Differential Data Quality Verification on Partitioned Data

Sebastian Schelter, Stefan Grafberger, Philipp Schmidt, Tammo Rukat,
Mario Kiessling, Andrey Taptunov, Felix Biessmann, Dustin Lange
Amazon Research
{sseb,stefgraf,phschmid,tammruka,kiesslin,taptunov,biessman,langed}@amazon.com

***Abstract*—Modern companies and institutions rely on data to guide every single decision. Missing or incorrect information seriously compromises any decision process. In previous work, we presented *Deequ*, a Spark-based library for automating the verification of data quality at scale. *Deequ* provides a declarative API, which combines common quality constraints with user-defined validation code, and thereby enables *unit tests for data*.**

However, we found that the previous computational model of *Deequ* is not flexible enough for many scenarios in modern data pipelines, which handle large, partitioned datasets. Such scenarios require the evaluation of dataset-level quality constraints after individual partition updates, without having to re-read already processed partitions. Additionally, such scenarios often require the verification of data quality on select combinations of partitions.

We therefore present a differential generalization of the computational model of *Deequ*, based on algebraic states with monoid properties. We detail how to efficiently implement the corresponding operators and aggregation functions in Apache Spark. Furthermore, we show how to optimize the resulting workloads to minimize the required number of passes over the data, and empirically validate that our approach decreases the runtimes for updating data metrics under data changes and for different combinations of partitions.

I. Introduction

Data is a central resource for modern enterprises and institutions. Online retailers, for example, rely on data to support customers' buying decisions, to forecast demand [3] or to schedule deliveries. Any such decision can be seriously compromised by missing or incorrect information with detrimental impact on downstream processes [6], especially if it is automated with machine learning [2], [15]. A further complication is that many modern data sources such as key-value stores and distributed file systems do not support integrity contraints and quality checks, and often do not even provide a schema for the data.

In order to address this challenge, we have recently introduced *Deequ*[1], a Spark-based library for the automation of data quality verification at scale [14]. *Deequ* allows its users to define *unit tests for data* based on a declarative API, which combines common quality constraints with user-defined validation code. Many modern data pipelines, however, operate on partitioned and evolving datasets that pose particular opportunities and challenges for data quality verification. For instance, a dataset may consist of different partitions which are updated or replaced individually. In this case, we may want to evaluate dataset-level constraints without having to re-read already processed partitions. In another cases we might be interested in verifying the data quality of certain combinations of partitions, e.g., for particular timespans. Previous versions of *Deequ* do not support these scenarios, only the special case of strictly growing datasets.

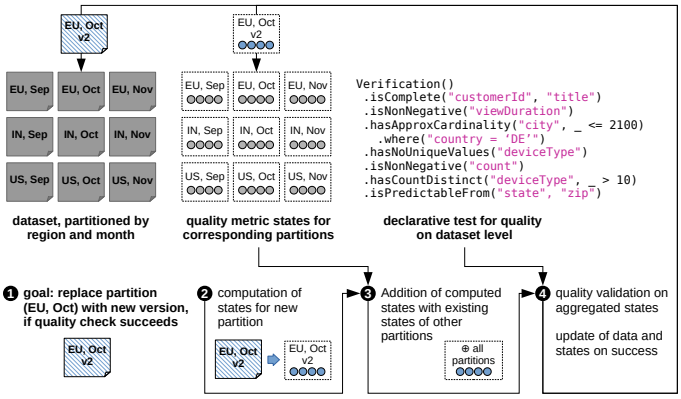

Fig. 1. Ingestion of an updated data partition for a partitioned dataset with differential data quality validation. *Deequ* evaluates the quality constraints using aggregated quality metric states and does not need to touch the existing data partitions.

In this paper, we address this gap and extend *Deequ*'s computational model by introducing states with algebraic monoid properties (Section II). This enables computation of metrics from combinations of partition states and flexible adaption to various scenarios of constraint evaluation on partitioned data.

