B. Mitschang et al. (Hrsg.): BTW 2017 – Workshopband,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 227

# Preserving Recomputability of Results from Big Data Transformation Workflows

Matthias Kricke [1], Martin Grimmer [1], Michael Schmeißer [2]

Matthias Kricke, Martin Grimmer und Michael Schmeißer

**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. When external systems are used or data changes over time this becomes even more challenging. 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.

**Keywords:** BigData, recomputability, bitemporality, time-to-consistency

## 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. 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[OS95]. Temporal features have also been incorporated in the SQL:2011 standard[KM12]. Problems like concurrency control[GR92] have been solved for modern, distributed systems of e.g. Microsoft[Dr15], Google[Co13] and SAP[Le13]. However, those systems either can't be used on-premises or have high licensing costs. In addition, recomputability of data products in big data systems with dependencies to external systems 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

[1] Universität Leipzig, Fakultät für Mathematik und Informatik, Institut für Informatik, Augustusplatz 10, 04109 Leipzig, {kricke, grimmer}@informatik.uni-leipzig.de
[2] mgm technology partners GmbH, Neumarkt 2, 04109 Leipzig, michael.schmeisser@mgm-tp.com

Apache Accumulo[3], Apache HBase[4] or Apache Cassandra[5]. Nonetheless, those stores do not offer a proper solution for recomputability.

In this paper, we present ELSA (**E**xterna**L** **S**ystem **A**daptor), a big data system architecture which ensures recomputability of data products with dependencies to external systems by using a variant of the concept of bitemporality[JSS94]. 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[RD00]. Those processes are incorporating data from external systems. This leads to the demand to relieve the external systems from high-frequent distributed requests. A solution has to be linearly scalable with low to none 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 Section 2 a decoupling of external systems and the data transformation process is necessary. The **E**xtern**l** **S**ystem **A**daptor (ELSA) is the

[3] https://accumulo.apache.org/
[4] https://hbase.apache.org/
[5] https://cassandra.apache.org/

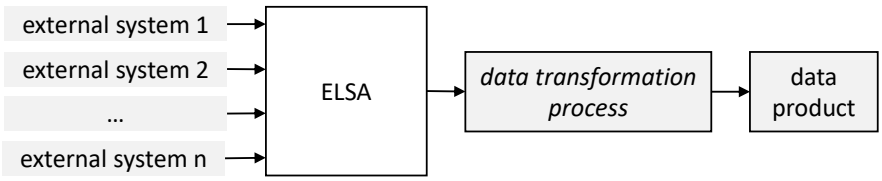


Fig. 1: Decoupling of external systems and the data transformation process with ELSA.

junction shown in Figure 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*

$$insert = (k, t_e, v).$$

*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*

$$delete = (k, t_e)$$

*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 $t_3 = (a, 4, v'')$, a delete operation defined as $delete = (a, 3)$ would remove $t_1$ and $t_2$ in the external system and retain $t_3$. However, to preserve recomputability, in ELSA data is not deleted but invalidated.

To fulfill 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 are now using the ELSA Store instead of the external systems, which removes the logic of handling external systems from them. Moreover, the ELSA Data Synchronization highly reduces the throughput and latency requirements for an external system since data is only read once by it. Since those requirements now has to be handled by the store it is an internal, distributed, scalable, multi-version key-value data store. To ensure recomputability, records in the store are immutable and never overwritten or deleted. In addition the store is underlying a strict back up process to avoid data loss.

### 3.1 ELSA Get Requests and Bitemporality

Assuming that ELSA is using 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 is visible in an external system. This time further increases when ELSA gets the change history. An external system may only send unregular updates to ELSA. In that case two get requests for the same record with the same timestamps may lead to different results.

For example a car sends its current GPS position to its manufacturer at 9:00 a.m., who is sending 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 send again at 9:45 a.m. the result would be the stored value. Hence, the result of the same request is inconsistent and thus not recomputable.

To ensure recomputability, get request result consistency is necessary. For achieving 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 $q(k_q, t_E, t_I) = r$ 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 $r = (k, t_i, o, t_e, v)$. 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 could 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.*

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 the both aforementioned queries two distinct queries. While the first request is using a maximum ingest timestamp of 9:15 a.m. the second one is using 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 Figure 2.

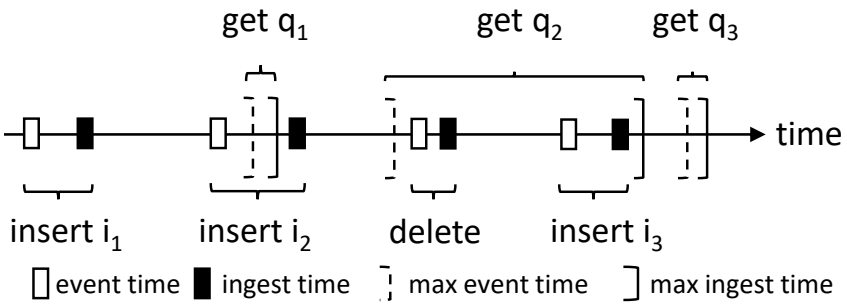


Fig. 2: All data versions of a record with different ingest and event times with several get requests.

