Datenbank Spektrum (2017) 17:245–253
https://doi.org/10.1007/s13222-017-0265-6

SCHWERPUNKTBEITRAG

Preserving Recomputability of Results from Big Data Transformation Workflows

Depending on External Systems and Human Interactions

Matthias Kricke[1] · **Martin Grimmer**[1] · **Michael Schmeißer**[2]

Received: 9 June 2017 / Accepted: 24 August 2017 / Published online: 12 September 2017

Abstract The ability to recompute results from raw data at any time is important for data-driven companies to ensure data stability and to selectively incorporate new data into an already delivered data product. However, data transformation processes are heterogeneous and it is possible that manual work of domain experts is part of the process to create a deliverable data product. Domain experts and their work are expensive and time consuming, a recomputation process needs the ability of automatically adding former human interactions. It becomes even more challenging when external systems are used or data changes over time. In this paper, we propose a system architecture which ensures recomputability of results from big data transformation workflows on internal and external systems by using distributed key-value data stores. Furthermore, the system architecture will contain the possibility of incorporating human interactions of former data transformation processes. We will describe how our approach significantly relieves external systems and at the same time increases the performance of the big data transformation workflows.

Keywords BigData · Recomputability · System architecture · Bitemporality · Time-to-consistency

✉ Matthias Kricke
kricke@informatik.uni-leipzig.de

Martin Grimmer
grimmer@informatik.uni-leipzig.de

Michael Schmeißer
michael.schmeisser@mgm-tp.com

1 Leipzig University, Augustusplatz 10, 04109 Leipzig, Germany

2 mgm technology partners, Neumarkt 2, 04109 Leipzig, Germany

1 Introduction

For data-driven organizations, the possibility to selectively incorporate new data into an already calculated data product (like a chart, report or recommendation, etc.) can be required. Hence, the option to recompute information from their raw data is needed, even if this data comes from several external systems. In addition those recomputable data transformation processes may contain human interactions of domain experts as a step towards the end result. It is obvious that temporal features are necessary for this kind of recomputability. In the past, a lot of research has been done on temporal databases [10]. Temporal features have also been incorporated in the SQL:2011 standard [8]. Problems like concurrency control [5] have been solved for modern, distributed systems of e.g. Microsoft [4], Google [3] and SAP [9]. Unfortunately, those systems either can't be used on-premises or have high licensing costs. Additionally, recomputability of data products in big data systems with dependencies to external systems and manual changes to the data has not been addressed in recent research. However, this is a problem which mgm technology partners GmbH has been asked to solve for a customer. The recomputability of data products enables mgm's customer to reconstruct earlier data versions, reports and analysis results to compare them with newer ones. Moreover, it is now possible to incorporate data changes only from specific external systems. To deal with the requirements of low license costs and an on-premises system, mgm's customer has decided to use a scalable and distributed key-value data store like Apache Accumulo[1], Apache HBase[2] or Apache Cassandra[3].

1 https://accumulo.apache.org/
2 https://hbase.apache.org/
3 https://cassandra.apache.org/

Nonetheless, those stores do not offer a proper solution for recomputability on their own.

In this paper, we present *ELSA* (the **E**xterna**L** **S**ystem **A**daptor) and *ANNA* (the **A**utomated Ma**N**ual Actio**N** **A**pplication System), both building a big data system architecture which ensures recomputability of data products with human interactions and dependencies to external systems by using a variant of the concept of bitemporality [6]. We show an efficient way to recompute data products even in scenarios where the external systems aren't versioned.

Customer specific application details are confidential, yet we will provide simple examples to follow our explanations in the next sections.

2 Requirements

Mgm's customer wants a cost efficient system which is capable of handling external systems from either other company departments or companies as data sources. There is a strong requirement for distributed data transformation processes [11]. Those processes are incorporating data from external systems. This leads to the demand to relieve the external systems from high-frequent distributed requests. Furthermore, intermediate results are subject to manual or automated quality checks by domain experts which may lead to adjustments on the intermediate results. During (re)computation, manual adjustments made by domain experts shall be applied automatically when certain preconditions are met.

A solution has to be linearly scalable with low to no license costs and must support many simultaneous, distributed data transformation processes. In addition, it has to be able to selectively incorporate new data into an already calculated data product by recomputing it. Hence, all records are immutable and each version of a record has to be stored. On the one hand, a high volume of external data is sent to the system. On the other hand, the system has to store all versions of this data, which leads to a data base increasing by several terabytes a month.