Figure 1 illustrates how *Deequ* would execute differential quality validation in an exemplary scenario of ingesting an updated partition for a large dataset, which is partitioned by region and month. ❶ We receive a new version of the `(EU, Oct)` partition and want update our dataset, *under the condition that the quality test for the dataset as a whole still succeeds*. If this were not the case, we would reject the update. *Deequ* allows us to efficiently execute the validation as follows: ❷ we compute states for the metrics of the test from the new partition, and ❸ aggregate them with the existing states for the remaining partitions (which we receive from a state store). ❹ Afterwards, we execute the data quality validation on the aggregated states, and ingest the new partition in case of success. Note that this process is efficient, as we do

[1]https://github.com/awslabs/deequ

not need to touch the remaining data partitions, and the states are typically small and fast to aggregate.

We show how to implement the proposed abstraction efficiently and detail how to generate aggregation queries in SQL for our stateful metrics computation. We describe how to execute the resulting queries with a highly reduced number of passes over the data compared to naive execution (Section III). Finally, we experimentally show that our proposed extensions give rise to large savings in computational cost (Section V).

In summary, we provide the following contributions:

- We present a formal algebraic computation model for differential metrics computation (Section II).
- We outline an algorithm for the optimized simultaneous computation of a large set of data quality metrics (Section III).
- We provide an experimental evaluation showcasing the reduction in the number of Spark jobs by our execution algorithm and the decrease in runtime for computing metrics on growing datsets, for updating dataset metrics under data changes, and for computing metrics on different combinations of partitions (Section V).

II. Computational Model

In the following, we detail the proposed extension to our computational model. The basic operation in *Deequ* is to compute a metric (typically a single real number) from data $d \in \mathcal{D}$ of a specific domain $\mathcal{D}$ (e.g., rows in a dataframe with a given schema) and allow users to define constraints on this metric (such as a threshold for the ratio of missing values in a column of the data). In *Deequ*, so-called *analyzers* allow users to compute these metrics, where a single analyzer is in general modeled as a function $\mathcal{D} \to \mathbb{R}$. As already discussed in the introduction, this computational model is unfortunately not flexible enough to handle various common scenarios: (i) we might encounter data $d^{(t+1)} = d^{(t)} \cup \Delta d$ that grows over time, where a set of new records Δd is regularly appended (e.g., log events of production systems)[2]. In such cases we want to update the metrics for the data based on the small delta Δd and not have to re-scan the previous data $d^{(t)}$; (ii) the second scenario is when data is partitioned and individual partitions get replaced, e.g. a table $d = \bigcup_p d_p$ with p partitions (e.g., log data partitioned by country) where the data for a partition is updated atomically. Again, we do not want to have to re-scan the whole dataset to update its metrics, but have our computation work with the data d_j of a particular partition j; (iii) a third scenario is when we only want to compute the metrics for a particular subset of the partitions of a dataset.

Introduction of algebraic states. In order to gain the required flexibility to tackle the mentioned scenarios, we enhance the computational model underlying *Deequ*'s analyzers. We redesign the analyzer to provide a function $s : \mathcal{D} \to \mathcal{S}$ to compute a state (sufficient statistics) of a domain $\mathcal{S}$ for a metric from data, and a second function $m : \mathcal{S} \to \mathbb{R}$ to compute the actual metric from the state. Given data $d \in \mathcal{D}$, we compute a metric by composing s and m as $m(s(d))$. Additionally, we introduce a generalized addition function $\oplus : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ for the states of a metric, and enforce the algebraic structure of a *commutative monoid* on $(\mathcal{S}, \oplus, \mathbb{0})$. That means that the generalized addition must be associative: $\forall d_1, d_2, d_3 \in D : [s(d_1) \oplus s(d_2)] \oplus s(d_3) = s(d_1) \oplus [s(d_2) \oplus s(d_3)]$. Furthermore, we require the existence of an identity element $\mathbb{0}$ (the state of an empty set), such that $\forall d \in D : \; s(d) \oplus \mathbb{0} = \mathbb{0} \oplus s(d) = s(d)$ holds, and we enforce commutativity: $\forall d_1, d_2 \in D : s(d_1) \oplus s(d_2) = s(d_2) \oplus s(d_1)$. These algebraic properties allow us to implement the state computations as parallel aggregations in in Apache Spark: associativity and commutativity enable parallel execution of the computations [4] and the identity element allows us to handle edge cases like columns which only consist of NULL values. Finally, we require the property that state computation and generalized addition distribute over the union of disjoint sets of data tuples: $\forall d_1, d_2 \in D, d_1 \cap d_2 = \emptyset : s(d_1 \cup d_2) = s(d_1) \oplus s(d_2)$. This allows us to compute metrics from aggregated states of parts of the data instead of the dataset as a whole, which is the basic requirement that we need to tackle our use cases: $m(s(d_1 \cup d_2)) = m(s(d_1) \oplus s(d_2))$.

