase, exist as well. The strength of Presto is a nearly full ANSI-SQL support—including joins and window functions—and its ten times higher speed than Hive and MapReduce. But again, only relational queries on relational tables with static schemas are possible.

The HBase Query Language by Jaspersoft[7] can be used to support more complex queries on an HBase table. It is better suited for wide-column stores than SQL, but not easy to use. One has to define a query as a JSON document that can be very long, even for simple queries. The syntax of our language NotaQL is inspired by *Sawzall* [19], a programming language used by Google to define log processing tasks instead of manually writing a MapReduce job. The input of a Sawzall script is one single line of input (e.g. a log record) and the

[4] http://phoenix.apache.org.
[5] http://prestodb.io.
[6] https://cwiki.apache.org/confluence/display/Hive/HBaseIntegration.
[7] https://community.jaspersoft.com/wiki/jaspersoft-hbase-query-language.

output are insertions into virtual tables. A Sawzall script runs as a MapReduce job and the input and output is not an HBase table but a CSV file. The language *Pig Latin* [17] provides relational-algebra-like operators to load, filter and group data. Pig programs can be interconnected with a workflow manager like Nova [16]. Google BigQuery [20] is the publicly-available version of Dremel [14]. One can import and analyze data that is stored in the Google Cloud Storage using SQL. As the data is stored in a column-oriented manner, it can be filtered and aggregated very fast. In the paper, it is recommended to use BigQuery in combination with MapReduce. First, MapReduce can join and pre-process data, then this data can be analyzed using BigQuery. As NotaQL transformations are based on MapReduce, one can replace the complex MapReduce transformations by NotaQL scripts and combine them with fast query languages like BigQuery, Phoenix, or HBase QL.

Graph-Processing Frameworks. The language Green-Marl [9] is used to describe graph-analysis algorithms. Green-Marl programs are compiled into multi-threaded and highly efficient C++ code. In contrast to NotaQL, Green-Marl was not designed for graph transformations but to calculate a scalar value (e.g. diameter) of the graph, to add a property to every node (e.g. PageRank, see Sect. 3.3), or to select a subgraph of interest. Pregel [12] is a system to develop iterative graph algorithms by exchanging messages between nodes. Every node can send messages to its neighbors and change its properties and its state depending on incoming messages. When there is no incoming message, a node becomes inactive. When all nodes are inactive, the algorithm terminates. In PowerGraph [6], instead of sending messages, every node computes a MapReduce job for all its incoming edges (Gather phase) and computes a new value for a node attribute (Apply phase). In the end, for each edge, a predicate is evaluated (Scatter phase). If it is false for every incoming edge of a node, the node becomes inactive and it will be skipped in the next Gather-Apply-Scatter (GAS) round. Pregel and PowerGraph are not well-suited for graph construction and transformation. GraphX [23] is an extension of Spark [24] and it solves this problem by introducing a Resiliant Distributed Graph (RDG) interface for graph construction, filters, transformations, mappings, updates and aggregations. GraphX algorithms are seven times slower than PowerGraph jobs but eight times faster than native Hadoop jobs because they use a tabular representation of the graph and a vertex-cut partitioning over many worker nodes. As the GraphX interface is not easy to use, it is recommended to develop one's own API based on that interface. Pregel and PowerGraph can be reimplemented using GraphX in twenty lines of code. The NotaQL language is more user-friendly. We are planning to use the GraphX interface to develop a NotaQL-based API. This makes graph processing not only fast but also easy to develop. With Naiad [15], one can define a computation as a dataflow graph. A vertex can send messages to other vertexes for the next iteration which is executed in an incremental fashion. Like GraphX, it is recommended not to use Naiad directly but to build libraries for higher languages on top of it.

3 Transformations on Wide-Column Stores

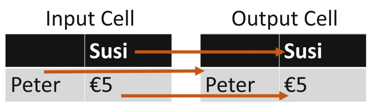
In this section, we present the NotaQL language to define transformations. The first examples can be solved with SQL as well, but later in this article, there are graph algorithms and others which are not expressible in SQL.

