ITC 1/47
Journal of Information Technology and Control
Vol. 47 / No. 1 / 2018
pp. 5-25
DOI 10.5755/j01.itc.47.1.18171
© Kaunas University of Technology

A Time-Constrained Algorithm for Integration Testing in a Data Warehouse Environment

Received 2017/05/11 | Accepted after revision 2018/01/22

http://dx.doi.org/10.5755/j01.itc.47.1.18171

A Time-Constrained Algorithm for Integration Testing in a Data Warehouse Environment

Ljiljana Brkić, Igor Mekterović

University of Zagreb, Faculty of Electrical Engineering and Computing, Unska 3, 10000 Zagreb, Croatia,
e-mail: ljiljana.brkic@fer.hr, igor.mekterovic@fer.hr

Corresponding author: ljiljana.brkic@fer.hr

A data warehouse should be tested for data quality on regular basis, preferably as a part of each ETL cycle. That way, a certain degree of confidence in the data warehouse reports can be achieved, and it is generally more likely to timely correct potential data errors. In this paper, we present an algorithm primarily intended for integration testing in the data warehouse environment, though more widely applicable. It is a generic, time-constrained, metadata driven algorithm that compares large database tables in order to attain the best global overview of the data set's differences in a given time frame. When there is not enough time available, the algorithm is capable of producing coarse, less precise estimates of all data sets differences, and if allowed enough time, the algorithm will pinpoint exact differences. This paper presents the algorithm in detail, presents algorithm evaluation on the data of a real project and TPC-H data set, and comments on its usability. The tests show that the algorithm outperforms the relational engine when the percentage of differences in the database is relatively small, which is typical for data warehouse ETL environments.

KEYWORDS: Data Warehouse Testing, ETL, Integration Testing, Data Quality.

1. Introduction

Data quality is a key factor in data warehouse (DW) and business intelligence solutions. Continuous testing of a DW can provide a solid assessment of the data quality. DW testing process should be implemented at very early stages of DW system development, leading to early error detection and correction which will, in turn, increase DW's credibility and decrease operational costs in the long run.

DW testing is closely related to the quality of stored data. Data quality issues in DW environment are studied in [3], [20]. Typically, quality of data delivered to users is described and evaluated using quality attributes [38]. These attributes are called data quality dimensions. Each dimension covers a specific aspect of quality. For quantitative assessment of quality dimensions, it is necessary to define metrics and measurement methods. Researchers have recognized the importance of this issue and many research papers (some of which are [13], [18], [21], [30], [38]) deal with methods of measuring or quantifying dimensions of data quality.

Apart from the source data, the quality of DW data is also affected by the ETL process used to integrate and transfer the data. Faulty ETL process, whether because of the faulty logic, inappropriate data refreshment strategy or just plain programming errors, can cause data not to be transferred to the DW or not to be transferred in a timely fashion. With that in mind, it is important to assess how accurate and complete data are: whether all required data from the data sources are extracted, stored in the staging area, transformed and subsequently loaded into the DW and whether the data in the DW are accurate with regards to the source data. We adopt the **accuracy** definition from [3] "Accuracy is defined as the closeness between a value *v* and a value *v'*, considered as the correct representation of the real-life phenomenon that *v* aims to represent" and **completeness** definition from [2]: "Presence of all defined content at both data element and data set levels." The assessment of accuracy and completeness of the data in a DW imposes comparing large data sets which is at the core of the DW's integration testing [24].

This paper is concerned with those very issues – we present an algorithm for integration testing of accuracy and completeness of the DW data in the sense of the aforementioned definitions. Integration testing is an approach to software testing where software components are combined and tested as a group, with the purpose of ensuring that interacting components or subsystems interface correctly with one another. Integration testing in DW environment usually only includes testing the ETL application, which comprises of numerous packages. ETL packages are tested by examining data at the endpoints (input and output) of those packages. In DW environment, we can identify three typical subsystems that ETL application deals with: data source(s), data staging area and DW production tables (typically dimension and fact tables). We perform integration testing by comparing various corresponding data sets from those three sources using the proposed TCFC (Time-Constrained Fragment and Compare) algorithm. The algorithm is generic and widely applicable, not limited to DW environment. It provides an overview of the differences between two data sets depending on the assigned time frame, from performing shallow comparisons to pinpointing exact differing tuples. We present the algorithm in detail, comment on its features, evaluate it on data of a real project as well as on mock data and comment on the results.

2. Motivation

The primary objective of the integration testing and TCFC algorithm presented in this paper is to get a global overview of the data quality in the DW with a focus on accuracy and completeness measures. These measures are considered with respect to the source systems, as we assume that the data in the source system are accurate and complete. In this generic test integration scenario, involving source, staging and DW subsystems, testing should provide answers to the following questions:

a whether all required data from the data sources been extracted and transferred to the staging area (completeness), and whether staging area attribute values are equal to corresponding values in source systems (accuracy);

b whether all required data from the staging area been transferred to the DW (completeness), and whether corresponding attribute values are equal (accuracy).

To answer the first question, the data from the data sources must be compared to the data stored in the staging area, where they are stored in the identical or similar schemas. To answer the second question, data from the staging area must be compared to the data in the DW, where they may be stored in different (though mappable) structures, e.g., dimensional model. In both cases, the problem boils down to comparing two sets of database tables having identical or different but mappable schemas to find missing or excess tuples on either side, and/or matching tuples with different values of non-key attributes.

Relational database engines are a natural solution to that problem, since they are highly optimized for set operations on data: Two tables can be compared for differences with a single SQL statement. This approach, however, which will be referred to as **reference implementation** hereafter, has two serious drawbacks that motivated our research:

3 It is not possible to span a single SQL query across heterogeneous platforms. That is a common scenario in DW environment, especially between source systems and staging area. To compare tables from different environments, table from one server would have to be transferred to a (temporary) table on the other server or some sort of database integration software would have to be used that would abstract that operation. Either way, the very operation of moving the data from one DBMS to another is subject to error. Transfer errors can occur for various reasons. For instance, IBM Informix's DATE type has a wider DATE range than SQL Server's, and rows with such dates simply cannot be inserted into the SQL Server table having the exact same schema.

4 It is an all-or-nothing approach. It is not possible to perform a "shallow comparison" – a comparison that would take much less time to execute but would not be completely accurate (i.e. detect all differences in all tables) and/or precise (i.e. pinpoint the exact tuples causing differences). All rows from one table are compared to all rows from the other table and in the case when there is a substantial number of rows (tens of millions and more), such a comparison can also take a substantial resources to execute. A large table comparison could block all others and take up all available time. In other words, such queries cannot be appropriately **time-managed** to fit into a given time frame. Time managing queries is very important because testing procedures have to fit into the ETL's acceptable time frame. It is unacceptable to query the data sources at arbitrary times, inflicting additional burden on the production systems.

Sometimes, when comparing two tables via a single SQL statement, it is handy to use a hash function (e.g., MD5) to produce the checksum of the tuple. This shortens the SQL statement, but, in general, calculating checksums additionally slows the comparison and is not suitable for large data sets, especially if checksums cannot be stored (e.g., ETL has read only access to source systems). Checksums are applicable to TCFC with similar properties, as commented in Section 3.1.

In terms of complexity, comparing two data sets is O(n^2) complex, where every record from the first set has to be compared with records from the second set. Another valid approach, used by some commercial tools (e.g., SQL Data Examiner [35]), is to fetch records sorted by the primary key from the databases and perform a less complex merge-sort style comparison; however – it must be taken into account that sorting data at the sources incurs additional costs. Both approaches suffer from the 2nd drawback described above: they are prone to blocking the entire comparison when large tables are examined. In addition, such approaches rely heavily on the client's resources (disk and memory) as both sets must be retrieved in their entirety. Time-managing testing (queries) is what makes this problem difficult, and, to the best of our knowledge, has not been addressed in the literature so far (see the section "Related Work" for more).

This motivated us to develop an algorithm that will surpass these limitations, yet remain comparable to the reference implementation in terms of speed. Since the expected number of differences between the tables is relatively small when compared to the total number of rows, we formulate the TCFC algorithm requirements as follows:

_ When applied to large data sets with a small number of differences, the algorithm should be comparable in terms of speed with the reference implementation. One could argue that having 10k or 100k faulty tuples is "the same" because such a large number of errors is either a sign of systemic error or an unmanageable number of data errors that cannot be manually corrected. Therefore, the algorithm should work well for a manageable number of differing rows, hundreds or thousands of errors, not millions.

_ Must be suitable for comparing data across heterogeneous platforms.

_ Must be time manageable, to fit into a configured time frame, at the expense of precision. In other words, it must be able to perform "shallow comparisons" and, consequently, has to be configurable.

The idea is to leverage the relational engine by breaking down the exhausting reference query into many faster queries that search for potential errors in a greedy fashion. This is done by subsequently breaking data into horizontal fragments and comparing fragment counts (and/or other aggregated values) of corresponding fragments. As the algorithm progresses, fragmentation filters are refined, so that the fragments become smaller. Fragments where inequalities are detected are examined with higher priority. Eventually, if so configured and if time allows it, the comparison could be brought to the primary key level, thus yielding the exact missing or excess tuples.

When designing the TCFC algorithm, we were guided by the thought that, given a limited time, it is better to acquire a possibly not completely accurate overview of **all** tables rather than an accurate comparison of **some** tables, while leaving others not examined. We denote this approach as “good enough global overview”. In other words, if there is a large set of table pairs with only several containing differences, it is considered better to report that there are *some differences* in all of the pairs actually containing differences, than to pinpoint exact differences in just a few of them, and leave the rest of the tables unexamined. It is a position that we have attained after participating in few real world projects and, though it might not be the best strategy for all applications, we believe that it is the best one in the most of DW ETL scenarios.

3. TCFC Algorithm

The TCFC algorithm is designed to be as generic as possible and to work on heterogeneous platforms and with large data sets. To apply the proposed algorithm to a DW, the following conditions must be met:

_ Data sources are relational databases, since the algorithm leverages relational engine. Other sources of data (text files, spreadsheets or other office documents, XML files, etc.) are not supported. As a workaround to this limitation, data from other sources can be processed and stored in a relational database (“piped through”).