As mentioned before, recomputability can be ensured by versioning the data.

Definition 1 (Versioning) A record is versioned if each of its occurred states is accessible with the corresponding timestamp. A system is versioned if each of its records is versioned.

Reality shows that full versioning in external systems is nothing that can be relied on. Furthermore, it is possible that external systems do not meet the latency or throughput requirements of the distributed data transformation process. Therefore, the system has to ensure scalability, recomputability, high throughput and low latency itself which leads to the necessity of a suitable big data system architecture.

3 External System Adaptor

To deal with the requirements stated in Sect. 2 a decoupling of external systems and the data transformation process is necessary. The **E**xterna**l** **S**ystem **A**daptor (*ELSA*) is the junction shown in Fig. 1, which fulfills all requirements. However, some demands to an external system are inevitable. Every external system used in the data transformation process needs a defined interface which provides a change history on their data. Furthermore, each change needs to be well defined as operation as stated in Definition 2.

Definition 2 (External key-value Operations) There are insert and delete operations. An insert operation is defined as a tuple

$$insert = (k, t_\mathrm{e}, v).$$

Where k is the key, t_e is the event timestamp and v is the value of the external record.

The delete operation is defined as a tuple

$$delete = (k, t_\mathrm{e})$$

which invalidates every record with an event timestamp lower or equal to t_e.

Assuming an external system inserted the tuples $t_1 = (a, 1, v)$, $t_2 = (a, 2, v')$ and $t_3 = (a, 3, v'')$, a delete operation defined as $delete = (a, 2)$ would remove t_1 and t_2 in the external system and retain t_3, since the timestamps of t_1 and t_2 are lower or equal to the timestamp of the delete operation. However, to preserve recomputability, in *ELSA* data is not deleted but instead not returned in a get request.

To fulfil the aforementioned requirements *ELSA* consists of:

- the *ELSA Store*, which is replacing the external systems functionality by providing a queryable data history and
- the *ELSA Data Synchronization*, which is used for keeping the *ELSA Store* synchronized with the external systems.

Hence, distributed data transformation processes now use the *ELSA Store* instead of the external systems, which removes the logic of accessing external systems from them. Moreover, the *ELSA Data Synchronization* highly reduces the throughput and latency requirements for an external

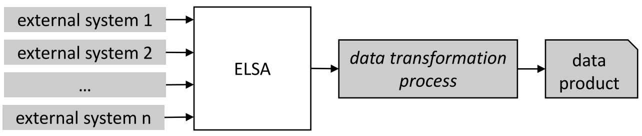

Fig. 1 Decoupling of external systems and the data transformation process with *ELSA*

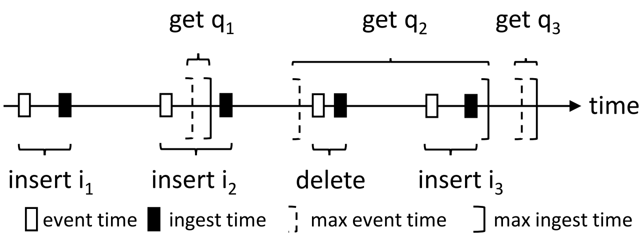

Fig. 2 All data versions of a record with different ingest and event times with several get requests

system since data is only read once by it. Since those requirements now have to be handled by the store, it has to be a distributed, scalable, multi-version key-value data store. Furthermore, the *ELSA Store* has to be located within the same cluster infrastructure as the big data transformation processes to meet the requirements for high throughput and low latency. To ensure recomputability, records in the store are immutable and never overwritten or deleted. In addition the store is subject to a strict back up process to avoid data loss.

3.1 ELSA Get Requests and Bitemporality

Assuming that *ELSA* uses the timestamp an external system provides for storing and querying data, a significant problem for consistency and production arises: There is always a delay between the time when an event happens and when it becomes visible in an external system which is similar to the problem of application time skew described by Srivastava in [12]. This duration further increases if the external system doesn't offer push-based change notifications and *ELSA* has to pull the change history on its own. Even if the external system supports push-based change notifications, it may only send them intermittently which would cause get requests to *ELSA* to return different results for the same timestamp due to late updates from the external system.

For example, a car sends its current GPS position to its manufacturer at 9:00 a.m., who forwards the position to *ELSA* at 9:30 a.m.. At 9:15 a.m., a get request to *ELSA* wants to know the last position of the car five minutes ago (9:10 a.m.). The request will return no results. If the request for the last car position at 9:10 a.m. is sent again at 9:45 a.m., the result would be the stored value. Hence, the result of the first request is no longer recomputable after the late update has been incorporated in the *ELSA Store*.