3.1 Mapping of Input Cells to Output Cells

As we learned in the motivation section, each row in a wide-column store has a unique row-id. In the following examples, we access columns independent of their column family. If a table consists of multiple column families, their names can be used as a prefix, e.g. `information : born` instead of `born`. Each row can have an arbitrary number of columns and the column names are unique within one row. The combination of row-id and column name `(_r, _c)` is called a *cell*. Each cell has one atomic value, so the triple `(_r, _c, _v)` represents one cell together with its value, for example `(Peter, born, 1967)`[8].

The basic idea of NotaQL is to define mappings between input and output cells, or—more precisely—to specify how to construct output cells based on the input. These mappings are executed in three steps: (1) Selection on the input table; (2) For each row, split it into cells and perform a cell mapping; (3) Aggregate all values for the same output cell using an aggregate function.

Figure 2 shows the identity mapping where each cell is simply copied. Here, no row selection and no aggregation function is used.



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

Fig. 2. Table copy with NotaQL

When this transformation is executed, a snapshot of the input table is analyzed row by row. In every row, for each of its cells an output cell will be produced with exactly the same row-id, column name and value. So the result of this transformation looks just like the input. An equivalent SQL query would be: `INSERT INTO out (SELECT * FROM in)`. It copies a full table. HBase comes with a backup tool *CopyTable*[9] to solve this problem.

When not all columns should be copied, but only the `salary` and `born` columns, the second line in the table-copy example can be replaced by `OUT.salary <- IN.salary, OUT.born <- IN.born`.

Figure 3 shows the NotaQL syntax in BNF. We will see that most algorithms can be specified in one block containing one row and one cell specification. We illustrate the syntax further in the following subsections. In general, a NotaQL script can consist of many blocks to perform multiple cell mappings in one transformation.

[8] These triples are known as entity-attribute-value or object-attribute-value. They are very flexible regarding the number of attributes of each entity.

[9] http://blog.cloudera.com/blog/2012/06/online-hbase-backups-with-copytable-2/.

```
   ⟨NotaQL⟩ ⊨ [IN-FILTER: ⟨predicate⟩, ]⟨rowspec⟩, ⟨cellspec⟩(, ⟨cellspec⟩) * [; ]
  ⟨rowspec⟩ ⊨ OUT._r <- ⟨vdata⟩
 ⟨cellspec⟩ ⊨ OUT.(⟨colname⟩ | $(⟨input⟩)) <- (⟨vdata⟩ | ⟨aggfun⟩(⟨vdata⟩))
    ⟨input⟩ ⊨ (IN._r | IN.[⟨colfamily⟩ :](_c | _v) | IN.⟨colname⟩)[?(⟨predicate⟩)]
    ⟨vdata⟩ ⊨ ⟨input⟩ | ⟨const⟩ | ⟨vdata⟩(+ | − | * | /)⟨vdata⟩
   ⟨aggfun⟩ ⊨ COUNT | SUM | MIN | MAX | AVG
    ⟨const⟩ ⊨ '(A...Z | a...z | 0...9) + ' | (0...9) +
  ⟨colname⟩ ⊨ [⟨colfamily⟩ :](A...Z | a...z | 0...9) +
⟨colfamily⟩ ⊨ (A...Z | a...z | 0...9) +
⟨predicate⟩ ⊨ (⟨colname⟩ | @ | col_count([⟨colfamily⟩]))[⟨op⟩(⟨colname⟩ | ⟨const⟩]
                | (NOT | !)⟨predicate⟩ | ⟨predicate⟩(AND | OR)⟨predicate⟩
       ⟨op⟩ ⊨ = | ! = | < | <= | > | >=
```

Fig. 3. NotaQL language definition (simplified)