_ Each record in the destination table corresponds to exactly one record in the source table. This requirement can be worked around by adding an additional metadata layer to describe the acceptable differences. For instance, with slowly changing dimensions, the number of regular “duplicates” can be kept. Furthermore, a generic system of human reviewing and (dis)approving differences could be put in place, where an analyst would mark the correct differing tuples and they would be taken into account in the future comparisons. We do not describe such system in more detail here, though. As a side note, our real-world project use-case required for a 100% match between the source and DW (students’ exams, year and course enrollments, etc.).

_ Data lineage [4-5] must be established, there must be a way to determine the source for each tuple in data warehouse tables. To do so, it might be necessary to make minor modifications to existing ETL procedures. The procedure for establishing data lineage in existing systems, which is used in this paper, is described in detail in [26].

In the following two sections, we formally describe the table-level part of the TCFC algorithm for comparing tables with identical and non-identical schemas, then present the overall algorithm in a pseudo code, provide a running example, present and discuss the performance testing and the associated results.

3.1. Determining Inequalities in the Content of Tables Having Identical Schemas

The algorithm works by fragmenting tables according to a fragmentation set, then one or more aggregate functions are evaluated upon each fragment, and finally, the results from the corresponding fragments are compared. Any differences, if found, indicate not only that the contents of r and s are different, but also point out to the group of tuples, i.e. fragment of the relation to which the problem pertains. What follows is a formal description of the stated.

Let r and s be the tables (relations) having the schema R. Let r denote a relation with schema $R = \{A_1, \ldots, A_N\}$. Domain for attribute A is denoted by $dom(A)$. Let X be a (possibly empty) subset of R. Let t denote a tuple. Let $t[X]$ denote an X-value of t, i.e. tuple t restricted to X.

Let $\mathcal{E}(r, X)$ denote the equivalence relation on r derived from equivalency of tuple’s X-values. We say that tuples $t_1, t_2 \in r$ are equivalent under $\mathcal{E}(r, X)$ iff the tuples have equal X-values, i.e. $(t_1, t_2) \in \mathcal{E}(r, X) \iff t_1[X] = t_2[X]$. It should be noted that if $X = \emptyset$ then $t_1[X] = t_2[X]$ for each $(t_1, t_2) \in r$, rendering all tuples in r equivalent under $\mathcal{E}(r,X)$.

Different X-values of r correspond with X-values of tuples in $\pi_X(r)$ (projection on the attributes contained in X). Each tuple $v \in \pi_X(r)$ unambiguously identifies an equivalence class, i.e. fragment $\mathcal{F}(r,X,v) = \{t \in r \mid t[X] = v\}$.

Consequently, equivalence relation $\mathcal{E}(r,X)$ partitions relation r into the set of fragments $\overline{\mathcal{F}}(r,X) = \{\mathcal{F}(r,X,v) \mid v \in \pi_X(r)\}$. Because the set X determines the fragmentation strategy of the r, hereinafter we will refer to the set X as to the fragmentation set. In the specific case when fragmentation set X is empty, $\pi_X(r)$ produces a single-tuple zero-degree relation, with the effect that $\mathcal{E}(r,X)$ partitions r into exactly one fragment which is equal to r.

Let $\mathcal{AF}$ be the set of available aggregate functions, where $a\!f$ denotes an aggregate function (e.g., count, sum, max, min). Let $\mathcal{B}$ be the set of arbitrary attribute names $B_1, B_2, \ldots$, subjected to the constraint that none of the attribute names appears in X. Relation $\mathcal{A}_{R,X}$ is a set of triplets $(a\!f, A, B)$ which relates aggregate functions from $\mathcal{AF}$, attributes from $R \setminus X$ and attribute names from $\mathcal{B}$ in the following way: if $(a\!f, A, B) \in \mathcal{A}_{R,X}$, then aggregate function $a\!f$ is evaluated for attribute A and the result is named B. Henceforth, we will refer to the relation $\mathcal{A}_{R,X}$ as the aggregation set.

Aggregate function $a\!f$ must be applicable to the domain of the attribute A. More than one aggregate function $a\!f$ can be applied upon each attribute A and each $B \in \mathcal{B}$ appears exactly in one triplet. Formally,

$$\mathcal{A}_{R,X} = \{(a\!f, A, B) \mid (a\!f, A, B) \in \mathcal{AF} \times (R \setminus X) \times \mathcal{B} \wedge a\!f \text{ is applicable to } dom(A)\}$$

$$\forall (a\!f_i, A_i, B_i), (a\!f_j, A_j, B_j) \in \mathcal{A}_{R,X}: B_i = B_j \Leftrightarrow a\!f_i = a\!f_j \wedge A_i = A_j$$

Designated aggregate functions from $\mathcal{A}_{R,X}$ are applied upon fragment $\mathcal{F}(r,X,v)$, the results of aggregate functions are renamed and concatenated to tuple v. The overall result of the operation is a single-tuple relation $q(r,X,\mathcal{A}_{R,X},v)$ with schema $X \cup \{B_1, B_2, \ldots B_n\}$, where $n = card(\mathcal{A}_{R,X})$:

$$q(r,X,\mathcal{A}_{R,X},v) = v \,^\wedge\, \rho_{B \leftarrow a\!f(A)} \left(\mathcal{G}_{\{a\!f(A) \mid (a\!f, A, B) \in \mathcal{A}_{R,X}\}} \mathcal{F}(r,X,v) \right).$$

Applying the aggregate functions upon each fragment, $\mathcal{F}(r,X,v) \in \overline{\mathcal{F}}(r,X)$ would yield a set of relations $q(r,X,\mathcal{A}_{R,X},v)$, exactly one relation per each $v \in \pi_X(r)$. Union of these relations, denoted as $q(r,X,\mathcal{A}_{R,X})$, can be effectively evaluated with relational algebra grouping operator:

$$q(r,X,\mathcal{A}_{R,X}) = \bigcup_{v \in \boldsymbol{\pi}_X(r)} q(r,X,\mathcal{A}_{R,X},v) = \rho_{B \leftarrow a\!f(A)} \left({}_X\mathcal{G}_{\{a\!f(A) \mid (a\!f, A, B) \in \mathcal{A}_{R,X}\}} r \right).$$

Given the relation schema R, relations $r(R)$ and $s(R)$, beforehand determined sets X and $\mathcal{A}_{R,X}$, one can easily evaluate $q(r,X,\mathcal{A}_{R,X})$ and $q(s,X,\mathcal{A}_{R,X})$.

For instance, only consider the relation "exam" in the left part of the Figure1, and suppose that we want to compare it with the identical-schema relation in another database. One possible fragmentation setup could be:

r(R)=exam(exam_date, student_id, course_id, has_passed)

s(R)=exam(exam_date, student_id, course_id, has_passed)

X={exam_date,student_id,course_id}

$$\mathcal{A}_{R,X} = \left\{ \begin{matrix} (sum, has\_passed, sumPassed), \\ (avg, has\_passed, avgPassed), \\ (count, has\_passed, recCount) \end{matrix} \right\}.$$

Tuples $t_r \in q(r,X,\mathcal{A}_{R,X})$ and $t_s \in q(s,X,\mathcal{A}_{R,X})$, where $t_r[X] = t_s[X] = v$ contain the results of aggregate functions applied on fragments $\mathcal{F}(r,X,v)$ and $\mathcal{F}(s,X,v)$, respectively. The inequality of the tuples implies the difference between the corresponding fragments, i.e. $t_r \neq t_s \Rightarrow \mathcal{F}(r,X,v) \neq \mathcal{F}(s,X,v)$, which leads to the conclusion that $q(r,X,\mathcal{A}_{R,X},v) \neq q(s,X,\mathcal{A}_{R,X},v) \Rightarrow r \neq s$.

Actually, any difference between $q(r,X,\mathcal{A}_{R,X})$ and $q(s,X,\mathcal{A}_{R,X})$ implies the inequality of r and s. Unfortunately, the inverse is not true, i.e. even when $q(r,X,\mathcal{A}_{R,X}) = q(s,X,\mathcal{A}_{R,X})$, it is still possible that r and s are not equal. This happens in very rare cases when differences of attribute values cumulatively nullify each other during aggregate function evaluation [26]. Opportunely, already low probability of such an event can be further reduced to the acceptable level by applying larger set of aggregate functions. This is the reason why the equivalence $q(r,X,\mathcal{A}_{R,X}) = q(s,X,\mathcal{A}_{R,X}) \Leftrightarrow r = s$, although not strictly correct, will be considered as adequate for practical purpose of comparing fragments. As a side note, an interesting approach that would also almost guarantee the equivalence would be to use a single aggregate function that aggregates tuple hashes. Such an aggregate function should be commutative, because ordering tuples would incur additional costs. For instance, SQL Server provides CHECKSUM_AGG function (which, though undocumented, we suspect is a simple XOR function) that can be used to that purpose. Similar remarks apply here as for the reference implementation – this is not suitable for large tables because calculating hashes on the fly is costly, and storing and maintaining additional hash data is often not possible, especially at source data systems.

The reason why the aggregate functions are applied only upon attributes in $R \setminus X$ is straightforward. Applying aggregate functions upon an attribute from X would be unproductive because X-values of tuples in corresponding fragments of r and s are identical by definition, so are the results of aggregate functions.

The number of fragments in $\overline{\mathcal{F}}(r, X)$ depends on the contents of the fragmentation set X. Generally, increasing the cardinality of X, increases the cardinality of $\pi_X(r)$, which in turn increases the number of tuples in $q(r, X, \mathcal{A}_{R,X})$, hence decreasing the average number of tuples per fragment. There is a trade-off in selection of attributes for the fragmentation set X. Using larger fragmentation set provides more precise determination of relation's subset which is the source of differences, but simultaneously degrades the performance due to increased number of groups for which aggregate functions have to be evaluated and their results compared. On the other hand, the contents of the aggregation set $\mathcal{A}_{R,X}$ do not significantly affect the performance, because the cost of aggregate function evaluation is negligible compared with the cost of grouping operation. Obviously, the main issue is to appropriately determine the content of the fragmentation set. The two extreme cases are: to use empty fragmentation set, which will produce exactly one fragment, or, to use one of the keys or superkeys for r, which will produce altogether $card(r)$ single-tuple fragments.

