Text Reuse In Large Historical Corpora: Insights from the Optimization of a Data Science System [Scalable Data Science]

Ananth Mahadevan, Michael Mathioudakis, Eetu Mäkelä and Mikko Tolonen
University of Helsinki, Finland

ABSTRACT

Text reuse is of fundamental importance in humanities research, as similar pieces of text in different documents provide invaluable information about the historical spread and evolution of ideas. Traditionally, scholars have studied text reuse at a very small scale, for example, when comparing the writings of two philosophers; however, modern digitized corpora spanning entire centuries promise to revolutionize humanities research through the detection of previously unobserved large-scale patterns.

This paper presents insights from ReceptionReader, a system for large-scale text reuse analysis over almost all known 18th-century books, articles, and newspapers. The system implements a data management pipeline for billions of text reuse instances and supports analysis tasks based on database queries (e.g., retrieving the most reused quotes from queried documents).

The paper describes the data management considerations that led from an originally *deployed* to an extensively *evaluated* and substantially optimized version of the system, as the performance of different normalization levels and query execution engines was evaluated for various queries of interest. The paper offers insights from the observed trade-offs and how they were resolved to fit our requirements. In summary, the paper explains how, for our system, (1) the row store engine (MariaDB Aria) emerged as the overall optimal choice for query processing, while (2) big data processing (Apache Spark) proved irreplaceable for preprocessing.

PVLDB Reference Format:
Ananth Mahadevan, Michael Mathioudakis, Eetu Mäkelä and Mikko Tolonen. Text Reuse In Large Historical Corpora: Insights from the Optimization of a Data Science System [Scalable Data Science]. PVLDB, 17(1): XXX-XXX, 2023.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://version.helsinki.fi/ads/evaluating-receptionreader.

1 INTRODUCTION

Text reuse is an essential methodological element in humanities research, which involves the detection and analysis of similar pieces of text across different documents. Researchers use it to trace the evolution and influence of ideas, quantify the impact of specific authors and literary works, and identify context crucial for the meaning of a text [20]. Traditionally, text reuse has been considered on a small scale, for example, when comparing the writings of two philosophers. However, large-scale digitized historical corpora spanning centuries are transforming this field, as they enable the automatic detection and analysis of text reuse, uncovering large-scale patterns previously indiscernible with smaller-scale analysis.

This paper presents insights from ReceptionReader[1][17], a system for large-scale text reuse analysis developed by a multidisciplinary team of experts in history, NLP, and data science that has already pioneered intellectual history research [18]. ReceptionReader is built upon billions of text reuse instances identified over almost all known 18th-century documents (Section 2), and provides two main downstream analysis tasks: **reception**, asking for all the text reuse instances stemming from a particular document (Section 3.3.1); and **top quotes**, asking for the most reused quotes stemming from a queried set of documents (Section 3.3.2).

In more detail, the paper presents the data management considerations that led from an originally *deployed* but unoptimized version to an extensively *evaluated*, substantially optimized, and soon-to-be-deployed version of the system. The optimization targets system performance metrics (Section 3), such as query latency, storage size (disk space to materialize the database and related indexes), and computing costs (billing expenses for cloud computing).

The paper dives into design choices that raise significant trade-offs in the performance metrics of interest and the considerations that led to optimal choices for the system at hand. In particular, we consider three levels of database normalization: fully normalized (referred to as Standard), task-agnostic normalized (Intermediate), and task-specific denormalized (Denormalized); as well as three query execution frameworks: big-data processing (Apache Spark), indexed row store database (MariaDB Aria), and compressed column store database (MariaDB Columnstore).

We evaluate each design choice over various queries for each downstream task and study the trade-offs that arise (Section 4). First, database normalization raises a trade-off between storage size and query latency. For example, Denormalized data lead to minimal query latency at the expense of increased storage size, especially for the reception task. Second, the query execution frameworks raise their own trade-offs. For example, Columnstore has a smaller storage size than Aria for the same level of normalization at the expense of higher query latency, while Spark exhibits the highest query latency but is the only framework to handle efficiently heavy pre-processing tasks. Considering the computing costs and application constraints for our setting (e.g., what query latency or billing budget is deemed acceptable), we navigate the observed trade-offs and arrive at a chosen multi-modal design, with Denormalized data in Aria as an optimal arrangement at the user-end of the pipeline, but Spark as an irreplaceable back-end for pre-processing.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

[1] See https://receptionreader.com/ for the deployed version of the system.

2 BACKGROUND

We begin with a brief presentation of the corpora (Section 2.1), text reuse detection algorithm (Section 2.2), and pre-processing pipeline (Section 2.3) of ReceptionReader, which are considered fixed for the evaluation presented later in the paper.

2.1 Historical Text Corpora

The system hosts the digitized corpora shown in Table 1. ECCO (Eighteenth Century Collections Online) [8, 9] and EEBO-TCP (Early English Books Online Text Creation Partnership) [13] contain books in english, while Newspapers (British Library Newspapers) [7] contains newspaper articles. Most documents were published in the 17th and 18th centuries. The collections were digitized with OCR (optical character recognition) by Christy et al. [5]. While of immense value, the raw OCR'ed texts are noisy, partly due to archaic fonts and layouts [10, 24], leading to varying OCR quality [12].

Table 1: Document collections in the historical text corpora. Document length is measured in number of characters.