To ensure recomputability, get request result consistency is necessary. To achieve this, the concept of bitemporality, as stated in Definition 3, is used.

Definition 3 (Bitemporality) A data record is bitemporal if it has two decoupled timestamps which distinguish between the event time t_e and the ingest time t_i.

Be $q(k_\text{q}, t_\text{E}, t_\text{I}) = r$ a get request to the *ELSA Store*, where k_q is the key to be queried. t_E is the maximum event timestamp and t_I the maximum ingest timestamp. The resulting r is an *ELSA Record* or empty, as defined in Definition 4.

Definition 4 (ELSA Record) An *ELSA Record* r is created from an external key-value operation, see Definition 2, and defined as $r = (k, t_\text{i}, o, t_\text{e}, v)$. Where k is the key of an external key-value operation and t_i is the time the key-value operation was ingested into *ELSA*. The type of the operation is given in o and can either be *insert* or *delete*. The event timestamp t_e is derived from the *insert* or *delete* operation. The value v is set for an *insert* operation or empty for a *delete* operation.

Our event timestamp t_e and ingest timestamp t_i are similar to concepts of temporal databases like the valid and transaction time as described by Ozsoyoglu et al. [10] and Jensen et al. [6]. Nonetheless, the valid and transaction times are defined as intervals while the event and ingest timestamps are singular values. However, intervals in our system are defined implicitly by newer versions of the same *ELSA Record*.

Let R_q be a set of *ELSA Records* for a certain external system

- whose keys are equal to k_q
- whose ingest times are smaller than t_I
- whose event times are smaller than t_E

The *ELSA* get request takes the record r with the maximum event time of R_q. If operation o of r is *delete*, the get has no result. Otherwise the *ELSA Record* r is the result of the get request.

Regarding the car manufacturer example, t_E is set to 9:00 a.m. and t_I is set to 9:30 a.m.. The *ELSA* get request has to use both timestamps to identify the correct value which will make both aforementioned queries two distinct queries. While the first request uses a maximum ingest timestamp of 9:15 a.m. the second one uses 9:45 a.m.. Thus, the first get will return no results while the second will return a GPS location.

A more complex example which contains delete operations is shown in Fig. 2.

During get q_1, the event time of *insert* i_2 is matching the query but the queries maximum ingest time is not. Hence,

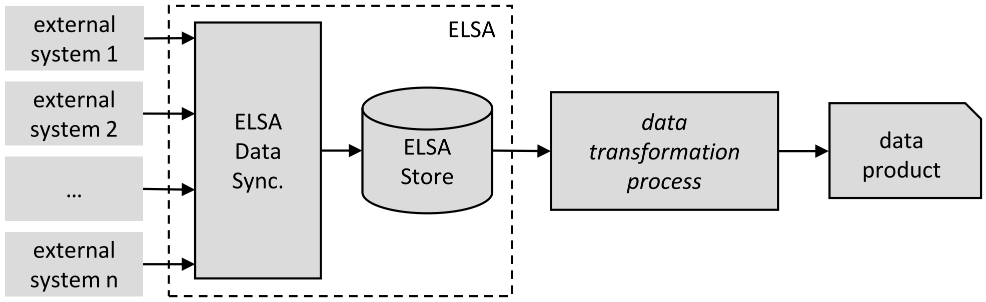

Fig. 3 The *ELSA* dataflow

the result of query q_1 is the record inserted by *insert* i_1. For request q_2 the event times of the *delete* and *insert* i_3 are too large and they are filtered although the ingest time requirements are met. This leads to the record of *insert* i_2 as result for the query. In the case of q_3, the result is i_3 since it has the largest event timestamp and the ingest time requirements are met.

In the case of mgm's customer the gap between event and ingest time varies a lot depending on the type of data, but the offset is mostly within days. Events are not transmitted in (near) real-time from the event source. However, in the future this gap may become smaller for the general case with the rise of more streaming based industrial applications but will never vanish.

Specific ingest times are not required in the regular case for mgm's customer. However, there are rare cases which require a specific ingest time, e. g. to recompute intermediate results for analytical purposes on the intermediate data which was incorporated into a previously computed final result of the data transformation process. The alternative to this would be to keep all intermediate results for all executed data transformation processes, which is impractical.