Examples. The *completeness* metric denotes the ratio of non-missing values in a column. The corresponding monoid is $(\mathbb{N}^2, +, [0, 0])$, states are 2-tuples from a subset of $\mathbb{N}^2$ which represent the number of non-null values in a column as well as the overall number of rows. Merging two of these states requires simple addition and the identity element is the zero tuple. Another example is estimating the cardinality of a column using hyperloglog (HLL) sketches [10]. The monoid for this metric is $(\{0,1\}^d, \vee, [0, \ldots, 0])$; states correspond to the d-dimensional bitstring in which the sketch stores its counters, addition is conducted by logical disjunction of two such bitstrings and the identity element is the zero string. Another simple example is the *Maximum* metric which denotes the maximum value in a numeric column and uses the $(\mathbb{R}, max, -\infty)$ monoid: states are real numbers, the addition of two states reduces the computation of the maximum between the two values and the identity element is represented by negative infinity. We provide a list of the metrics supported by our library and their corresponding states and addition functions in Table I.

In the following, we elaborate how these algebraic properties enable the previously mentioned use case scenarios.

Scenario 1: Append-only growth of a table. This lets us easily handle the case of incrementally growing data $d^{(t+1)} = d^{(t)} \cup \Delta d$ where we already have access to the state $s(d^{(t)})$ from the previous version t. We compute the state $s(\Delta d)$ for the new records and merge it with the existing state $s(d^{(t)})$ without needing to re-scan $d^{(t)}$ as follows $m(s(d^{(t+1)})) = m(s(d^{(t)} \cup \Delta d)) = m(s(d^{(t)}) \oplus s(\Delta d))$.

Scenario 2: Partition replacement. For a partitioned table $d = \bigcup_p d_p$ with p partitions, where we updated a single partition d_j, we recompute the metrics by aggregating the states of the remaining partitions with the newly computed state $s(d_j)$

[2]We proposed a non-general incremental extension for this case in [14], which is only a special case subsumed by the model outlined in this work.

TABLE I
SUPPORTED DIFFERENTIAL ANALYZERS WITH CORRESPONDING TYPE OF STATE AND ADDITION FUNCTION FOR MERGING STATES.

Analyzer	Semantic	State	Addition function
Analyzers that only require a shareable scan of the input data			
ApproxCountDistinct	hyperloglog (HLL) cardinality estimate	HLL sketch [10]	HLL sketch merge
ApproxQuantile	approximate quantile of column values	quantile sketch [9]	quantile sketch merge
Completeness	fraction of non-missing values in a column	(#non-null values, #rows)	element-wise addition
Compliance	ratio of values matching a predicate	(#matches, #rows)	element-wise addition
Correlation	correlation between two columns	online covariance statistics [5]	online covariance update
DataType	data type inference for a column	type match counts	element-wise addition
Maximum	maximal value in a column	current maximum	max()
Mean	mean value in a column	(sum, #rows)	element-wise addition
Minimum	minimal value in a column	current minimum	min()
PatternMatch	ratio of values matching a regex	(#matches, #rows)	element-wise addition
Size	number of records	current size	addition
StandardDeviation	standard deviation of column values	1st and 2nd central moment	online update of moments [12]
Analyzers that require a re-partitioning of the input data			
CountDistinct	number of distinct values in a column	(value frequencies, #rows)	outer join + summation
Distinctness	unique row ratio in a column	(value frequencies, #rows)	outer join + summation
Entropy	entropy of the value distribution	(value frequencies, #rows)	outer join + summation
MutualInformation	mutual information between two columns	(value frequencies, #rows)	outer join + summation
Uniqueness	unique value ratio in a column	(value frequencies, #rows)	outer join + summation
UniqueValueRatio	ratio of unique values to cardinality	value frequencies	outer join + summation