Taking into account the aforementioned trade-off and presuming that only a relatively small number of pairs of relations is expected to be actually different, we concluded that the process of comparing relations should start with the empty or nearly empty fragmentation set X_1. If $q(r, X_1, \mathcal{A}_{R,X_1}) = q(s, X_1, \mathcal{A}_{R,X_1})$, then the pair can be left out of further inspection. Otherwise, in order to determine the group of tuples incurring differences more precisely, procedure can be iteratively carried out. In each step of the procedure, fragmentation set is augmented with additional attributes, thus increasing the number of fragments and decreasing the fragment's tuple count. The process is repeated until maximal fragmentation set (with all intended attributes) has been inspected or allotted time frame has expired.

In order to carry out the described procedure, the following information has to be defined (and stored in a metadata repository) for each pair of relations $r(R)$ and $s(R)$:

1 Nonempty list of fragmentation sets, denoted as $listX_R$:

$listX_R = \langle X_1, X_2, \dots, X_m \rangle$,

where $X_1 \subset X_2 \subset \cdots \subset X_m \subseteq R$.

X_1 can be an empty set. Only the last fragmentation set in the list is allowed (but not required) to be a key or superkey of the relation r, because grouping by the key or superkey of the relation is the identity operator. Fragmentation sets should also fulfil the constraint that none of antecedent fragmentation sets functionally determines its descendants in the $listX_R$, i.e. $\forall X_i, X_j \in listX_R, i < j: X_i \nrightarrow X_j$. The latter constraint is quite easy to justify: If tuples t_1 and t_2 pertain to the fragment $\mathcal{F}(r, X_i, v)$, then $t_1[X_i] = t_2[X_i]$. If $X_i \rightarrow X_j$, then $t_1[X_i] = t_2[X_i] \Rightarrow t_1[X_j] = t_2[X_j]$, making t_1 and t_2 members of $\mathcal{F}(r, X_j, v)$ as well. As fragmenting sets X_i and X_j produce the same set of fragments, either X_i or X_j is superfluous in $listX_R$.

2 Nonempty list of aggregation sets, denoted as $list\mathcal{A}_R$, whose members correspond to the members of $listX_R$:

$$list\mathcal{A}_R = \langle \mathcal{A}_{R,X_1}, \mathcal{A}_{R,X_2}, \dots, \mathcal{A}_{R,X_m} \rangle.$$

Building on the previous example, these values could be:

$$listX_R = \left\langle \begin{array}{c} \{\}, \\ \{course\_id\}, \\ \{course\_id, student\_id\}, \\ \{course\_id, student\_id, exam\_date\} \end{array} \right\rangle$$

$list\mathcal{A}_R = \langle \mathcal{A}_{R,X}, \mathcal{A}_{R,X}, \mathcal{A}_{R,X}, \mathcal{A}_{R,X} \rangle$.

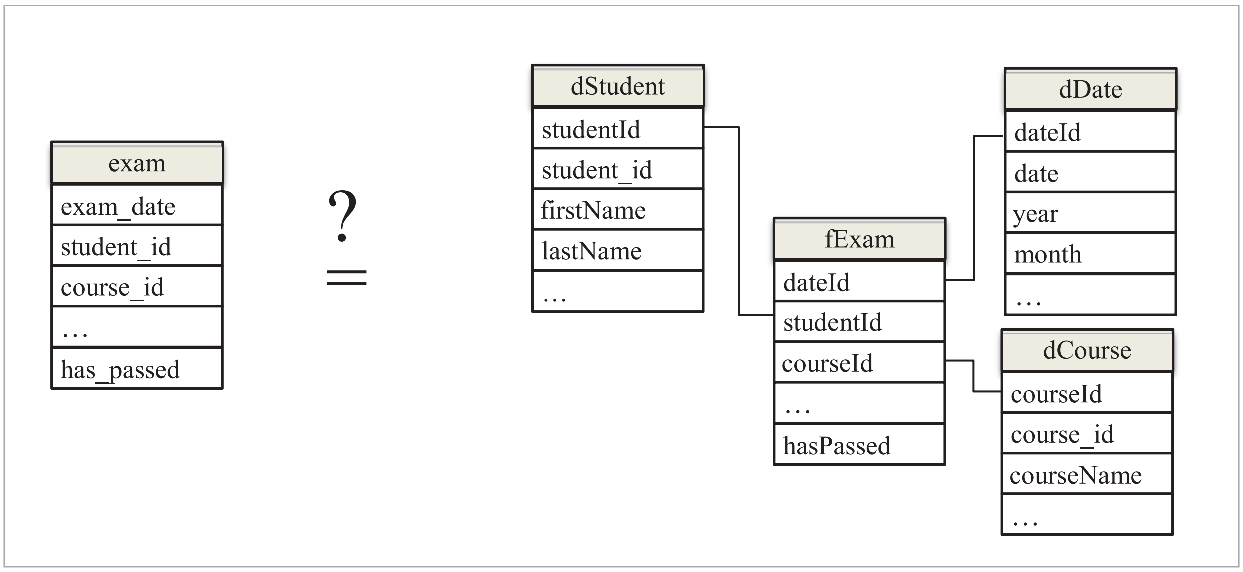
3.2. Determining Inequality in the Content of Tables Having Different Schemas

Comparing tables with different schemas typically occurs when comparing relational model tables and dimensional model [22] tables, e.g., staging area and DW. Dimension tables can be supported easier than fact tables since they mostly take over attributes from relational source(s). Sometimes attributes in the dimensional model are renamed using different nomenclature but different attribute names do not present a problem since they are described and paired via metadata. Fact tables, on the other hand, are more complex in this regard, because business keys are always replaced with surrogate keys from related dimension tables. In addition, some non-key attributes can be cataloged and replaced with surrogate keys. For instance, as shown in Figure 1, table *exam* is the source table for the fact table *fExam*. In table *fExam*, attribute *exam_date* has been cataloged and replaced with the surrogate key *dateID*, while the actual exam date has been renamed and stored in the dimension table *dDate* as *date.* The so-called, "junk dimensions" are another example of similar transformations.

To fragment and compare the contents of tables *exam* and *fExam* based upon the fragmentation set *X*={*exam_date*}, the fact table *fExam* has to be joined with the dimension table *dDate* before fragmentation. Effectively, it is necessary to evaluate q (*exam*, {*exam_date*}) and q (*fExam* $\bowtie$ *dDate*, {*date*}) and then compare the contents of the two, having in mind that attribute *exam_date* corresponds to attribute *date*.

In general, we compare an input set (any relation r in the source database or in the staging area) with an output set (a set of relations in the staging area or in the DW). For given X and $\mathcal{A}_{R,X}$, the relation r will be fragmented and specified aggregate functions will be evaluated for the fragments i.e. $q\left(r, X, \mathcal{A}_{R,X}\right)$ will be evaluated in accordance with the algorithm described in Section 3.1. The problem here is that the relation $s(R)$ does not exist. However, relation equivalent to $s(R)$ can be reconstructed from a set of relations $\mathbb{s}$. The set of relations $\mathbb{s}$ is the result of the transformation τ over r, i.e. $\mathbb{s} = \tau(r)$. To compare input and output sets using our algorithm it is necessary to apply inverse transformation τ^{-1} over output set, such that for each instance of the relation r we can state that $r = \tau^{-1}(\tau(\mathrm{r}))$. The existence of such an inverse transformation is not questionable if we adhere to the limitations specified in the introductory part of Section 3 (each record in a fact table corresponds to exactly one record in a data source table). Note that, for identical schema, both τ and τ^{-1} are the identity operators.In this example (Figure 1), the inverse

Figure 1
Comparing tables with different schemas (relational and dimensional)



transformation is simply defined with the operation of relational algebra:

$$\tau^{-1}(\text{fExam}, \text{dStudent}, \text{dDate}, \text{dCourse}) = \rho_{\boldsymbol{exam}(exam\_date, student\_id, course\_id, has\_passed)} (\pi_{date, student\_id, course\_id, hasPassed} (\text{fExam} \bowtie \text{dStudent} \bowtie \text{dDate} \bowtie \text{dCourse})).$$

Performing transformation τ^{-1}({fExam, dStudent, dDate, dCourse}) we acquire relation $s(R)$ which can be compared to the relation $r(R)$ using algorithm described in Section 3.1. More precisely, it is not necessary to reconstruct $s(R)$, it is sufficient to reconstruct the relation having all attributes contained in X and in $\mathcal{A}_{R,X}$. Commonly, the relation $s(S)$ will not be evaluated. The transformation τ^{-1} will be incorporated into an SQL statement which serves to evaluate $q(s, X)$.

The following example illustrates the procedure of fragmenting two data sets having different schemas, shown in Figure 1:

$$r(R) = exam(exam\_date, student\_id, course\_id, has\_passed)$$
$$X = \{exam\_date, student\_id, course\_id\}$$
$$s(R) = \rho_{exam2(exam\_date, student\_id, course\_id, has\_passed)} (\pi_{date, student\_id, course\_id, has\_passed} (fExam \bowtie dStudent \bowtie dDate \bowtie dCourse))$$
$$\mathcal{A}_{R,X} = \left\{ \begin{matrix} (sum, has\_passed, sumPassed), \\ (avg, has\_passed, avgPassed), \\ (count, \quad has\_passed, recCount) \end{matrix} \right\}$$
$$listX_R = \left\langle \begin{matrix} \{\}, \\ \{course\_id\}, \\ \{course\_id, student\_id\}, \\ \{course\_id, student\_id, exam\_date\} \end{matrix} \right\rangle$$
$$list\mathcal{A}_R = \langle \mathcal{A}_{R,X}, \mathcal{A}_{R,X}, \mathcal{A}_{R,X}, \mathcal{A}_{R,X} \rangle.$$

The algorithm produces the SQL statements shown in **Table 2.**

As it can be seen, the relation $s(S)$ is not evaluated, we just used transformation τ^{-1} when we needed it.