3.2 ELSA Data Synchronization and Store

Fig. 3 shows a more detailed view of *ELSA*. The *ELSA Data Synchronization* subscribes to all changes done in the external systems and writes them into a queue. This queue is a replicated queue which is able to handle peaks. An entry in the queue is an external key-value operation as specified in Definition 2. A write process pulls the external key-value operation from the queue and transforms it into an *ELSA Record* as defined in Definition 4, which is written into the *ELSA Store*.

The *ELSA Store* is a Big Table [2] like persistent, distributed, scalable, multi-version key-value data store. It is able to handle massive, parallel requests from data transformation processes. Furthermore, it is target of backup processes to ensure that no data is lost and recomputability stays intact even in the case of a system failure.

4 Automated Manual Action Application System

After the introduction of *ELSA*, our system architecture is able to fulfil most of the aforementioned requirements from Sect. 2. The last missing requirement refers to human interactions on intermediate results during quality checks. A quality check is an automated or manual process which ensures that functional requirements are met. If not, a domain expert has to manually adjust the result of a data transformation process in such a way that the requirements are met. We call this adjustments manual actions. Domain experts who make these manual actions are expensive and the process is time consuming. In the case of recomputation, we don't want to invest their precious resources in previously solved problems again. Hence, the architecture needs a component which can extract, store and apply manual actions resulting from quality checks. Therefore, we introduce *ANNA* (Automated Manual Action Application System) which is designed to fulfil these requirements. It consists of three elements:

- the *manual action definition*, the process which is executed after the requirements of a quality check were not met. It creates a manual action which is the result of the manual action done by the domain expert.
- the *ANNA Store* is the place where manual actions are stored.
- the *automated manual action application*, which describes the process of applying a stored manual action to the result of a data transformation process.

In the system architecture, *ANNA* encapsulates the data transformation process as shown in Fig. 4. Those additional elements have to be integrated into the data transformation process.

4.1 Data Transformation Process

A data transformation process describes the complete process for creating a data product. It is subdivided by several data transformation steps. To support the automated application of manual actions we have to integrate changes to the data transformation step workflow. This adapted work-

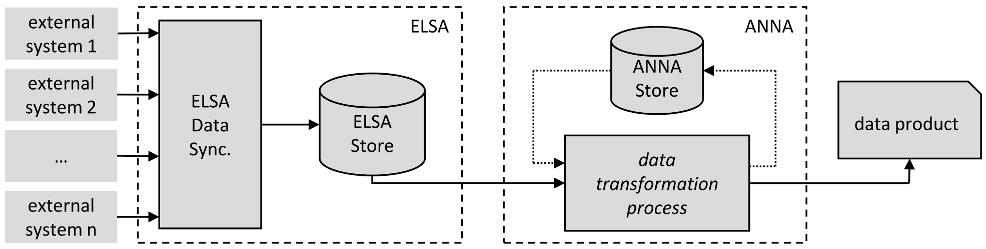

Fig. 4 The *ELSA* dataflow combined with *ANNA*

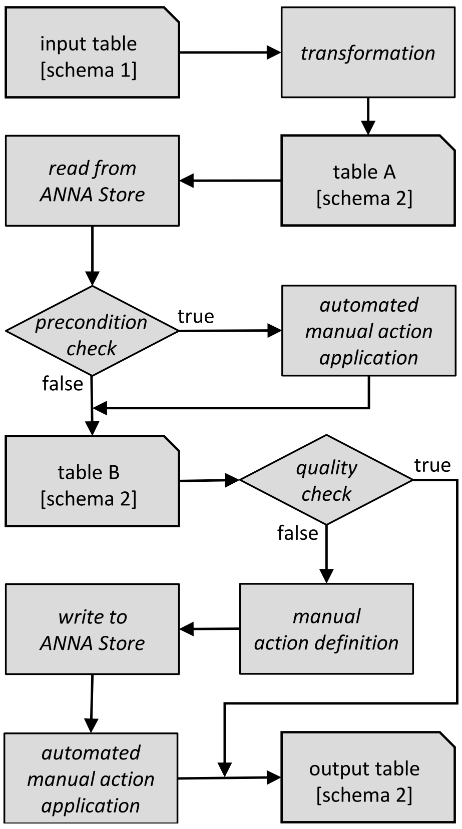

Fig. 5 The flowchart of the data transformation step in combination with *ANNA*

flow of a data transformation step is depicted in Fig. 5. The Figure shows that each data transformation step operates on data organized in tables using a specified schema as input. Each cell in the table must be identifiable by using a corresponding row and column id. The data transformation step contains the functional logic and may change the schema of the table. For the first run of a data transformation process we can skip the automated manual action application and table *A* is equivalent to table *B*. The intermediate table *B* undergoes quality checks which may lead to the need of cell adjustments.