for partition j via $m(s(d)) = m\left(s(d_j) \oplus [\bigoplus_{p \neq j} s(d_p)]\right)$.

Scenario 3: Different 'views' of a dataset We might want to run tests on different combinations of partitions of a dataset. An example would be a case where the data is partitioned by days D and regions R and we are interested in the metrics for the day *'2018/08/12'* in all regions $m\left(\bigoplus_{r \in R} s(d_{\{\text{'2018/08/12'},r\}})\right)$ or the metrics for a week in a particular region, e.g., *'EU'* as $m\left(\bigoplus_{d \in D} s(d_{\{d,\text{'EU'}\}})\right)$.

III. IMPLEMENTATION

We introduce our Scala-based implementation of the proposed computational model. At the heart of the implementation are *analyzers*: operators that compute *metrics*. We first describe their design and later detail how we execute them efficiently. As required by our computational model, every analyzer has a corresponding state with the generalized addition function $\oplus$ for other states of the same type:

```
trait State[S <: State[S]] {
 def +(other: S): S /* Addition operation */
}
```

The analyzer itself then defines the function $s : D \rightarrow \mathcal{S}$ which computes a state from a Spark dataframe as `computeStateFrom` and the function $m : \mathcal{S} \rightarrow \mathbb{R}$ which computes the final numeric metric from the state as `computeMetricFrom`. The identity element $\mathbb{0}$ is represented by the `None` variant of Scala's `Option[S]`.

```
trait Analyzer[S <: State[S], +M <: Metric[_]] {
 /* Compute the state from a partition */
 def computeStateFrom(part: DataFrame): Option[S]
 /* Compute the metric from the state */
 def computeMetricFrom(state: Option[S]): M
}
```

While the implementation presented so far would be sufficient to cover our computational model, it would result in suboptimal execution behavior, as every analyzer would have to conduct a pass over the data in its `computeStateFrom` method. We designed a special implementation for analyzers that compute simple aggregations over the data, which can share scans over the data with other analyzers of this type:

```
trait ScanShareableAnalyzer[S <: State[S],
 +M <: Metric[_]] extends Analyzer[S, M] {
 /* Aggregations to compute on the data */
 def aggregationFunctions(): Seq[Column]
 /* Compute state from the aggregation result */
 def fromAgg(result: Row, offset: Int): Option[S]
}
```

The idea here is that the analyzers generate aggregation functions in `aggregationFunctions` (instead of accessing the data directly). These aggregation functions will be executed by our system in an efficient manner. Each analyzer constructs its state from the result of the aggregation function via `fromAgg` afterwards. *Example*. We provide a concrete example of how a metric can be computed based on this design. We choose the *completeness* metric, which is defined as the ratio of non-null values in a column (e.g., a completeness of 1.0 implies no missing values). The state `CState` corresponding to the metric lives in a subset of $\mathbb{N}^2$; it consists of the number of non-null values in the investigated data as well the overall number of rows. Adding states of this type only requires us to add the respective counts.

```
case class CState(nonNull: Long, count: Long)
  extends State[CState] {
 def +(o: CState): CState =
  CState(nonNull + o.nonNull, count + o.count)
}
```