Table 2
Steps in fragmenting and comparing tables having different schemas

Depth	Source (RDB)	Destination (DWH)

1	`SELECT SUM(has_passed)   AS sumPassed,`
	`       AVG(has_passed)   AS avgPassed,`
	`       COUNT(has_passed) AS recCount`
	`  FROM exam`
		`SELECT SUM(hasPassed)   AS sumPassed,`
		`       AVG(hasPassed)   AS avgPassed,`
		`       COUNT(hasPassed) AS recCount`
		`  FROM fExam`

2	`SELECT course_id,`
	`       SUM(has_passed)   AS sumPassed,`
	`       AVG(has_passed)   AS avgPassed,`
	`       COUNT(has_passed) AS recCount`
	`  FROM exam`
	` GROUP BY course_id`
		`SELECT dCourse.course_id,`
		`       SUM(hasPassed)   AS sumPassed,`
		`       AVG(hasPassed)   AS avgPassed,`
		`       COUNT(hasPassed) AS recCount`
		`  FROM fExam`
		`  JOIN dCourse`
		`    ON fExam.courseId = dCourse.courseId`
		` GROUP BY dCourse.course_id`

Let's suppose we've found mismatch for the course_id=101 in the previous step:		

3	`SELECT course_id,`
	`       student_id,`
	`       SUM(has_passed)   AS sumPassed,`
	`       AVG(has_passed)   AS avgPassed,`
	`       COUNT(has_passed) AS recCount`
	`  FROM exam`
	` WHERE course_id = 101`
	` GROUP BY course_id,`
	`          student_id`
		`SELECT dCourse.course_id,`
		`       dStudent.student_id,`
		`       SUM(hasPassed)   AS sumPassed,`
		`       AVG(hasPassed)   AS avgPassed,`
		`       COUNT(hasPassed) AS recCount`
		` FROM fExam`
		` JOIN dCourse`
		`   ON fExam.courseId = dCourse.courseId`
		` JOIN dStudent`
		`   ON fExam.studentId=dStudent.studentId`
		`WHERE course_id = 101`
		` GROUP BY dCourse.course_id,`
		`          dStudent.student_id`

Depth	Source (RDB)	Destination (DWH)
Let's suppose we've found mismatch for the student_id=36 and student_id=37 in the previous step:		

4	`SELECT course_id,`
	`       student_id,`
	`       exam_date,`
	`       SUM(has_passed)   AS sumPassed,`
	`       AVG(has_passed)   AS avgPassed,`
	`       COUNT(has_passed) AS recCount`
	`  FROM exam`
	`  WHERE (course_id = 101 AND student_id = 36)`
	`     OR (course_id = 101 AND student_id = 37)`
	`GROUP BY course_id,`
	`         student_id,`
	`         exam_date`
		`SELECT dCourse.course_id,`
		`       dStudent.student_id,`
		`       dDate.date       AS exam_date,`
		`       SUM(hasPassed)   AS sumPassed,`
		`       AVG(hasPassed)   AS avgPassed,`
		`       COUNT(hasPassed) AS recCount`
		`  FROM fExam`
		`  JOIN dStudent`
		`    ON fExam.studentId=dStudent.studentId`
		`  JOIN dDate`
		`    ON fExam.dateId = dDate.dateId`
		`  JOIN dCourse`
		`    ON fExam.courseId = dCourse.courseId`
		` WHERE (course_id = 101 AND student_id = 36)`
		`    OR (course_id = 101 AND student_id = 37)`
		`GROUP BY dCourse.course_id,`
		`         dStudent.student_id,`
		`         dDate.date,`

Finally, after result sets from the previous steps are compared, the algorithm produces e.g., the following results:		

	`Missing: (101, 36, '2013-01-01')`
	`Excess:  (101, 37, '2012-02-02')`
		`Missing: (101, 37, '2012-02-02')`
		`Excess:  (101, 36, '2013-01-01')`

In general, a fact table schema does not have to contain business keys (primary keys) of the originating table [26]. It is possible that the business key is comprised of dimension tables' business keys; however, that is not always the case. When so, to positively identify the tuples, we employ the data lineage mechanism described in [26]. In short, every fact table has a coupled lineage table $linF$ with relation schema: $LinF = \{SK_F, PK_1, \dots, PK_i, \dots, PK_m\}$ where SK_F is the surrogate key of the fact table and PK_i are the business key attributes of the originating table. $linF$ is simply incorporated into τ^{-1} transformation when needed.

3.3. The Overall TCFC Algorithm

In the previous two sections, we have formally described the algorithm at table level. Using the described algorithm, with the help of metadata, we can now introduce a global, time-constrained algorithm for providing a "global overview" of the differences between two sets of tables.

The metadata needed for comparing an input set (relation $r(R)$) with an output set (set of relations $\mathbb{s}$) is denoted as the metadata quintuple $\mathcal{C} = (r(R), \mathbb{s}, \tau^{-1}, listX_R, list\mathcal{A}_R)$. Since the transformation τ is the transformation that has been applied to relation r to produce a set of relations $\mathbb{s}$, τ^{-1} will reconstruct relation $s(R)$, thus harmonizing the schema of the two data sets prior to comparison. The relations are then iteratively fragmented, aggregated and compared according to the fragmentation sets from $listX_R$ and aggregate sets from $list\mathcal{A}_R$. With $\mathcal{C}$ we denote a set of all metadata quintuples needed for comparing a pair of databases.

A comparison of two data sets, according to a metadata quintuple $\mathcal{C}$, is an iterative process that can be illustrated by an unbalanced tree where each node represents a fragment based on the aggregation set and the fragmentation set defined for that level (Figure 2). This means that comparing r and s according to X_1 and A_1 is represented with the root of the tree. If the comparison at the root level results with differences in fragments F_1, F_2, ..., F_n, the further comparison of fragments F_1, F_2, ..., F_n will be carried out according to X_2 and A_2 and will produce the root's children, etc.

In accordance with a request to examine as many tables as possible, perhaps at the expense of the completeness of the result, the algorithm should perform the comparison operations at the first level for all tables, then all assigned (and necessary) comparison of the fragments on the second level and so on.

This can be ensured by traversing tree according to the breadth-first order, which can be straightforwardly implemented using a queue.

Comparison of two data sets (databases) is comprised of a number of individual table comparisons. Each table comparison is a job to be performed. The idea of the TCFC algorithm (Algorithm 1) is to put all jobs in a priority (sorted) queue, and execute them one by one until the time runs out (lines 5 to 7), or all jobs get executed (line 29).

Algorithm 1: Time-Constrained Fragment and Compare (TCFC)

```
1   Input:   $\mathcal{C}$
2   Output:  differences report and completion status
3   report ← empty list          // list of differences
4   cQueue ← empty list          // compare jobs queue
5   on event (time frame has expired):
6     cancel running job(s)
7     return report, UNCOMPLETED
8   Begin
9     // initialization
10    $t_e$ ← empty tuple
11    for each c ∈ $C$
12      $cQueue.$**enqueue**$(c.r, \tau^{-1}(\mathbb{s}), c.listX_R, c.list\mathcal{A}_R, t_e)$
13    // computing
14    repeat
15      $cQueue.$**dequeue** into $(r, s, listX_R, list\mathcal{A}_R, t_x)$
16      $X_R \leftarrow head(listX_R)$
17      $\mathcal{A}_R \leftarrow head(list\mathcal{A}_R)$
18      $q_r \leftarrow$ **Compute Aggregates**$(r, X_R, \mathcal{A}_R, t_x)$
19      $q_s \leftarrow$ **Compute Aggregates**$(s, X_R, \mathcal{A}_R, t_x)$
20      for each $t \in (\pi_X(q_r) \setminus \pi_X(q_s))$
21        $report.$**add**$(r, t,$ MISSING$)$   // meaning: for input set r, t is missing from output set
22      for each t $\in (\pi_X(q_s) \setminus \pi_X(q_r))$
23        $report.$**add**$(r, t,$ EXCESS$)$   // meaning: for input set r, t is excessive in output set
24      for each $t_r \in q_r, t_s \in q_s :\ t_r[X] = t_s[X] \wedge t_r \neq t_s$
25        $report.$**add**$(r, t_r[X],$ DIFF$)$   //meaning: for input set r, $t_r[X]$ is excessive in the output set
26        if $tail(listX_R)$ is not empty
27          $cQueue.$**enqueue**$(r, s, tail(listX_R), tail(list\mathcal{A}_R), t_r[X])$
28    until $cQueue.isEmpty$
29    return report, COMPLETED
30  End
```

To achieve a global overview, tables are compared with an increasing level of detail in a round robin fashion. We use a list ($cQueue$) as an appropriate data structure for storing and managing the jobs. If, for example, we have to compare 10 pairs of tables, this list will initially contain 10 elements (jobs). The content of $cQueue$ is based on the set of metadata – quintuples $\mathcal{C}$ (line 1). The lists $cQueue$ and *report,* initialized at the beginning of the algorithm (lines 3 and 4), are used to store the metadata for the tables to be compared (line 12) and to store output data. Metadata, required to compare two data sets and stored in a database that serves as a repository for this algorithm, include: schema of the relation r; transformation τ^{-1} used to obtain relation s from the set of relations $\mathbb{s}$; nonempty list $listX_R$ of fragmentation sets; nonempty list $list\mathcal{A}_R$ of aggregation sets.

Each element of the $cQueue$ list contains elements $listX_R$ and $listA_R$ being lists as well. Since we use $cQueue$ list as a priority queue, the first element represents the job with the highest priority that will be next to be pulled out from the queue (line 15) and processed. One job may include a comparison on a number of levels (that number is determined by the number of $listX_R$ elements). Thus, the next step is to pull first elements from $listX_R$ (line 16) and $list\mathcal{A}_R$ (line 17) using function *head.* Based on the metadata, fragmentation and aggregate calculation is performed for relations r (line 18) and s (line 19). The result of the *Compute Agregates* algorithm (*Algorithm 2*) is a set of tuples whose schema is determined by the attributes contained in fragmentation set and aggregation set. The resulting sets of tuples q_r and q_s are being compared (lines 20 through 25) to find differences. Each tuple difference is added to the resulting list *report* (lines 21, 23 and 25). Instantly processed job will be removed from the list *cQueue* (lines 26 and 27). This way processed elements (jobs) are removed from the priority queue while elements (jobs) relating to unprocessed comparisons remain in the list. Differences t_r revealed between compared data sets are added to the priority queue as new jobs for further inspections. It is evident that this algorithm can generate children jobs – each tuple t_r generates a new job.