3.2 Predicates

```
IN-FILTER: born>1950,
OUT._r <- IN._r,
OUT.salary <- IN.salary
```

Fig. 4. Row predicate

```
OUT._r <- IN._r,
OUT.$(IN._c?(@='€5')) <- IN._v
```

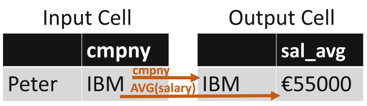
Fig. 5. Cell predicate

There are two kinds of predicates in NotaQL: a *row predicate* which acts as an input-row filter to perform a row selection and a *cell predicate* which selects specific cells in a row. The row predicate is an optional filter definition placed at the beginning of a NotaQL script using an `IN-FILTER` clause. If such a predicate is set, every row in the input table which does not satisfy it will be skipped. That means, before a mapping is performed, a whole row is handled as if it would not exist when the predicate is evaluated as false. In this predicate, comparison and logical operators as well as column names and constants can be used.

The transformation in Fig. 4 is executed as follows: Only rows that contain a column `born` with a value greater than 1950 are selected. The rest of the rows are skipped. In the remaining rows, only the column `salary` is read and returned. The result is one table with only one column `salary` and between zero and `n` rows, where `n` is the number of rows in the base table. The transformation is equivalent to the SQL query `SELECT salary FROM in WHERE born>1950`. Some more examples for row predicates:

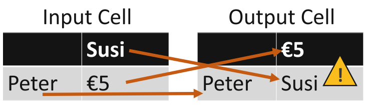
- `(born>1950 AND born<1960) OR cpny='IBM' OR col_count()>5`,
- `school` respectively `!school`— checks column existence / absence in a row.

When cells should be filtered within one row without knowing their names, a *cell predicate* can be used. It starts with a `?` and can be placed after an `IN._c` or `IN._v`. The transformation in Fig. 5 only copies columns with a value equal to €5, independent of their names. The question mark indicates the begin of a predicate so that cells are skipped which do not satisfy it. The `@` symbol is used to refer to the current cell's value. A cell predicate can also be used to drop columns, e.g. `OUT.$(IN._c?(!name)) <- IN._v`.



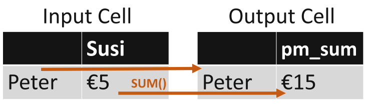
```
OUT._r <- IN.cmpny,
OUT.sal_avg <- AVG(IN.salary)
```

Fig. 6. Aggregation: AVG



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

Fig. 7. Corrupt NotaQL script



```
OUT._r <- IN._r,
OUT.pm_sum <- SUM(IN.children:_v)
```

Fig. 8. Aggregation: SUM

SQL does not support predicates for column existence or absence. Furthermore, it is not possible to drop columns or check values of columns independent from their names. For wide-column stores these predicates are necessary because of their schema flexibility.

The logical execution of a NotaQL transformation starts with splitting each input row into its cells. After a cell mapping is performed, new row-ids together with columns of an output row are collected. If there is more than one value for the same row-id/column pair, an implicit grouping is performed and the user has to define how the final value is aggregated based on the single values.

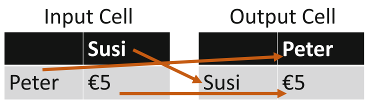
A popular example query is calculating the average salary values per company. In SQL, this is done by grouping and aggregation. Here, a row-id in the output table should be a company name which is stored as a value in the column `cmpny`. So, the first NotaQL mapping is `OUT._r <- IN.cmpny,`. The output column name is set to `'sal_avg'`. For setting the value of the output cell, the salary has to be read and summed up (see Fig. 6).