Algorithm 1: Optimized execution of a set of analyzers with potentially pre-existing state to aggregate.

```
   /* Precompute aggregations for shareable
      analyzers that do not require grouping */
1  F ← ∅
2  for analyzer a ∈ A with grouping_cols(a) = ∅ :
3    if can_share_scans(a) :
4      F ← F ∪ aggregation functions defined by a
5  R ← compute aggregations F on data d
   /* Compute states and metrics for all analyzers
      that do not require grouping */
6  for analyzer a ∈ A with grouping_cols(a) = ∅ :
7    if can_share_scans(a) :
8      s_a ← compute state for a from R
9    else:
10     s_a ← compute state on data d
11   ŝ_a ← previous state for a or 𝕆
12   m_a ← compute metric from ŝ_a ⊕ s_a
13   add (a, s_a, m_a) to result
   /* States and metrics for all analyzers that
      require a particular grouping of the data */
14 G ← set of all sets of grouping columns from A
15 for grouping columns g ∈ G :
16   d_g ← group data by g and count values
     /* Aggregations for shareable analyzers */
17   F_g ← ∅
18   for analyzer a ∈ A with grouping_cols(a) = g :
19     if can_share_scans(a) :
20       F_g ← F_g ∪ aggregation functions defined by a
21   R_g ← compute aggregations F_g on grouped data d_g
22   for analyzer a ∈ A with grouping_cols(a) = g :
23     if can_share_scans(a) :
24       s_a ← compute state for a from R_g
25     else:
26       s_a ← compute state on grouped data d_g
27     ŝ_a ← previous state for a or 𝕆
28     m_a ← compute metric from ŝ_a ⊕ s_a
29     add (a, s_a, m_a) to result
```

The following code depicts the actual implementation of the analyzer. This analyzer can be shared within scans and therefore has to specify its required aggregation functions. In the first aggregation, we cast the column values to 1 or 0 depending on whether they are NULL or not, and sum these up, which gives us the number of non-null values in the column to investigate. The second aggregation function simply counts the number of rows. In the `fromAgg` method, we pick the results of the aggregrations from the overall aggregation result and return a `CState`. Finally, we can compute the completeness metric by dividing the number of non-null values by the overall number of rows.

```
case class Completeness(column: String) extends
 ScanShareableAnalyzer[CState] {

 def aggregationFunctions(): Seq[Column] =
  sum(col(column).isNotNull.cast(LongType)) ::
  count(column) :: Nil

 def fromAgg(r: Row, offset: Int): Option[CState] =
  CState(r.long(offset), r.long(offset + 1))

 def computeMetricFrom(state: Option[CState])
  : Option[DoubleMetric] =
  state.map { cState => DoubleMetric(
   cState.numNonNull.toDouble / cState.count) }
}
```

This implementation pattern works for a large number of analyzers which only require the execution of a set of aggregation functions over the data (even for more exotic cases such as approximate cardinality estimation with hyperloglog sketches [10]). However, a special case are analyzers which require to group the data by particular columns (e.g., to compute the unique value ratio of a column). The state here corresponds to a vector of cardinalities of the contained values and the addition function needs to be able to merge such vectors via outer joins. These joins are costly in presence of a large number of distinct values. The metrics for which this is required a shown in the lower part of Table I.

Algorithm for Optimized Execution. Enabling scan-sharing for the analyzers provides us with the building blocks for optimizing the execution of a set of analyzers. The resulting algorithm is depicted in Algorithm 1 and attempts to minimize the number of passes over the data[3]. The algorithm is given a set of analyzers to execute and schedules them as follows: First, we focus on the subset of analyzers which does not require us to group the data. We collect the aggregation functions from all such analyzers that support scan sharing (line 4) and combine these into a single query which we execute in line 5. Next we compute the states from either the pre-computed aggregation result (line 8) or from the data (in line 10) in case the analyzer does not support scan sharing. Finally, we load and add potentially existing states and compute the final metric for the analyzer in lines 11 to 13. Next, we identify all required sets of grouping columns for the remaining analyzers and repeat the execution scheme for all analyzers that require the same grouping columns (lines 14 to 29). A difference is that we first need to execute a group by query and count the resulting values in line 16 before we execute the analyzers, where we again apply scan sharing.