The *Compute Agregates* algorithm constructs an SQL statement similar to the statements from **Table 2.**

```
Algorithm 2:   Compute Aggregates
1   Input:     $(r, X, \mathcal{A}, t)$          // r is an relation or relational algebra expression
2   Output:    set of tuples
3   Begin
4     selectClause ← attributes from X and list of aggregate functions from $\mathcal{A}$, renamed accordingly
5     fromClause ← r
6     if t is empty tuple
7       whereCondition ← true
8     else
9       T ← schema of tuple t
10      whereCondition ← $A \in T \wedge A = t[A]$
11    groupClause ← list of attributes from X
12    execute sql statement for selectClause, fromClause, whereCondition, groupClause
13    return resulting tuples
14  End
```

The `SELECT` clause contains all attributes from the *X* list, and a list of aggregate functions with accompanying arguments and associated names (line 4). The `WHERE` part contains specific conditions, but only when input tuple *t* is not an empty tuple. It is not empty when difference between compared tables is detected and further fragmentation is done in order to find more detailed information. The join conditions (lines 9 and 10) are being built according to the metadata from mentioned repository which takes into account the attributes contained in *X*. The output of the algorithm is a relation – a set of tuples with a known schema (line 2).

For the sake of clarity, some implementation details are omitted in the pseudo-code. They will be described in the remainder of this section.

Job order: execution path (tree, e.g., Figure 2) is defined by the ordering of jobs. Jobs are ordered according to three parameters, in the following order:

1 **Error depth**, the algorithm will first pursue the branches in which errors are spotted.

2 **Depth**: if there are no errors, the algorithm will consider job depth (level) and thus uniformly spread the execution over all tables (compare jobs with depth 1 for all tables, then jobs with depth 2 for all tables, etc.).

3 **Priority**: users can set priority for each table. Tables with higher priority are inspected sooner within the same (error) depth.

Seek depths: algorithm considers two depths for a single job (comparison) that are defined and stored in the metadata repository:

1 **Full depth** is implicitly defined with the number of levels (and according attribute sets) defined in the metadata. Full depth is used only when an error has been pursued.

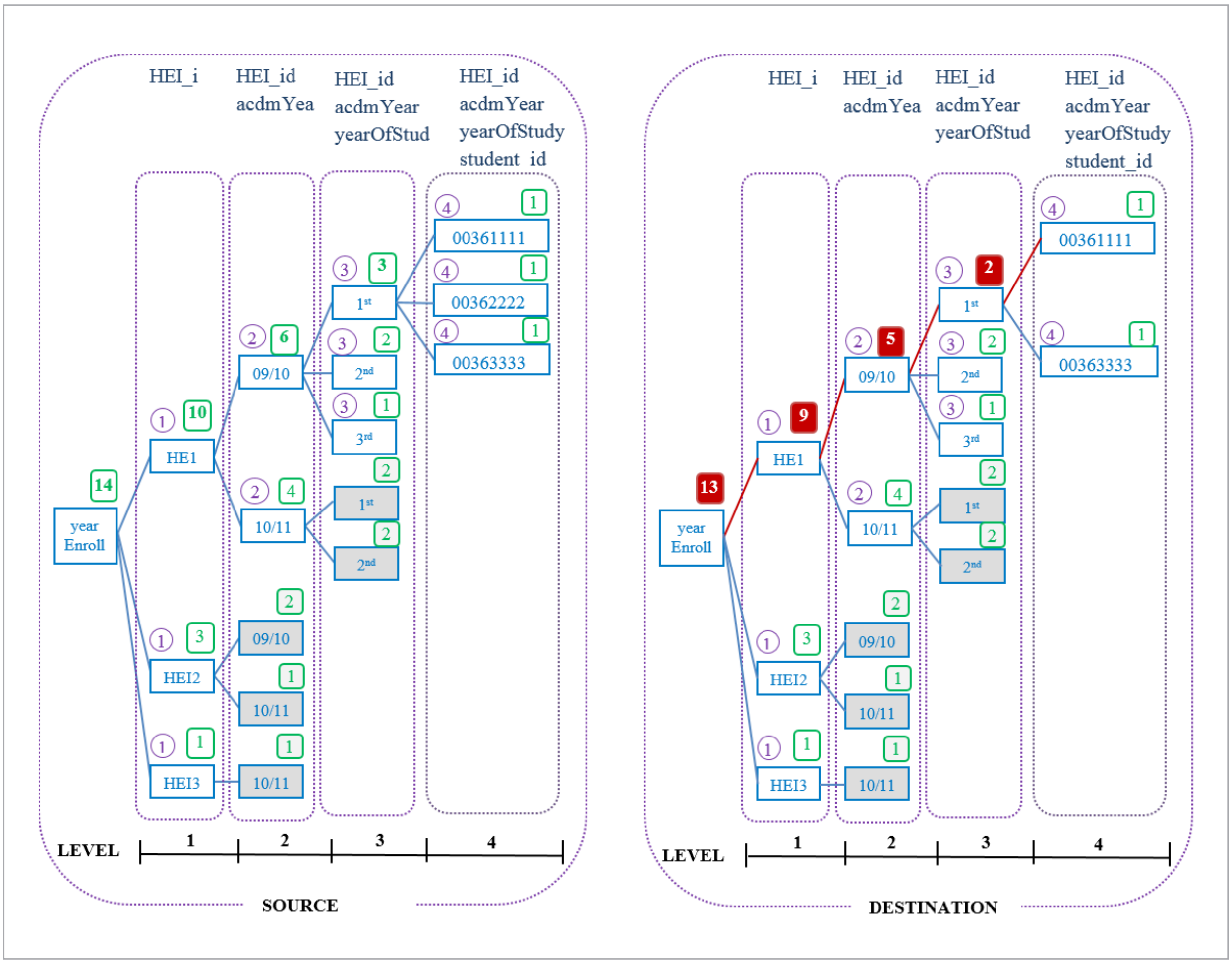
2 **Healthy job depth:** the other, smaller depth, is defined for the jobs where no errors were found ("healthy jobs").

Defining two depths allows us to find precise differing rows, while abandoning the search sooner for tables which show no differences in the early part of the algorithm. Comparing huge tables with no differences all the way to the primary key would be too slow and inefficient.

Figure 2 shows the fragmentation of the *yearEnroll* table that pertains to the year enrolment process at a HEI (High(er) Education Institution): a student (*student_id*) enrolls in a year of study (*yearOfStudy*) in an academic year (*acdmYear*). The left-hand side shows a *yearEnroll* table in the source system, and the right-hand side shows its copy – *yearEnroll* table in the staging area. The primary key is $K_{yearEnroll}$= {*HEI_id, acdmYear, yearOfStudy, student_id*}. At the first level, the fragmentation is carried out according to *HEI_id*, then by *acdmYear* and so forth. For the sake of simplicity, we show only the count aggregate function in green colour (on the left side) and in green and red (where they differ) on the right side. Healthy job depth is set to the second level. Had there been no errors, the comparison would have been carried out and stopped on {*HEI_id, acdmYear*} fragment/depth. In this example though, the difference in counts is found at the first level, for the HEI1 (10<>9). Following on that, the algorithm focuses on the HEI1 and continues to fragment that branch. At the second level, the difference is narrowed to (HE1, 09/10), then to the (HE1, 09/10, 1st), and, finally, the exact tuples are found. The algorithm steps (ordered path) are denoted with numbers in circles. Note that, in accordance with the job ordering strategy described above, the algorithm primarily explores the erroneous branches.

Figure 2

A tree representation of TCFC algorithm's execution



The algorithm will not further explore healthy branches if it finds an erroneous branch. Healthy children jobs are loaded only if there are no errors. This means that algorithm will not further explore branch (HEI1, 10/11) in Figure 2. This behaviour could be easily modified by loading the healthy branches into the queue with the lowest priority (to be executed last, if there is any time left). However, this would be suboptimal, since there could be a great number of healthy branches. Imagine we have additional 50 years (besides 2009 and 2010) in Figure 2 – this would spawn another 50 queries. In general, it would be better to execute one larger query (with all 51 years) and then ignore the already processed erroneous branches. This functionality implies a more significant change to the algorithm and is not presented here.

Another feature of the algorithm that was omitted from the formal description is the ability to reuse the results from the previous run. If we accept the assumption that most of the errors between two ETL refresh cycles remain the same, especially when incremental loading of DW is employed, it is prudent to store results in the metadata repository and examine them first in the next run. These "narrow" queries are much faster than queries at the ground levels that are targeting the whole table. The results of those queries can be used to prune the execution tree as soon as the difference is accounted for. For instance, say we add one row to the (HEI3, 10/11, 5th) branch and run the algorithm again: what were steps 1 and 4 would now be carried out in the first step at level one. But the difference is now in fragments HEI1 and HEI3. The difference (one row) found in the leaf job (HEI1, 09/10, 1st) would account for the HE1 branch and it would be pruned. The algorithm would continue to drill down on HEI3 branch only, to find the newly added difference. In general: a branch can be pruned if there is a leaf job with matching prefix fragment values ((HEI1) matches (HEI1, 09/10, 1st)) and error counts. This feature of the TCFC algorithm gives the ability to benefit from previous runs (learn from the data, in a way) and drastically reduce the execution time. Note that, this way, it is possible to refine search results after being interrupted (because of the limited time) – a job that was interrupted today might finish tomorrow, or the day after. This feature was not included in the comparison with the reference implementation in Section 3.1. because it would give us an unfair advantage.