4.2 Manual Action Definition

The process of defining an adjustment to a cell in the table is called manual action definition and shown in Fig. 5. Those operations are schema-preserving and based on pre-defined operators like set, add, subtract, multiply, divide and so on. During the definition of the manual action, the user has to define preconditions for the case of recomputation. After the manual action definition, the manual action is applied to the table cell. To support domain experts with the definition of these manual actions, the process is tool based and guides the users through each necessary step. It prompts the user to specify the above mentioned information:

- the precondition for this manual action based on the input table
- the cell to modify in the input table
- and the manual action from the set of predefined operations which modifies exactly one cell.

Furthermore the manual action is transformed into an *ANNA Record* as defined in Definition 5 and saved in the *ANNA Store*.

Definition 5 (ANNA Record) An *ANNA Record a* is created from a manual action and defined as tuple

$$a = (\mathrm{dtp}_{\mathrm{uid}}, \mathrm{dtp}_{\mathrm{r}}, \mathrm{dtp}_{\mathrm{s}}, \mathrm{tab}_{\mathrm{r}}, \mathrm{tab}_{\mathrm{c}}, t_{\mathrm{i}}, t_{\mathrm{e}}, s).$$

Where the data transformation process is uniquely identified by dtp_{uid}, its run number $\mathrm{dtp}_{\mathrm{r}}$ and the specific data transformation step $\mathrm{dtp}_{\mathrm{s}}$ within the data transformation process. The modified table cell is uniquely identified by the table row $\mathrm{tab}_{\mathrm{r}}$ and the table column $\mathrm{tab}_{\mathrm{c}}$. Furthermore an ingestion timestamp t_{i} and an event timestamp t_{e} for the creation of the manual action is necessary to fulfil recomputability requirements as described in Sect. 3.1. The script s contains the actual precondition and transformation of the manual action.

With the *ELSA Store*, we already have a store in our system architecture. To reduce the necessary technology stack which has to be maintained for the system architecture, we design the *ANNA Store* to be similar to the *ELSA Store*. With an appropriate key-value store, it will be possible that

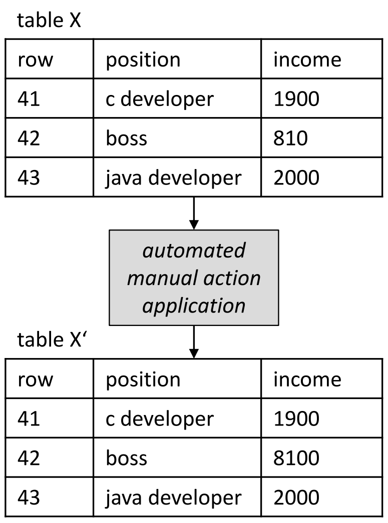

table X

row	position	income
41	c developer	1900
42	boss	810
43	java developer	2000

table X‘

row	position	income
41	c developer	1900
42	boss	8100
43	java developer	2000

Fig. 6 The Figure shows an example where the boss's salary in table X is determined as too low by the quality check. Therefore, a manual action was defined by a domain expert which adjusted the income accordingly. This manual action was then automatically applied during a recomputation on table X resulting in table X'

the data of *ANNA* is another namespace in the distributed key-value data store.

4.3 Automated Manual Action Application

The stored manual actions can now be applied to recomputation processes. Like it is done in *ELSA*, a recomputation with *ANNA* has to be annotated with a maximum *ANNA* ingestion timestamp t_{AI} and a maximum *ANNA* event timestamp t_{AE}. Before a recomputation run is executed, all applying adjustments are read from the *ANNA Store* and added to the corresponding data transformation step. In Fig. 5 the automated manual actions are applied to the table right after the previous transformations. Since our table may contain new data, it is possible that not all preconditions are met to execute the stored operations. In that case, the corresponding manual action is not executed. Instead it is possible that new manual actions are necessary in respond to the new data. For reproduction, it is important to store those manual actions in the *ANNA Store* as well.

An example for the automated manual action application is given in Fig. 6.