IV. Related Work

Declarative definitions of data quality standards are well-established [6]–[8], and an integral part of every database management systen in the form of *integrity constraints*. Our differential approach is related to view maintenance in data warehouses [1] and builds upon well understood algebraic properties for aggregation functions [4], [11]. Domain-specific sanity checks and explicit data validation components are

[3]Implementation available at https://goo.gl/Le1H2n.

actively researched as well in the upcoming field of data management for machine learning [2], [3], [13], [15].

V. Experimental Evaluation

We evaluate *Deequ* on a dataset of 54,504,410 reddit comments with 22 attributes from May 2015 obtained from Kaggle[4]. We partition the data into 14 different partitions by the week day of the comment creation as well as by the binary `controversiality` attribute which indicates whether a particular comment was controversial.

We profile the data and generate two tests. The first *basic test* asserts that the columns `created_utc`, `week_day`, `ups`, `downs`, `id`, `name`, `subreddit_id`, `link_id`, `subreddit`, `author`, `controversiality` and `parent_id` are complete and that `removal_reason` has less than 5% non-null values. Additionaly, it checks that `created_utc`, `week_day`, `ups`, `downs` and `controversiality` are integer typed values, evaluates a range constraint on `week_day` and `controversiality`, and tests whether the approximate cardinality of `subreddit` and `subreddit_id` is within the approximation error of the true cardinality. Finally, it evaluates a constraint stating that the 90th percentile of `ups` must be less than 10. Note that all these constraints can be evaluated in a single scan over the data. The second *advanced test* is an extended version of the basic test and evaluates additional constraints on the unique value ratio and exact cardinality of `subreddit` and `subreddit_id` as well as the cardinality of their combination. Note that the advanced test requires us to group the data multiple times.

We execute our evaluation on an Elastic MapReduce cluster in AWS with 4 workers (*c4.xlarge* instances) running Apache Spark 2.2 and HDFS 2.7.3 using the *emr 5.8.2* profile.

Benefits of Optimized Execution. We first evaluate the benefits of our proposed algorithm for executing a large number of analyzers (Algorithm 1). We execute both the basic test and the advanced test with and without the optimized execution and repeat this for cached inputs. We measure the number of Spark jobs and stages scheduled and depict the results in Figure 2. We observe the expected reduction in the number of passes over the data (due to scan sharing) compared to the naive execution. We encounter a ten-fold reduction for the basic test (which does not require a grouping of the data) and a factor of four less jobs for the advanced test which groups the data several times. We also observe a huge reduction in the number of corresponding Spark stages which is larger in cases where the data is not cached (which we attribute to the fact that Spark's query optimizer has better statistics about the data when it has been cached).

Benefits of Differential Computation. In the next set of experiments, we evaluate differential computation in the three different scenarios outlined in Section II: append-only growth of a dataset, replacement of partitions in a partitioned dataset

[4]https://www.kaggle.com/reddit/reddit-comments-may-2015

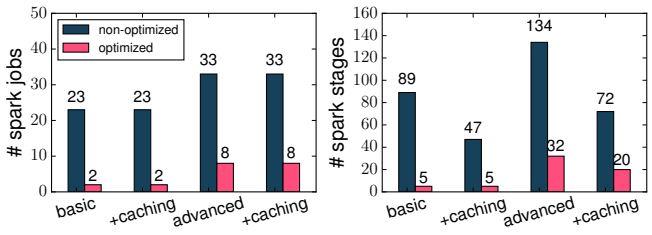

Fig. 2. Reduction in the number of Spark jobs and stages by our optimized execution in comparison to naive sequential execution.

and metrics for different combinations of a partitioned table. We run the basic and advanced test for every scenario, and compare differential computation (with states stored on HDFS) to recomputation from scratch (both with optimized execution activated). We repeat every experiment 5 times and report the mean runtime.