3.4. Fragmentation Strategy Recommendations

For any non-trivial table, a number of fragmentation paths is huge. For instance, for just four possible fragmentation attributes (e.g., Figure 2) there are 148 different fragmenting strategies: 1 for zero level (no grouping), 14 one level strategies, 49 two level strategies, 60 three level and 24 four level strategies. Based on our experience, we provide the following recommendations:

- Fragmentation strategy should be determined with the help of a domain expert – a person who has a good knowledge of the data semantics and business processes.
- Healthy job depth should be set for all relations, except for low-cardinality relations where only one level should be used, i.e. where primary key is used at the first level.
- Higher fragmentation levels must not include attributes that are functionally dependent on the attributes from lower fragmentation levels.
- The number of tuples in children fragments should be for at least an order of magnitude less than the number of tuples in the parenting fragment. This is particularly important for the first fragmentation level – this is where a huge table is reduced to *N* smaller "tables" (fragments). Hopefully, if the data are relatively uniformly distributed over the first fragmentation attribute set (usually, data are uniformly distributed at least over the time attribute), then the number of tuples can be reduced by the chosen order of magnitude by creating tens or hundreds of fragments. On the other hand, the number of fragments should not be too big because this would facilitate error scattering over different fragments. The best-case scenario is if all the errors are in the same fragment, so that a single branch is pursued.
- The total number of fragmentation levels should not be too big or too small. Based on tests conducted in this paper, for the relations of 100 million tuples, we consider two to four to be the optimal number of levels.
- If possible, build a composite index on the corresponding attributes of the last level of fragmentation.

In the case where DBMS keeps accurate statistics on the row count of tables, it is suitable to start comparing data sets with empty fragmentation set (at first level) with just one aggregate function – COUNT, since the result of the SELECT COUNT(*attributeName*) will be instantaneous. If statistics are not kept, such a statement would cause a full table scan and then it is better to perform the fragmentation with nonempty fragmentation set and multiple aggregate functions.

3.5. Algorithm Evaluation

The TCFC algorithm was tested in terms of speed and accuracy. It is implemented in C# programming language. Testing was executed on a PC with Intel ® Core ™ i7-4770 CPU processor with 16 GB of RAM and Microsoft Windows 8 operating system. For database servers, two virtual machines with equal configurations were used, namely, Intel ® Xeon ® processor E7540 (2.00 GHz clock speed), 8 GB of RAM, Windows Server Standard operating system and SQL Server 2014 DBMS (with auto update statistic option).

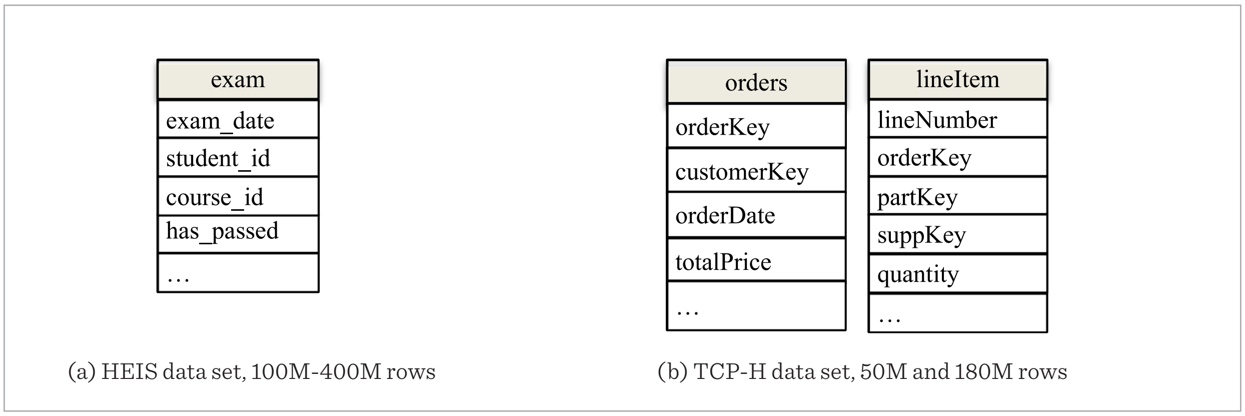
Testing was carried out on data sets with cardinality between 50 million and 400 million tuples which we consider comparable to cardinality in a real-world data warehousing systems (at least for the incremental daily load) and for variable number of the differences between compared data sets.

Figure 3 shows schemas of two data sets from different domains used for algorithm evaluation:

a Real world data from the field of higher education [25] (relation exam in Figure 3(a)) that was extrapolated (from initial 10 million) to relations exam100M, exam200M and exam400M with 100, 200 and 400 million tuples i.e. 8GB, 15GB and 30GB data, respectively.

b Well-known TPC Benchmark ™ H (TPC-H) programmatically generated data set [37] (relations orders and lineItem in Figure 3(b), containing 50 and 180 million tuples, i.e. 5GB and 25GB data, respectively).

Figure 3
Schemas of data sets used for algorithm evaluation



Primary keys are as follows:

$$K_{EXAM} = \{student\_id, course\_id, exam\_date\}$$
$$K_{ORDERS} = \{orderKey\}$$
$$K_{LINEITEM} = \{orderKey,\ suppKey,\ pArtKey\},$$

where pArtKey is the article key.

TCFC's speed is compared to the execution speed of the referent SELECT statements, shown below whose execution time is referred to as *referent execution time* in the rest of the paper. In this experiment, we have deliberately used the same DBMS on both servers, because SQL Server provides the execution of queries involving tables stored on remote servers (via the linked server feature). The SQL statement used to find the differences between the two instances of ***exam100M*** tables (and analogous statements were used for the remaining ***exam*M*** table pairs) is shown below. To connect to the staging area we used linked server named "SASrv.ZPR.FER.HR":

```
SELECT *

  FROM HEISsrc.dbo.exam100M src
  FULL OUTER JOIN [SASrv.ZPR.FER.HR].HEISsa.dbo.exam100M dest
    ON src.student_id = dest.student_id
   AND src.course_id = dest.course_id
   AND src.exam_date = dest.exam_date
 WHERE src.student_id IS NULL
    OR dest.student_id IS NULL
    -- OR (dest.NonKeyAttrib <> src.NonKeyAttrib) , for each remaining Non-Key attribute
```

Similar statements were used to compare ***orders*** and ***lineItem*** pairs:

```
SELECT *
  FROM TPCHsrc.dbo.orders src
  FULL OUTER JOIN
  [SASrv.ZPR.FER.HR].TPCHsa.dbo.orders dest
    ON src.orderKey = dest.orderKey
 WHERE src.orderKey IS NULL
    OR dest.orderKey IS NULL
-- OR (dest.NonKeyAttrib <>
--  src.NonKeyAttrib) , for each remaining
--                      Non-Key attribute
```

```
SELECT *
  FROM TPCHsrc.dbo.lineitem src
  FULL OUTER JOIN
  [SASrv.ZPR.FER.HR].TPCHsa.dbo.lineitem dest
    ON src.orderKey = dest.orderKey
   AND src.partKey  = dest.partKey
   AND src.suppKey  = dest.suppKey
 WHERE src.orderKey  IS NULL
    OR dest.orderKey IS NULL
-- OR (dest.NonKeyAttrib <>
--  src.NonKeyAttrib) , for each remaining
--                      Non-Key attribute
```

For the sake of brevity, the above SQL statements do not list all OR statements that check the potential differences in non-key attributes; that part is represented as a comment.

The fragmentation presented in Table 1 was used. The last fragmentation level for each relation includes all primary key attributes and consequently the comparison will pinpoint the exact missing/excess tuples. Differences between the tables were generated by deleting tuples from the source table. Both, number and dispersion on differences across fragments were varied, since error dispersion significantly affects the performance. For instance, if a pair of ***exam100M*** relations differs in 1‰, i.e., 100.000 tuples, in the worst case, at the last level, an equal number of queries would be generated (though it is very unlikely errors would align with the fragmentation set in such way at the penultimate level) and in the best case a single query would detect all

Table 1
HEIS and TPC-H fragmentation strategy

r	$X, \mathcal{A}_{R,X}, list\mathcal{A}_R$
exam	$X = \{student\_id, course\_id, exam\_date\}$ $\mathcal{A}_{R,X} = \{(count, has\_passed, recCount)\}$ $listX_R = \left\langle \begin{matrix}\{student\_id\},\\ \{student\_id, course\_id, exam\_date\}\end{matrix} \right\rangle$
orders	$X = \{customerKey, orderKey\}$ $\mathcal{A}_{R,X} = \{(count, orderKey, recCount)\}$ $listX_R = \left\langle \begin{matrix}\{customerKey\},\\ \{customerKey, orderKey\}\end{matrix} \right\rangle$
line Item	$X = \{partKey, suppKey, orderKey\}$ $\mathcal{A}_{R,X} = \{(count, orderKey, recCount)\}$ $listX_R = \left\langle \begin{matrix}\{artKey\},\\ \{pArtKey, suppKey, orderKey\}\end{matrix} \right\rangle$

100k differences, as they would all be in the same fragment. Dispersion was varied using 20%, 50% and 100% dispersion. Dispersion of 20% means that the differences are scattered in 20% of the total fragments in the penultimate fragmentation level.

Figure 4 presents evaluation results for the two stated data sets. Legend items indicate the names of relations, cardinality and the error dispersion. In all graphs in Figure 4, the x-coordinate represents the number of differences in per mills of the relation cardinality, while the y-coordinate shows the time spent finding differences relative to the reference implementation time, meaning that TCFC algorithm finds differences faster than the referent SELECT statement for all scenarios where parts of the graphs are below a 100% line. For instance, at the top left graph, the point where the graph exam100M-20% reaches 100% of the referent execution time is for the number of errors between 12 and 13 ‰ (i.e. between 1.2 million and 1.3 million differences), while the graph named ispit100M-100% reaches the referent execution time for approximately 2.5 ‰ (i.e., 250.000 differences). Increasing dispersion from 20% to 100% reduces the speed of the process by 80% (approximately). In the same time frame, TCFC finds only 20% differences dispersed with the 100% rate compared to finding differences dispersed with the 20% rate.

Testing was conducted in two different modes:

1. non-indexed: both tables without indexes (even primary key constraints), shown in Figure 4(b) and Figure 4(d), and
2. indexed, with indices appropriate to the chosen fragmentation strategy/primary key, shown in Figure 4(a) and Figure 4(c).

Graphs show a few common characteristics for all experiments:

_ For a sufficiently small number of differences, TCFC outperforms the relational engine, i.e. it is faster than the SELECT statement spanning tables from remote servers.

_ Indices that follows a fragmentation strategy improve the performance of TCFC.

_ Error dispersion, which is closely related to the fragmentation strategy, significantly affects the performance.