5 Implementation

ELSA and *ANNA* has been brought into production by mgm and its customer. In this section, some of the experienced pitfalls will be described and solved. For a better understanding, those pitfalls will be described on specific technologies. Therefore, the following explanations will be under the assumption that the software for distributed computing is Apache Hadoop[4] and the distributed database is Apache Accumulo which stores its data into the Hadoop Distributed File System (HDFS). Apache Accumulo is a BigTable clone and implements the data model as described by Chang et al. [2] and the Accumulo documentation [1].

5.1 Time-to-Consistency

In databases, there is a delay between the ingest time assigned by the store and the moment the value is written. To write a record, a database has to determine the ingest timestamp e. g. by using the system clock, and only then it is able to finally write it. This problem even increases in distributed databases because of network latency and communication for replication and distribution. This may lead to the situation where data with a certain ingest time t_{i} is written at a later time t_{y}, which breaks the recomputability.

The *time-to-consistency* t_{con} defines an upper bound for how long it may take from the determination of the ingest timestamp of a record till the record is written. A system fulfills the time-to-consistency if its writing processes at minimum use the current time t_{now} as ingest timestamp. Furthermore, read operations may only choose a maximum ingest timestamp t_{I} for their read requests which is at most $t_{\mathrm{now}} - t_{\mathrm{con}}$.

The concept of time-to-consistency is applied to all read operations in *ELSA* and *ANNA*. An example value for t_{con} can be:

$$t_{\mathrm{con}} = \text{write timeout}$$

This would be 60 seconds for an Apache Hadoop environment with the default write timeout value. However, this is only valid if the ingest timestamp is set automatically within the tablet server of Apache Accumulo. In the case of a failing write operation, this timestamp would be set again to a new value determined by the tablet server within the next write retry.

If for certain reasons (for example the usage of an external clock for ingest timestamps) the ingest timestamp has to be set during the writing process itself, before the data is send to Apache Accumulo, t_{con} can be:

$$t_{\mathrm{con}} = \text{write timeout} \cdot (\text{write retries} + 1) + \Delta$$

In this case Δ resembles the time needed between the different write attempts.

Even though we describe *time-to-consitency* as part of *ELSA* and *ANNA* it is a general concept which can be used

[4] https://hadoop.apache.org/

```
$result \leftarrow NULL$;
foreach $r \in R$ do
    if $r.t_e > q.t_E$ then
        if $result.o = insert$ then return $result$;
        else return $NULL$;
    if $r.t_i > q.t_I$ then
        continue with next $r$;
    else
        $result \leftarrow r$;
return $result$;
```

Fig. 7 Custom Server-Side Iterator Algorithm

Table 1 Example *ELSA Store* table instance ext

Record	Row ID	Col. Family	Version	Value
r	k	t_e	t_i	operation & v
r_1	x	5	10	insert & v_1
r_2	x	10	30	delete
r_3	x	12	20	insert & v_2
r_4	x	35	40	insert & v_3

with every database. For mgm's customer the time-to-consistency used varies betweeen 10 seconds and 10 minutes. It is as low as 10 seconds for transactional databases with low transaction timeouts or CP-type distributed databases like Apache Accumulo. In the case of relational databases with long-running transactions or processes where the database is inaccessible while it is built, e. g. in the case of file-based databases (RocksDB[5]) which need to be distributed to all cluster nodes before reading, 10 minutes are used.

5.2 Schema and Server-Side-Iterator

Table 1 shows the *ELSA* schema of record r for the distributed and sorted key-value data store Apache Accumulo.

Each external system has its own table named by an unique identifier for the external system. The Apache Accumulo row id is the key k of an *ELSA Record* r. The column family contains the event time of r and the version is set by Apache Accumulo and resembles t_i. Operation and value of r are encoded into the value field. For this version of the implementation the column qualifier is not used.

When a request is sent to Apache Accumulo, it by default returns the latest value of a record. Since *ELSA* uses a bitemporal approach for the get requests (see Sect. 3.1), this is not appropriate. Therefore, the version iterator had to be removed and the problem has been solved by using custom server-side iterators. This is possible since an iterator processes all records of the same key in ascending order by column family and column qualifier and in descending order by version. Even in the case that data is inserted out-of-order, like it is shown in Table 1, the algorithm returns the correct result.

For a given get request $q = (k, t_I, t_E)$ and store ext the server-side iterator iterates the records R of table ext with row-key equal to k.