In the very first example which copies a table, the output mapping is uniquely defined in the sense that there is a single value for each row-id/column pair. This can be easily proven: As each input cell `(_r,_c)` is unique, each output cell `(_r,_c)` is unique, too. Figure 7 shows a transformation where the uniqueness is not given because different input cells can have equal values. Whenever a transformation can produce multiple values for the very same cell `(OUT._r, _c)`, an aggregation function must be used. The output table of the query in Fig. 8 has the same number of rows as the input table, but each row only consists of one column (`pm_sum`) with the sum of all column values. These *horizontal aggregations* are not possible in SQL.

3.3 Graph-Processing Applications

The queries from the previous subsections are typical log-processing queries with projection, selection, grouping, and aggregation. These operations are well known from the relational algebra and from SQL and have been generalized further in NotaQL for wide-column stores. But NotaQL supports more complex computations as well. In this section, we show that graph-processing algorithms like PageRank and breadth-first search can be implemented in NotaQL. This illustrates the power of the simple NotaQL language and demonstrates that it enables new kinds of transformations which are not possible with classical query languages yet.

Graphs are often modeled as adjacency lists in a wide-column store. Each row represents one vertex in a graph and each column represents an edge to another vertex. If the edges are weighted, the value of a column contains the weight. In a relational database, columns are part of the meta-data level of a table. In a wide-column store, they are part of the data level. This is why SQL is not well-suited for graph algorithms.



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

Fig. 9. Reversing a graph

Reversing a Graph. A simple graph algorithm that reverses the directions of edges in a graph can be defined by simply taking the first example *Table Copy* and swapping `IN._r` and `IN._c` (see Fig. 9). When this script is executed on our example table, it will produce a new table where for every person their parents can be found. On a web-link graph, this script produces an inverted graph, i.e. a list of web sites together with their incoming links. The given script can be extended to manipulate the graph structure. In the transformation in Fig. 10, row and cell predicates are used to remove vertices (people with less than two children) and edges (to children receiving €10 or less).

```
IN-FILTER: col_count()>=2,
OUT._r <- IN._c,
OUT.$(IN._r) <-
        IN.children:_v?(@>10);
```

Fig. 10. Parents of persons with two or more children that give more than €10 of pocket money.

PageRank. The PageRank algorithm is an iterative algorithm to rank a vertex in a graph depending on the rank of vertices pointing to it. The full algorithm can be found in [18]. Here we concentrate only on the most interesting part of the PageRank formula—the random-jump factor is not relevant for our discussion and is therefore omitted. The NotaQL script for computing the PageRank is very close to its mathematical definition (see Fig. 11).

The idea is to start with a PageRank value of $\frac{1}{n}$ for each vertex (with n being the number of vertices in the graph) and running some iterations of the formula above until the PageRank values converge. In a wide-column store, the graph is stored in a table with the row-id being the vertex identifier, one column `PR` in the column family `alg` with the starting value of $\frac{1}{n}$ and one column for each outgoing edge in the column family `edges`.

The fraction between the PageRank value and the outdegree of a node y can be used as one addend of the new PageRank values of the nodes x to whom y has an outgoing edge. In our example table, the outdegree is the number of columns

$$PR(x) = \sum_{y|y\rightarrow x} \frac{PR(y)}{outdegree(y)}$$

```
OUT._r <- IN._c,
OUT.alg:PR <- SUM(IN.alg:PR/col_count(edges));
```

Fig. 11. The PageRank algorithm

in the column family `edges`. In this transformation, the input and output tables are the same, so there are no steps needed to preserve the graph structure. The results are only updated output cells for the column `PR`. An example: Nodes A (PR: 0.1) and B (PR: 0.3) have one outgoing edge each, namely to node C. So, C's new PageRank value is $\frac{0.1}{1} + \frac{0.3}{1} = 0.4$.

Like updates in SQL, NotaQL transformations have snapshot semantics. This means, logically the full input is read, then all cells are mapped to output cells and at the end the output is written. So writes into the input table during job-execution do not interfere with the remaining transformation process. For our example, the execution framework has to decide after each execution whether more iterations are needed or not. PageRank can be executed iteratively until the changes of the PageRank values are below a specific accuracy value. One approach to control the number of iterations is a change measurement after each iteration. Depending on the amount of changes since the previous iteration, a new run is started or the overall job terminates. Another approach is the usage of an input format that compares the last two versions of each cell value and ignores a row when the changes are below a threshold. Then, the job terminates when the input is empty.