Scenario 1: Append only. In the first experiment, we mimick a growing dataset by adding the two partitions for each weekday in every run. We compare the differential computation which updates a persisted state based on the new partition to the full recomputation of all metrics for every version of the dataset[5]. Figure 3 illustrates the resulting runtimes. We observe the expected behavior: In case of full recomputation, the runtime grows linearly with the size of the dataset, while the runtime is roughly constant for the differential computation, where we operate on a constantly sized new partition in each run.

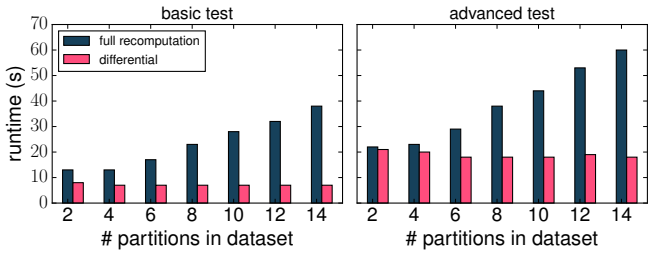

Fig. 3. Constant runtime for differential computation on growing data (compared to linear growth with the data size in the case of full recomputation).

Scenario 2: Partition replacement. In this set of experiments, we evaluate a scenario where partitions for a particular weekday change. Without a differential approach, we need to scan the whole table to re-compute its metrics. In the case of differential computation, we compute states per partition once in the beginning, and only have to re-compute the states for the changed partition; afterwards we can cheaply re-compute the metrics for the whole table from the aggregated states. The initial computation of the states in the differential case takes 55 seconds for the basic test and 121 seconds for the advanced

[5]We presented a similar experiment in our previous work [14], which we repeat here to showcase that our new algebraic model subsumes the special case formulation provided previously.

test. The resulting runtimes for the updates are shown in Figure 4. We see that the differential approach can quickly amortize the overhead of the initial computation by enabling cheap updates: Each update can be computed in a fraction of the time it takes to recompute the metrics for the whole table. For the basic test this fraction is about one quarter, while it is about a third in the case of the advanced test. The higher cost for the latter is due to the joins required to merge the states.

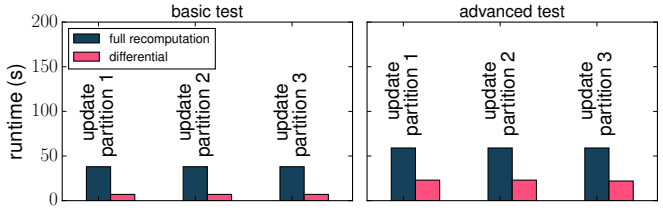

Fig. 4. Reduction in runtime of the re-computation of dataset metrics after a partition has changed.

Scenario 3: Different 'views' of a dataset. In our final experiment, we execute the tests on four different combinations of partitions: (a) we first evaluate them for each weekday individually, (b) then or all non-controversial comments, and afterwards for two weekday-controversiality combinations ((c) and (d)). We compare full recomputation against the differential approach, which computes states for all partitions once and aggregates the partitions states for the combinations afterwards. The initial computation takes 64 seconds for the basic test and 159 seconds for the advanced test. The results in Figure 5 show that, while the initial computation of the states per partition is costly, it enables us to afterwards execute tests on combinations of partitions almost instantly (e.g., in less than a second for the basic test).

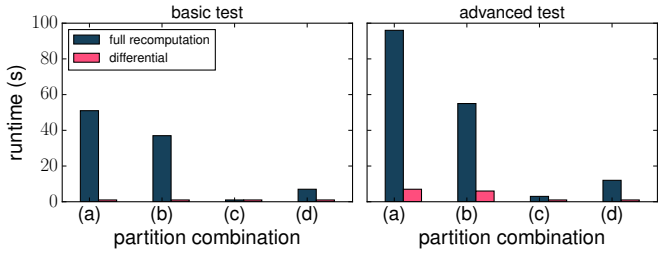

Fig. 5. Runtime for test execution on several different combinations of partitions for differential and full-recomputation.