The following examples describe the Algorithm (Fig. 7) used by the server-side iterator on Table 1. For a request $q_1 = (x, 15, 35)$ the iterator takes the following steps:

1. evaluate r_1: $5 < 15 \wedge 10 < 35 \rightarrow result = r_1$
2. evaluate r_2: $10 < 15 \wedge 30 < 35 \rightarrow result = r_2$
3. evaluate r_3: $12 < 15 \wedge 20 < 35 \rightarrow result = r_3$
4. evaluate r_4:
   $35 \geq 15 \wedge result.o = insert \rightarrow return\ result = r_3$

For example request $q_2 = (x, 11, 40)$ the iterator takes the following steps:

1. evaluate r_1: $5 < 11 \wedge 10 < 40 \rightarrow result = r_1$
2. evaluate r_2: $10 < 11 \wedge 30 < 40 \rightarrow result = r_2$
3. evaluate r_3:
   $12 \geq 11 \wedge result.o = delete \rightarrow return\ NULL$

During the last example request $q_3 = (x, 15, 15)$ the iterator takes the following steps:

1. evaluate r_1: $5 < 15 \wedge 10 < 15 \rightarrow result = r_1$
2. evaluate r_2: $10 < 15 \wedge 30 \geq 15 \rightarrow continue$
3. evaluate r_3: $12 < 15 \wedge 20 \geq 15 \rightarrow continue$
4. evaluate r_4:
   $35 \geq 15 \wedge result.o = insert \rightarrow return\ result = r_1$

5.3 ANNA Store

Table 2 shows the schema of the *ANNA Store* which is able to store *ANNA Records* as defined by Definition 5. The Apache Accumulo row id is a compound key of the data transformation process identification parameters $\mathrm{dtp_i}$, $\mathrm{dtp_r}$ and $\mathrm{dtp_s}$. This ensures fast prefix scans on the *ANNA Store* for a given data transformation process and its run number. The resulting set of *ANNA Records* contains all manual actions for each data transformation step. However, this format even ensures fast look ups for specific data transformation steps. The column family contains the table cell identification parameters $\mathrm{tab_r}$ and $\mathrm{tab_c}$. By making a prefix scan containing the unique data transformation process identification number $\mathrm{dtp_i}$ and the column family it is possible to get all changes made to a cell without taking into account in which run or step it was made. To ensure recomputability the rules for time-to-consistency as explained in Sect. 5.1 and the bitemporality from Definition 3 have to be applied. Therefore, the event timestamp t_e is stored in the column qualifier and the ingest timestamp t_i is stored as Apache

[5] http://rocksdb.org/

Table 2 The *ANNA Store* Schema and a stored *ANNA Record* (see Definition 5)

Row ID			Column Family		Column Qualifier	Version	Value
data transformation process			compound column family				
dtp$_i$	dtp$_r$	dtp$_s$	tab$_r$	tab$_c$	t_e	t_i	s
income report	1	boss	42	income	1496746276031	t_1	serialized script A
income report	1	worker	43	income	1496746289028	t_2	serialized script B

Accumulo timestamp similar to the *ELSA Store*. The value stores the serialized manual action definition script s of an *ANNA Record* which contains the transformation and its precondition. Those manual actions can be stored as JSON with a Base64 [7] encoded precondition and transformation field. The script language used to define the precondition and transformation could be any suitable script language. For this example we used javascript. Assuming a domain expert generates a manual action with the following data:

- `table.row = "42"`
- `table.column = "income"`
- t_{AE} = 1496746276031
- `data.transformation.process.identifier = "income report"`
- `data.transformation.run.number = "1"`
- `data.transformation.process.step = "boss"`
- precondition =

```
function preconditions(inTable) {
  if(inTable.get(42,"income") < 1000)
    return true;
  else
    return false;
}
```

- transformation =

```
function transformation(cellValue) {
  return cellValue * 10;
}
```

This would result in the following elements of an *ANNA Record*:

- dtp$_i$: `income report`
- dtp$_r$: `1`
- dtp$_s$: `boss`
- tab$_r$: `42`
- tab$_c$: `income`
- t_e: `1496746276031`
- t_i: t_1
- s:

{"precondition":"MS4gWW91IGhhdmUgZGVjb2
RlZCB0aGlzIEJBU0U2NCBjb2RlLg0KMi4gWW91I
GhhdmUgYSBnb29kIHNlbnNlIG9mIGh1bW9yLg==",
"transformation":"U2VuZCBhbiBlbWFpbCB0b
yB0aGUgYXV0aG9yczogbWF0dGhpYXMubWsua3Jp
Y2tlQGdtYWlsLmNvbSBhbmQgbWdyaW1tZXI0MkB
nbWFpbC5jb20sIHRoZXkgd2lsbCBiZSBoYXBweS
BhYm91dCBpdC4="}

This *ANNA Record* is stored in the ANNE Store as shown in Table 2 and can be applied as it was shown in Fig. 6.