Breadth-First Search. The distance between two vertices in a graph is the number of edges on the shortest path between them. In a weighted graph, it is the sum of (positive) weights of those edges. Breadth-first search [11] can be used to compute the distance from one predefined vertex V_0 to every other vertex. Therefore, a `dist` column is added for start vertex V_0 with the value 0. For all other vertices, the distance is ∞. This can be modeled by the absence of the `dist` column in the column family `alg`.

The NotaQL script in Fig. 12 is executed iteratively until the result does not change anymore. In a connected graph, the number of iterations is equal to the diameter of the graph. In each iteration, neighbors of vertices whose distance are known are updated.

The `IN-FILTER` skips rows with an unknown distance. For the others, the distance of each neighbor vertex is set to the vertex' own distance plus one. If multiple vertices have an edge to the same neighbor, the minimum value is taken. If the algorithm should take weighted edges into account, the `1` in the last line has to be replaced by `IN._v` to add the current edge weight to the own distance.

```
IN-FILTER: dist
OUT._r <- IN._c,
OUT.dist <- MIN(IN.dist+1),
```

Fig. 12. Breadth-first search

3.4 Text Processing

We extended the NotaQL language with a `split` function. It has one input parameter for a delimiter and it splits text values in multiple ones. Figure 13 shows a NotaQL transformation which counts the occurrences of each word in all input

```
OUT._r <- IN._v.split(' '),
OUT.count <- COUNT()
```

Fig. 13. Word-count algorithm

cells. The output is a table where for each word (row-id) a column `count` holds the number of occurrences of the word in all input cells.

With a small modification in the word-count script, one can calculate a *term index* with NotaQL: `OUT.$(IN._r) <- COUNT();` Here, each term row contains a count value for each document that contains the term. These can be used to support an efficient full-text search over large text data. In addition to these examples, many other graph and text algorithms can be expressed in NotaQL. For example, the computation of TF-IDF (term frequency/inverse document frequency) is a chain of three NotaQL transformations.

4 NotaQL Transformation Platform

There are different possibilities to execute NotaQL scripts. They can be mapped to other languages using a wrapper, the direct API to a wide-column store can be used, or one could make use of a framework like MapReduce. In this section, we present a MapReduce-based transformation platform with full NotaQL language support. It is accessed via a command line interface or with a GUI. The GUI can be used to plan, execute and monitor NotaQL transformations [4]. When writing a NotaQL script, the tool immediately visualizes the cell mapping using arrows, as in the figures in Sect. 3. Alternatively, the user can work just graphically by defining arrows between cells. The GUI user can define an update period, i.e. a time interval in which a script will be recomputed.

When a transformation is started, the input table is read row by row. Rows which violate the row predicate are skipped. Each remaining cell fulfilling the cell predicate is mapped to an output cell in the way it is defined in the NotaQL script. Cells with the same identifier are grouped and all its values are aggregated. We used the Hadoop[10] MapReduce framework for the execution because of its advantages for distributed computations, scalability, and failure compensation. Reading and transforming input cells is done by the *Map* function, the Hadoop framework sorts and groups the output cells, and finally, the aggregation and the write of the final output is done by the *Reduce* function.