We conclude that the differential approach enables a different asymptotical behavior in the case of growing data, where its runtime scales with the size of the additional delta (and not the overall data size). For cases where partitions change or where we are interested in a certain combination of partitions, the differential approach enables a trade-off for users. While it requires computation of the individual states for all partitions once, it subsequently enables much faster tests: We observed a runtime decrease by a factor of four and three in the case where a partition changed and we encountered near instant computation of metrics on selected combinations of partitions.

VI. Conclusion

We presented a formal algebraic computation model for differential metrics computation based on monoid properties. We described how to implement common data quality metrics in this model and outlined an algorithm for the optimized execution of a large set of data quality operators, while still enabling the aggregation of existing states. In an experimental evaluation, we showcased a reduction in the number of Spark jobs by our execution algorithm and a decrease in runtime for computing metrics on growing datasets, for updating table metrics under data changes and for computing metrics on different combinations of partitions. A limitation of our approach is that it only operates on disjoint partitions and therefore cannot handle cases like upserts of partitions. This would however require us to track states on a record level rather than a partition-level, which incurs substantial overhead.

In future work, we aim to investigate additional improvements for the optimized execution of complex metrics computation, e.g., to re-use intermediate results and to make automated decisions on materializing grouping results. Additionally, we aim to apply our differential approach to the generation of training data for anomaly detection algorithms.

References

[1] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. In *SIGMOD Record*, volume 26, pp. 417–427. ACM, 1997.

[2] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, V. Jain, L. Koc, et al. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. *KDD*, pp. 1387–1395, 2017.

[3] J.-H. Böse, V. Flunkert, J. Gasthaus, T. Januschowski, D. Lange, D. Salinas, S. Schelter, M. Seeger, and Y. Wang. Probabilistic demand forecasting at scale. *PVLDB*, 10(12):1694–1705, 2017.

[4] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin. Summingbird: A framework for integrating batch and online mapreduce computations. *PVLDB*, 7(13):1441–1451, 2014.

[5] T. F. Chan, G. H. Golub, and R. J. LeVeque. Algorithms for computing the sample variance: Analysis and recommendations. *The American Statistician*, 37(3):242–247, 1983.

[6] X. Chu, I. F. Ilyas, S. Krishnan, and J. Wang. Data cleaning: Overview and emerging challenges. *SIGMOD*, pp. 2201–2206, 2016.

[7] X. Chu, I. F. Ilyas, and P. Papotti. Discovering denial constraints. *PVLDB*, 6(13):1498–1509, 2013.

[8] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.-A. Saita. Declarative data cleaning: Language, model, and algorithms. *VLDB*, pp. 371–380, 2001.

[9] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. *SIGMOD Record*, volume 30, pp. 58–66, 2001.

[10] S. Heule, M. Nunkesser, and A. Hall. Hyperloglog in practice. *EDBT*, pp. 683–692, 2013.

[11] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. *CIDR*, 2013.

[12] X. Meng. Simpler online updates for arbitrary-order central moments. *arXiv:1510.04923*, 2015.

[13] S. Schelter, J.-H. Böse, J. Kirschnick, T. Klein, and S. Seufert. Automatically tracking metadata and provenance of machine learning experiments. *Machine Learning Systems Workshop at NIPS*, 2017.

[14] S. Schelter, D. Lange, P. Schmidt, M. Celikel, F. Biessmann, and A. Grafberger. Automating large-scale data quality verification. *PVLDB*, 11(12), 2018.

[15] M. Terry, D. Sculley, and N. Hynes. The Data Linter: Lightweight, Automated Sanity Checking for ML Data Sets. *Machine Learning Systems Workshop at NIPS*, 2017.