6 Conclusion and Future Work

We have defined a system architecture composed of *ELSA* and *ANNA* which can reliably recompute data products even when data changes, external systems are used or human interactions are applied to intermediate results. We have managed this by keeping all versions of the data and the human interactions in a bitemporal and consistent way. In the former system, the database state to be used was not configurable and depended on the time a get request was sent. This led to unrecomputable results. In an *ELSA* get request the state to be used of the external system is configurable and defined by the ingest timestamp. Therefore, the *ELSA* get request allows flexible recomputations of data transformation processes.

Besides recomputability, the architecture has several other benefits. It is linearly scalable, has a low latency and can handle massive parallel requests by using distributed technologies like Apache Hadoop and Apache Accumulo. Furthermore, temporary connection issues regarding the external systems are hidden from the data transformation process. Hence, we can even execute data transformation processes while an external system is not accessible. In addition, there is no possibility of overwhelming the external system with distributed queries when scaling up the data transformation processes. Moreover, by using *ANNA* data transformation processes of mgm's customer which contains human interactions of domain experts are now recomputable. Formerly automated data transformation processes which recompute data products were blocked by this human interactions. Those manual actions are defined with rich tooling that maps the user interactions onto a predefined set of operations which can then be applied automatically as a part of a data transformation step during a recomputation. To bring this even further, we plan to investigate whether it is possible to reuse those human interactions in other data transformation processes.

Additional future work may consider the version of the software used by data transformation processes since old versions may no longer be available to recompute results. We believe this is quite challenging since used framework versions, the runtime environment and hardware may change significantly. However, it might be possible by the use of recomputable virtualized environments.

Acknowledgements This work was partly funded by the German Federal Ministry of Education and Research within the project Competence Center for Scalable Data Services and Solutions (ScaDS) Dresden/Leipzig (BMBF 01IS14014B) and Explicit Privacy-Preserving Host Intrusion Detection System EXPLOIDS (BMBF 16KIS0522K).

References

1. Accumulo A (2017) Accumulo design – data model. http://accumulo.apache.org/1.8/accumulo_user_manual.html#_accumulo_design. Accessed 27 July 2017
2. Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE (2008) Bigtable: A distributed storage system for structured data. ACM Trans Comput Syst 26(2):4
3. Corbett JC, Dean J, Epstein M, Fikes A, Frost C, Furman JJ, Ghemawat S, Gubarev A, Heiser C, Hochschild P et al (2013) Spanner: Google's globally distributed database. ACM Trans Comput Syst 31(3):8
4. Dragojević A, Narayanan D, Nightingale EB, Renzelmann M, Shamis A, Badam A, Castro M (2015) No compromises: distributed transactions with consistency, availability, and performance. In: Proceedings 25th Symposium on Operating Systems Principles, ACM, pp 54–70
5. Gray J, Reuter A (1992) Transaction processing: concepts and techniques. Elsevier, Amsterdam
6. Jensen CS, Soo MD, Snodgrass RT (1994) Unifying temporal data models via a conceptual model. Inf Syst 19(7):513–547
7. Josefsson S (2006) The base16, base32, and base64 data encodings. https://tools.ietf.org/html/rfc4648. Accessed 9 June 2017
8. Kulkarni K, Michels JE (2012) Temporal features in sql:2011. SIGMOD Rec 41(3):34–43. https://doi.org/10.1145/2380776.2380786
9. Lee J, Muehle M, May N, Faerber F, Sikka V, Plattner H, Krueger J, Grund M (2013) High-performance transaction processing in sap hana. IEEE Data Eng Bull 36(2):28–33
10. Ozsoyoglu G, Snodgrass RT (1995) Temporal and real-time databases: a survey. IEEE Trans Knowl Data Eng 7(4):513–532
11. Rahm E, Do HH (2000) Data cleaning: problems and current approaches. IEEE Data Eng Bull 23(4):3–13
12. Srivastava U, Widom J (2004) Flexible time management in data stream systems. In: Proceedings twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, ACM, pp 263–274