The first observation can also be stated as the follows: after a certain point of differences, the TCFC algorithm shows worse results than the relational engine. However, the following applies:

_ In DW environment, it is reasonable to expect the number of differences to be small, and that is the scenario that we were targeting.

_ Differences of several per mills (where the TCFC outperforms the reference implementation) relative to hundreds of millions of rows amount to hundreds of thousands or even millions of errors: that is either an evidence of systemic error that will eliminate most of the erroneous rows once corrected, or an unmanageable amount of errors. Either way, with such large row counts, errors in per mills present more than enough work for a person to rectify between two ETL cycles.

_ Even when the number of differences is not small, the TCFC algorithm is time-constrained. For instance, the threshold can be set to match the referent implementation time.

_ If the assumption that errors remain between ELT cycles holds, the TCFC will improve over time as it will use previous results.

Finally, one must have in mind that it is often not even possible to write (execute) SQL statements that compare tables from different databases when different vendor's DBMSs are used.

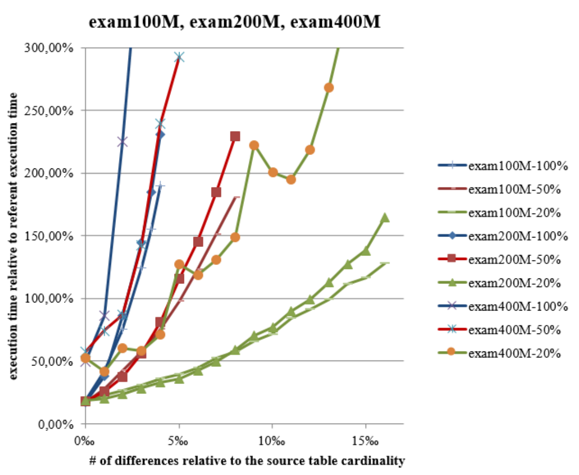
For all the reasons stated above, we believe that the TCFC algorithm is a perfect fit for ETL integration testing, and is potentially a very useful method for large set comparison in general.

While carrying out the experiments, different fragmentation strategies were used and it was proved that poor fragmentation strategy can deteriorate the performance of the algorithm. Finding the optimal fragmentation strategy is therefore a task that should be performed with care as it requires knowledge of the data (trends) and fair knowledge of SQL and query optimization techniques. This is somewhat similar to index creation in the relational databases where poorly chosen index can have a negative impact on the performance.

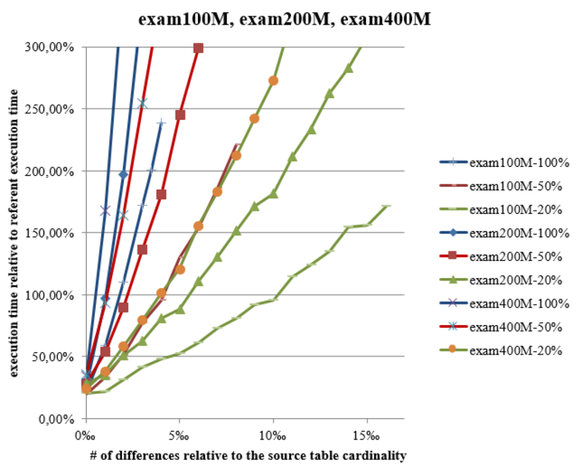
In terms of accuracy, the algorithm has proven to be accurate – all differences were found in each run (as mentioned before, there is a miniscule chance of errors being ignored when numbers nullify each other).

Figure 4

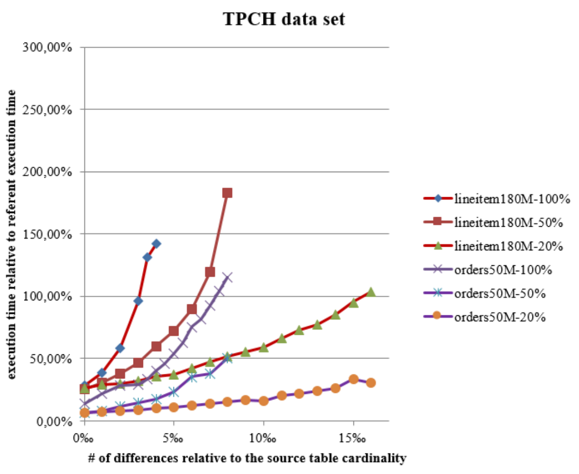
Evaluation results for different data sets and indexing strategies. Legend items indicate the names of the relations, cardinality and the error dispersion



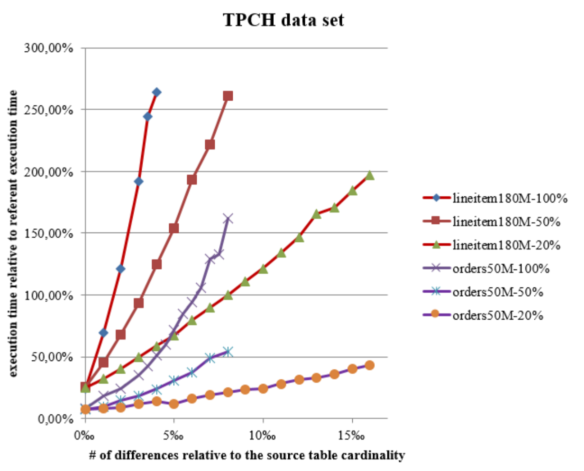
(a) **Indexed data**, exec time relative to SQL-outer-join



(b) **Non-Indexed data**, exec time relative to SQL-outer-join



(c) **Indexed data**, exec time relative to SQL-outer-join



(d) **Non-Indexed data**, exec time relative to SQL-outer-join

4. Related Work

The literature on software testing is vast and comprehensive, but DW and ETL testing has gained far less attention. Though fundamental principles and techniques apply, DW testing differs significantly from the software testing, mainly because DW testing is primarily data oriented (as opposed to program code), and data volume tends to be large. A thorough overview of DW testing differences can be found in [11], [15-17].

Mookerjea and Malisetty [27] described the main phases of DW testing, outline main challenges in DW testing and propose a set of best practices in ETL and DW testing, as well as Singh [33], while Singh and

Kawaljeet [34] gave a descriptive classification of data quality problems at all phases of DW project: data sources, data integration, data staging/ETL and DW design. Singh [33] also considered ideal to perform integration testing on real production data, while in Mookerjea and Malisetty [27] authors proposed to base part of the testing activities (those related to incremental load) on mock data. We started our evaluation process with real project data (Information System of Higher Education Institutions in Republic of Croatia – ISVU) [19], but in order to test on large data sets (over 100 million tuples), we proceeded with testing on mock (extrapolated real) data [37].

A number of papers deal with particular aspects of DW testing. For instance, in Thomsen and Pedersen [36] a semi-automatic regression testing framework used to compare data between ETL runs is proposed with the purpose of catching new errors when software is updated. Rodić and Baranović [32] proposed generation of data quality rules and their integration into ETL process. In Santos et al. [28-29], authors attempted to automate the selection and execution of previously identified test cases for loading procedures in BI environments based on a DW. To validate the approach, the authors have developed a unit test framework and conducted an experiment showing reduced test effort when compared with manual execution of test cases or generic framework, such as DBUnit [10]. In Dakrory et al. [6], authors proposed a framework for automating ETL testing for data quality which delivers a wide coverage for data quality testing by framing testing activities within a modular methodology that can be customized according to ETL specificities, business rules, and constraints. In Williams [39], the author employed Data Vault-based Enterprise Data Warehouse and concluded that such an architecture can simplify and enhance various aspects of testing, and curtail delays that are common in DW projects.

In a more generic and exhaustive sense, two contributions stand out.

Firstly, Golfarelli and Rizzi [14-16] provided a comprehensive view of DW testing addressing the problem from various perspectives. The authors identify the following components to be tested: Conceptual schema, Logical Schema, ETL procedures, Database and Front-end, and introduce the classification of testing activities in terms of "what" is tested (addressing data quality) and "how" it is tested (addressing test type, e.g., performance test). They define a comprehensive methodological framework for data mart testing which includes eight phases: Requirement analysis, Analysis and Reconciliation, Conceptual design, Workload refinement, Logical design, Data staging design, Physical design and Implementation.

Secondly, ElGamal et al [11-12] brought an extensive overview of DW testing approaches, divided in four categories: software based testing approaches, ETL based, Multi-perspective and CASE-tool based. The authors define a three-dimensional DW testing matrix with regards to where, what and when is tested, and conclude that none of the previous approaches address the entire matrix. Unlike previous approaches, the authors take into consideration different DW architectures and provide a much more detailed and comprehensive description of all test routines to be administered in a DW project. With all that in mind, a generic testing framework based on the generic Kimmon DW architecture including aforementioned routines is proposed. Interestingly, the proposed routines feature overall "record counts" comparisons and "random record comparisons", probably considering that the overall comparison of records is a very challenging task in a realistic DW environment, where huge amounts of data and heterogeneous database engines are typically found. Moreover, all the aforementioned research envisions record (count) comparison in various stages of ETL/DW project, but none of them comment on how to conduct the comparison. This is where our work nicely fits in, as it provides a detailed algorithmic instruction on *how to perform* this comparison that is at the foundation of any DW testing system.

Krawatzeck et al. [23] identified a gap between scientific approach and the actual implementation in the real-world scenarios and performed an evaluation of open-source DW unit testing tools (e.g., DBUnit [10]) to address that issue, concluding that some promising tools for the DW testing exist, with a preference for the DBFit tool [9] as the only vendor-independent tool.

Understandably, ETL/DW testing is also a major topic in the industry. Major vendors provide some sort of testing integrated with their data integration and data quality tools. For instance, Informatica [1] features a "Data Validation Option" tool [7] used to compare two data sets, though no information is provided as to how it is done. Furthermore, there are companies and tools developed solely for the purpose of testing a DW. The QuerySurge [31] and ETL Validator [8] are commercial CASE tools developed by RTTS and Datagaps companies, respectively, to automate the testing and validation in the Big Data and DW systems. Under ETL process testing, QuerySurge CASE tool offers column-level comparison, table-level comparison and row count comparison. Each of these testing types comes down to automatic generation of SQL queries for comparing pairs of data sets based on row counts or the values of attributes. Row count comparison only determines the number of tuples in comparing pairs of tables and does not indicate tuple(s) that caused the difference. Table-level testing type produces two data sets and subsequently compares them. The only difference between column and table-level testing is that at the column-level only certain attributes can be chosen and compared. Both types of tests can be performed using approach presented in this paper. The advantage of our approach is in speed – a consequence of TCFC algorithm's feature that only table fragments that may contain differences are inspected. In addition, to the best of our knowledge, none of the available commercial solutions provide a *time-constraint feature*, which is essential in a DW testing scenario.