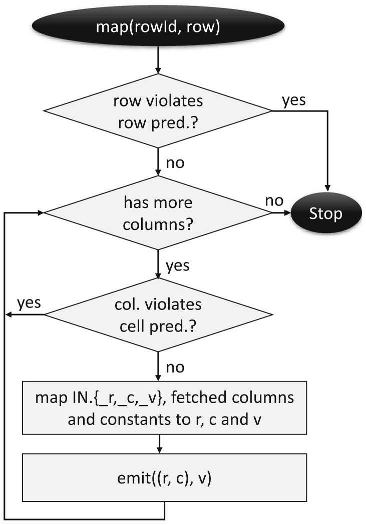


Fig. 14. NotaQL map function

Map. The input for one Map function is one row from the input table that consists of a row-id and a set of columns and values. Figure 14

[10] http://hadoop.apache.org.

shows how predicates are evaluated and the map output is produced. The Map-output key is a combination of an output row-id and a column qualifier. So, each Reduce function processes all the values for one specific cell. It is efficient to use a Partitioner function which transfers the data directly to the node which is responsible for storing rows with the given row-id.

Reduce and Combine. Figure 15 shows that the Reduce function just produces one output cell by aggregating all values for one column in a row. Similar to this, a generic Combine function can aggregate existing values as well. So, the network traffic is reduced and the Reducers have to aggregate fewer values for each cell.

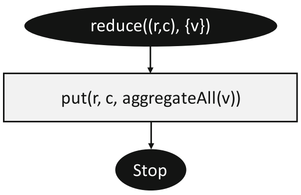


Fig. 15. NotaQL reduce function

	SQL	NotaQL
add/drop columns	(✔)DDL Statements (`ALTER TABLE`)	✔ `OUT.newCol <- ...`
check column existence	(✔)`col IS NOT NULL`; query system catalog	✔ use a predicate
transform data to metadata	✘ only with SchemaSQL [10]/FISQL [22]	✔ `OUT.$(IN._v) <- ...`
transform metadata to data	(✔)query system catalog; SchemaSQL/FISQL	✔ `OUT.xyz <- IN._c`
horizontal aggregation	✘ not possible	✔ `SUM(IN._v)`
graph processing	(✔)many joins over vertex/edge table	✔ see section 3.3
text processing	✔ e.g. `regex_split_to_table` in PostgreSQL	✔ see section 3.4

Fig. 16. SQL is not well-suited for wide-column stores, 4 easy (4) hard 8 impossible

5 Conclusion

In this paper, we presented NotaQL, a transformation language for wide-column stores that is easy to use and very powerful. With minimal effort, selections, projections, grouping and aggregations can be defined as well as operations which are needed in schema-flexible NoSQL databases like HBase. Each transformation consists of a mapping between input and output cells and optional predicates.

Figure 16 shows the limits of SQL and that NotaQL supports operations which are impossible or difficult to express in SQL. SQL is suitable for well-defined relational schemata, but not for wide-column stores where rows can have arbitrary columns. That's why we introduced the NotaQL language and we showed that it is very powerful. Not only relational operations are supported but also graph and text algorithms. One should choose a language depending on the given data model. No one would use SQL instead of XQuery on XML documents and as NoSQL stands for "Not only SQL", there is a need for new query and transformation languages.

As the title of this paper says, NotaQL is not a query language like SQL or XQuery. On the other hand, one can argue, there is no distinction between a query and a transformation language. But NotaQL is specialized for transformations over large tables and not for ad-hoc queries. NotaQL does not have an API. Queries are executed periodically to perform a data transformation on wide-column stores. The output table can be accessed in the application with a primitive GET API and the up-to-dateness of the data is defined by the query-execution interval.

We are currently working on language extensions for NotaQL to support more complex transformations, e.g. Top-k algorithms. For faster transformations, we are implementing an incremental component in our framework. This means, a transformation can reuse the results from a former run and it has only to read the delta. Currently, only standalone transformations are supported. Iterative algorithms need to be executed through a batch script which checks a termination criterion and supervises the iterations. A language extension for iterative transformations is planned.