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

### 3.2 ELSA Data Synchronization & Store

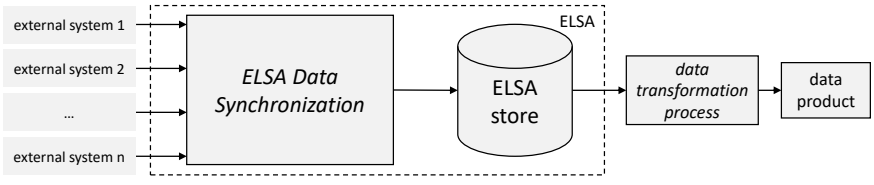


Fig. 3: The ELSA dataflow.

Figure 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 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[Ch08] 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 ensures that no data is lost and recomputability stays intact even in the case of a system failure.

## 4 Implementation

ELSA was brought into production by mgm and its customer. In this section some of the pitfalls will be described and solved. For a better understanding those pitfalls will be described on specific technologies. Therefore the following explanations will be done by assuming that the software for distributed computing is Apache Hadoop[6] and the distributed database is Apache Accumulo which stores its data into the Hadoop Distributed File System (HDFS).

### 4.1 Time-to-Consistency

In databases there is a small delay between the ingest time given 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 $t_{now} - t_{con}$.

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

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

In an Apache Hadoop environment with configured write timeout of 30 seconds and a maximum of 3 write retries $t_{con} = 120s$.

### 4.2 Schema and Server-Site-Iterator

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

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

| Record<br>$r$ | Row-Key<br>$k$ | Column Family<br>External Store | Column Qualifier<br>$t_e$ | Version<br>$t_i$ | Value<br>operation & $v$ |
|---|---|---|---|---|---|
| $r_1$ | $x$ | $ext_1$ | 5 | 10 | insert & $v_1$ |
| $r_2$ | $x$ | $ext_1$ | 10 | 30 | delete |
| $r_3$ | $x$ | $ext_1$ | 12 | 20 | insert & $v_2$ |
| $r_4$ | $x$ | $ext_1$ | 35 | 40 | insert & $v_3$ |

Tab. 1: Example ELSA Store table instance.

The row-key is the key $k$ of an ELSA record $r$. The data of external systems is stored in the same Apache Accumulo table but distinguished by the column family. However, by using locality groups, only data for a given external system is considered during a get request. The column qualifier 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.

When a request is send to Apache Accumulo, by default it returns the latest value of a record. Since ELSA is using a bitemporal approach for the get requests (see Section 3.2) this is not feasible. Therefore, the version iterator had to be removed and the problem has been solved by using custom server-site 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 $R$ with row-key equal to $k$ and column family equal to *ext*.

**Algorithm 1:** Costum Server-Side Iterator Algorithm

```
result ← NULL;
foreach r ∈ 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 ← r;
return result;
```

The following examples are describing the Algorithm 1 used by the server-site iterator on Table 1. For a request $q_1 = (x, 15, 35)$ the iterator is undergoing 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 is undergoing 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 is undergoing 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 Conclusion & Future Work

We have defined the ELSA architecture which can reliably recompute data products even when data changes or external systems are used. We have managed this by keeping all versions of the data in a bitemporal and consistent way. In the former system the used state of the database was not configurable and depended on the time a get request was sent. Which led to unrecomputable results. In an ELSA get request the used state of the 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 lineary 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 from the data transformation processes.

Some data transformation processes of mgm's customer contain manual interactions of business experts. In this case, automated data transformation processes which are recomputing data products are blocked by this manual interactions. To make this case non-blocking and recomputable, we would like to enhance ELSA by a manual action registry. In this case, manual interactions are done using rich tooling and are required to result in a data transformation process themselves. One can design a system which automatically applies

those manual interactions while recomputing a data product. To bring this even further, we plan to investigate whether it is possible to reuse those manual 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.

## 6 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

[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.

[Co13] Corbett, James C; Dean, Jeffrey; Epstein, Michael; Fikes, Andrew; Frost, Christopher; Furman, Jeffrey John; Ghemawat, Sanjay; Gubarev, Andrey; Heiser, Christopher; Hochschild, Peter et al.: Spanner: Google's globally distributed database. ACM Transactions on Computer Systems (TOCS), 31(3):8, 2013.

[Dr15] Dragojević, Aleksandar; Narayanan, Dushyanth; Nightingale, Edmund B; Renzelmann, Matthew; Shamis, Alex; Badam, Anirudh; Castro, Miguel: No compromises: distributed transactions with consistency, availability, and performance. In: Proceedings of the 25th Symposium on Operating Systems Principles. ACM, pp. 54–70, 2015.

[GR92] Gray, Jim; Reuter, Andreas: Transaction processing: concepts and techniques. Elsevier, 1992.

[JSS94] Jensen, Christian S; Soo, Michael D; Snodgrass, Richard T: Unifying temporal data models via a conceptual model. Information Systems, 19(7):513–547, 1994.

[KM12] Kulkarni, Krishna; Michels, Jan-Eike: Temporal Features in SQL:2011. SIGMOD Rec., 41(3):34–43, October 2012.

[Le13] Lee, Juchang; Muehle, Michael; May, Norman; Faerber, Franz; Sikka, Vishal; Plattner, Hasso; Krueger, Jens; Grund, Martin: High-Performance Transaction Processing in SAP HANA. IEEE Data Eng. Bull., 36(2):28–33, 2013.

[OS95] Ozsoyoglu, Gultekin; Snodgrass, Richard T: Temporal and real-time databases: A survey. IEEE Transactions on Knowledge and Data Engineering, 7(4):513–532, 1995.

[RD00] Rahm, Erhard; Do, Hong Hai: Data Cleaning: Problems and Current Approaches. 2000.