5. Conclusion

In this paper, we describe an integration testing procedure for a DW environment, with an emphasis on the generic time-constrained algorithm for comparing two tables. The goal of the algorithm is to provide the "global overview" of the data set's differences in a given time frame. By global overview, we mean that the largest possible set of tables should be examined, at the expense of the completeness of results for any single table. Metadata are used to describe data sets, and to configure and steer the algorithm. Both relational and dimensional data models are supported, so that the algorithm can be used to compare data from the various stages of the ETL cycle. The algorithm can use the results from the previous runs to execute more effectively. Parts of the algorithm are formally described, and the overall algorithm is presented in pseudo-code. Evaluation of the algorithm on the real-world data of the project and on TPC-H data set has shown that it outperforms the SQL Server relational engine SELECT statement when the percentage of missing or excess tuples is relatively small, which is a scenario typical of a DW environment. Although we state that the algorithm competes with the relational engine, it actually uses the relational engine and the whole process can be viewed as a query optimization technique: the exhaustive SELECT statement is "broken down" into a set of smaller statements that are trying to pinpoint the unequal fragments. Performing the search with many "small" queries provides additional benefits: the whole process can be gracefully time-constrained and more tables can be inspected to achieve the desired global view of the data, in contrast to a single exhaustive SELECT statement that might spend the entire available time on a single table.

Acknowledgment

This work was supported by the Croatian Science Foundation under the project "PROSPER – Process and Business Intelligence for Business Performance" (IP-2014-09-3729) and by the project "Advanced methods and technologies for data science and cooperative systems – DATACROSS" funded from the European Structural and Investment Funds through the Operational Programme Competitiveness and Cohesion 2014-2020, contract no. KK.01.1.1.01.0009.

References

1. 2017 Gartner Magic Quadrant for Data Integration Tools [Online], Gartner. Accessed 15.9.2017. Available: https://www.informatica.com/data-integration-magic-quadrant.html.
2. Ballou, D. P., Pazer, H. L. Modelling Completeness Versus Consistency Tradeoffs in Information Decision Contexts. IEEE Transactions on Knowledge and Data Engineering, 2003, 15(1), 240-243. https://doi.org/10.1109/TKDE.2003.1161595
3. Batini, C., Scannapieca, M. Data Quality Concepts, Methodologies and Techniques. Heidelberg, Berlin, Springer-Verlag, 2006.
4. Cui, Y., Widom, J., Wiener, J. L. Tracing the Lineage of View Data in a Warehousing Environment. ACM Trans-

actions on Database Systems (TODS), 2000, 25(2), 179-227. https://doi.org/10.1145/357775.357777

5. Cui, Y., Widom, J. Lineage Tracing for General Data Warehouse Transformations. The VLDB Journal, 2003, 12(1), 41-58. https://doi.org/10.1007/s00778-002-0083-8

6. Dakrory, S. B, Mahmoud, T. M., Ali, A. A. Automated ETL Testing on the Data Quality of a Data Warehouse. International Journal of Computer Applications, 2015, 131(16), 9-16.

7. Data Validation Option for PowerCenter (DVO) [Online]. Informatica. Accessed 15.9.2017. Available: https://www.informatica.com/services-and-training/informatica-university/find-training/data-validation-option-for-powercenter-dvo/ondemand.html.

8. Datagaps ETL Validator [Online]. Datagaps. Accessed 15.9.2017, Available: http://www.datagaps.com/etl-testing-tools/etl-validator.

9. DbFit Test-Driven Database Development [Online]. DbFit. Accessed 15.9.2017, Available: http://dbfit.github.io/dbfit/index.html.

10. DBUnit [Online]. Accessed 15.9.2017. Available: http://dbunit.sourceforge.net/.

11. ElGamal, N. Data Warehouse Testing. PhD Dissertation, Cairo University, Faculty of Computers and Information, Information Systems Department, 2015.

12. ElGamal, N., El-Bastawissy, A., Galal-Edeen G. An Architecture-Oriented Data Warehouse Testing Approach. 21st International Conference on Management of Data (COMAD), Pune, India, 2016, 24-34.

13. Even, A., Shankaranarayanan, G. Utility-Driven Assessment of Data Quality. The DATA BASE for Advances in Information Systems, 2007, 75-93. https://doi.org/10.1145/1240616.1240623

14. Golfarelli M., Rizzi S. A Comprehensive Approach to Data Warehouse Testing. DOLAP'09 Proceedings of the ACM 12th International Workshop on Data Warehousing and OLAP, Hong Kong, China, 2009, 17-24. https://doi.org/10.1145/1651291.1651295

15. Golfarelli, M., Rizzi, M. Data Warehouse Testing: A Prototype-Based Methodology. Information and Software Technology, 2011, 53(11), 1183-1198. https://doi.org/10.1016/j.infsof.2011.04.002

16. Golfarelli, M., Rizzi, S. Data Warehouse Testing. Developments in Data Extraction, Management, and Analysis, 2013, 91-108. https://doi.org/10.4018/978-1-4666-2148-0.ch005

17. Gupta, S. L., Pahwa, P., Mathur, S. Classification of Data Warehouse Testing Approaches. International Journal of Computers & Technology, 2012, 3(3), 381-386.

18. Heinrich, B., Klier, M. A Novel Data Quality Metric for Timeliness Considering Supplemental Data. 17th European Conference on Information Systems (ECIS), Verona, 2009.

19. Information System of Higher Education Institutions in Republic of Croatia – ISVU, [Online]. University of Zagreb Faculty of Electrical Engineering and Computing. Accessed 15.9.2017. Available: http://www.isvu.hr

20. Jarke, M., Lenzerini, M., Vassiliou, Y., Vassiliadis, P. Fundamentals of Data Warehouses. Berlin, Springer-Verlag, 2000. https://doi.org/10.1007/978-3-662-04138-3

21. Kaiser, M. A Conceptional Approach to Unify Completeness, Consistency and Accuracy as Quality Dimensions of Data Values. European and Mediterranean Conference on Information Systems, Abu Dhabi, UAE, 2010, 1-17.

22. Kimball, R., Ross, M. The Data Warehouse Toolkit: The Complete Guide to Dimensional Modelling (Second Edition), John Wiley & Sons, 2002.

23. Krawatzeck, R., Tetzner, A., Dinter, B. An Evaluation of Open Source Unit Testing Tools Suitable for Data Warehouse Testing. PACIS, 2015.

24. Mathen, M. Data Warehouse Testing. DeveloperIQ Magazine, 2010.

25. Mekterović, I., Brkić, Lj., Baranović, M. Improving the ETL Process and Maintenance of Higher Education Information System Data Warehouse. WSEAS Transactions on Computers, 2009, 8(10), 1681-1690.

26. Mekterović, I., Brkić, Lj., Baranović, M. A Generic Procedure for Integration Testing of ETL Procedures. Automatika, 2011, 52(2), 169-178.

27. Mookerjea, A., Malisetty, P. Data Warehouse/ETL testing: Best Practices [Online]. Accessed 15.9.2017. Available: http://test2008.in/Test2008/pdf/Anandiya%20et%20al%20-%20Best%20Practices%20in%20data%20warehouse%20testing.pdf, 2008.

28. Santos, I., Nascimento, A., Costa, J., Júnior, M. Experimentation in the Industry for Automation of Unit Testing in a Business Intelligence Environment. The 28th International Conference on Software Engineering and Knowledge Engineering (SEKE), Redwood City, USA, 2016, 466-469. https://doi.org/10.18293/SEKE2016-182

29. Santos, I., Costa, J., Júnior, M., Nascimento, A. Experimental Evaluation of Automatic Tests Cases in Data Analytics Applications Loading Procedures. Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS), 2017, 1, 304-311. https://

doi.org/10.5220/0006337503040311

30. Pipino, L. L., Lee, Y. W., Wang, R. Y. Data Quality Assessment. Communications of the ACM, 2002, 45(4), 211-218. https://doi.org/10.1145/505248.506010

31. QuerySurge: Automate Your Big Data & Data Warehouse Testing and Reap the Benefits [Online]. QuerySurge. Accessed 15.9.2017. Available: http://www.querysurge.com/.

32. Rodić, J., Baranović, M. Generating Data Quality Eules and Integration into ETL Process. Proceedings of the 12th International Workshop on Data Warehousing and OLAP (DOLAP 2009), Hong Kong, 2009, 65-72.

33. Singh, A. ETL Testing: Best Practices. Software Testing Conference, Bangalore, India, 2010.

34. Singh, R., Kawaljeet, S. A Descriptive Classification of Causes of Data Quality Problems in Data Warehousing. International Journal of Computer Science Issues, 2010, 7(3), 41-50.

35. SQL Data Examiner [Online]. TulaSoft. Accessed 15.9.2017. Available: http://www.sqlaccessories.com/sql-data-examiner/.

36. Thomsen, C., Pedersen, T. B. ETLDiff: A Semi-Automatic Framework for Regression Test of ETL Software. International Conference on Data Warehousing and Knowledge Discovery, DaWaK 2006, Krakow, Poland, 2006, 4081, 1-12. https://doi.org/10.1007/11823728_1

37. TPC-H [Online]. TPC. Accessed 15.9.2017. Available: http://www.tpc.org/tpch/default.asp.

38. Wang, R. Y., Reddy, M. P., Kon, H. B. Toward Quality Data: An Attribute-based Approach. Decision Support Systems, 1995, 13(3-4), 349-372. https://doi.org/10.1016/0167-9236(93)E0050-N

39. Williams, C. Experimentation with Raw Data Vault Data Warehouse Loading. Proceedings of the Southern Association for Information Systems Conference, St. Augustine, FL, USA, March 18-19, 2016, 1-6.