Although all experiments are based on the NoSQL database system HBase, NotaQL scripts can be defined on other wide-column stores and other NoSQL and relational databases as well. Next, we will apply our findings to Cassandra because the support of secondary indexes in Cassandra enables better optimizations for NotaQL computations. Our vision is for cross-platform transformations. Then, the input and output of a NotaQL transformation can be any data source from a relational or NoSQL database. So, one can transform a CSV log file into an HBase table, load a graph from HypergraphDB into MySQL or integrate data from Cassandra and a key-value store into MongoDB.

References

1. Buneman, P., Cheney, J.: A copy-and-paste model for provenance in curated databases. Notes **123**, 6512 (2005)
2. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. (TOCS) **26**(2), 1–14 (2008). Article 4
3. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)
4. Emde, M.: GUI und testumgebung für die HBase-schematransformationssprache NotaQL. Bachelor's thesis, Kaiserslautern University (2014)
5. George, L.: HBase: The Definitive Guide, 1st edn. O'Reilly Media, Sebastopol (2011)
6. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: distributed graph-parallel computation on natural graphs. In: OSDI, vol. 12, p. 2 (2012)
7. Gupta, A., Jagadish, H.V., Mumick, I.S.: Data integration using self-maintainable views. In: Apers, P.M.G., Bouzeghoub, M., Gardarin, G. (eds.) EDBT 1996. LNCS, vol. 1057, pp. 140–144. Springer, Heidelberg (1996)
8. Hernández, M.A., Miller, R.J., Haas, L.M.: Clio: A semi-automatic tool for schema mapping. ACM SIGMOD Rec. **30**(2), 607 (2001)
9. Hong, S., Chafi, H., Sedlar, E., Olukotun, K.: Green-marl: a DSL for easy and efficient graph analysis. ACM SIGARCH Comput. Archit. News **40**(1), 349–362 (2012)
10. Lakshmanan, L.V.S., Sadri, F., Subramanian, I.N.: SchemaSQL-a language for interoperability in relational multi-database systems. In: VLDB, vol. 96, pp. 239–250 (1996)

11. Lin, J., Dyer, C.: Data-intensive text processing with MapReduce. Synth. Lect. Hum. Lang. Technol. **3**(1), 1–177 (2010)
12. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, pp. 135–146. ACM (2010)
13. Grinev, M.: Do You Really Need SQL to Do It All in Cassandra? (2010). http://wp.me/pZn7Z-o
14. Sergey, M., Andrey, A., Long, J.J., Romer, G., Shivakumar, S., Tolton, M., Vassilakis, T.: Dremel: interactive analysis of web-scale datasets. Commun. ACM **54**(6), 114–123 (2011)
15. Murray, D.G., Sherry, F.M.C., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: a timely dataflow system. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 439–455. ACM (2013)
16. Olston, C., Chiou, G., Chitnis, L., Liu, F., Han, Y., Larsson, M., Neumann, A., Rao, V.B.N., Sankarasubramanian, V., Seth, S., et al.: Nova: continuous pig/hadoop workflows. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, pp. 1081–1090. ACM (2011)
17. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 1099–1110. ACM (2008)
18. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: bringing order to the web. Technical report 1999–66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120
19. Pike, R., Dorward, S., Griesemer, R., Quinlan, S.: Interpreting the data: parallel analysis with sawzall. Sci. Program. **13**(4), 277–298 (2005)
20. Sato, K.: An inside look at google bigquery. White paper (2012). https://cloud.google.com/files/BigQueryTechnicalWP.pdf
21. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: a warehousing solution over a map-reduce framework. Proc. VLDB Endow. **2**(2), 1626–1629 (2009)
22. Wyss, C.M., Robertson, E.L.: Relational languages for metadata integration. ACM Trans. Database Syst. (TODS) **30**(2), 624–660 (2005)
23. Xin, R.S., Gonzalez, J.E., Franklin, M.J., Stoica, I.: Graphx: a resilient distributed graph system on spark. In: First International Workshop on Graph Data Management Experiences and Systems, p. 2. ACM (2013)
24. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, p. 2. USENIX Association (2012)