Collection	Publication Years	# Docs	Avg. Doc Length
ECCO	1505-1839	207613	295641
EEBO-TCP	1473-1865	60327	18148
Newspapers	1604-1804	1769266	9519

2.2 Text Reuse Detection

For the purposes of this paper, *any pair of similar pieces of text from different documents* defines one instance of **text reuse**. Each *piece* is identified by a (document ID, start-offset, end-offset) tuple, and similarity is controlled via thresholds for length and edit distance determined during an earlier development phase of ReceptionReader [21, 23]. To provide some background, text reuse instances are identified as described in Vesanto [21], using BLAST (Basic Local Alignment Search Tool) [2], a fuzzy-string-matching algorithm that is widely used for DNA sequence matching and has proved to be more effective than alternatives in finding text reuses with noisy OCR data [23]. Essentially, BLAST is employed across the corpora in an all-to-all document comparison and outputs '**hits**' of similar piece pairs as instances of text reuse[2].

2.3 Pre-processing Pipeline

To become useful for downstream tasks, the detected text reuses are pre-processed in a three-phase pipeline (Figure 1).

Phase 1: Defragmentation. When a text passage is reused in several documents, its offsets might differ slightly across BLAST hits due to OCR noise (overlapping red boxes in documents B and D in Figure 1). This phenomenon is known as *fragmentation* and leads to downstream issues, either due to redundancy (many overlapping pieces referring to the same text) or loss (pieces shorter than the full passage). As remedy, ReceptionReader merges overlapping pieces of similar lengths within a document, resulting in new *defragmented pieces* and *defragmented reuses* (blue boxes and lines in Figure 1).

[2]In what follows, 'instance(s) of text reuse' will simply be referred to as 'text reuse(s)'.

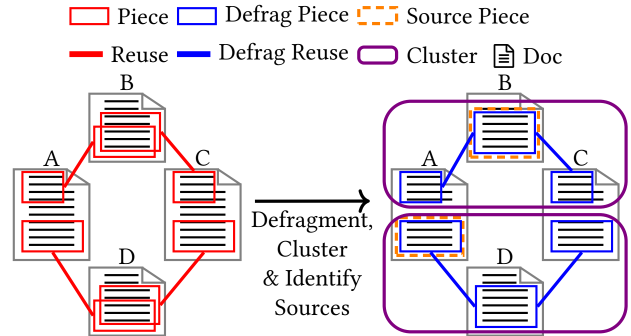

Figure 1: Pre-processing: BLAST hits (red) are defragmented (blue) and clustered (purple). In each cluster, the earliest piece is identified as the 'source' (dashed yellow).

Phase 2: Clustering. To identify how text is reused across multiple documents, ReceptionReader builds a network with defragmented pieces as nodes and text reuse pairs as undirected edges, and identifies node clusters in it, using a label-propagation clustering algorithm (specifically, Chinese Whispers [3]). The resulting clusters are groups of similar defragmented pieces from different documents, depicted with rounded purple rectangles in Figure 1.

Phase 3: Source identification. Within a cluster, the piece associated with the earliest publication date is of particular significance for historical analysis and labeled as the 'source' of text reuse (dashed yellow rectangles in Figure 1). Each other piece in the cluster is correspondingly labeled as a 'destination'.

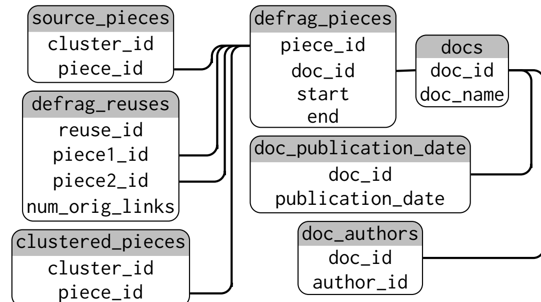

Figure 2: Simplified schema for pre-processed data. Relations are materialized for the Standard normalization level.

The pre-processing pipeline is considered *fixed* for this paper, as it was developed over years of field expertise by our team. It is implemented in Apache Spark, scales to billions of BLAST hits, and outputs Spark data frames as Parquet files. We omit the full schema but use a simplified one in what follows (Figure 2).

3 EVALUATION SETUP

This section presents the datasets (Section 3.1), design choices (Section 3.2), and performance metrics (Section 3.4) involved in the evaluation, with the evaluation performed for two analysis tasks, namely *reception* and *top quotes* (Section 3.3). At a high level, the evaluation is organized as follows: First, for a given dataset, a database is organized according to some design choices, which entail the materialization of relations and indexes in a particular format

(e.g., relation tables and associated indexes); next, representative query workloads for the two analysis tasks are executed under specified design choices; and finally, performance metrics (e.g., query latency) are collected and used to evaluate the design choices.

3.1 Datasets

In evaluating system design choices, we are interested in good scalability, particularly as we anticipate that more digital corpora will be added later to the system. Towards that end, we evaluated design choices on datasets of varying sizes, with datasets defined as subsets of the collections described in Section 2.1. For the economy of presentation, this paper uses two datasets at the larger end of the spectrum (Table 2), and evaluates scalability based on how performance changes when a new corpus is added to the data.

Table 2: Text Reuse Datasets. Numbers rounded to 3 digits; K: thousand, M: million, B: billion.

Attribute	Basic	Extended
Collections	ECCO & EEBO-TCP	ECCO, EEBO-TCP & Newspapers
documents	267K	2.04M
BLAST hits	966M	6.31B
defragmented pieces	384M	1.10B
avg. defrag. piece length	748	461
defragmented reuses	965M	5.61B
clusters	50.3M	91.6M
authors[3]	46.0K	46.0K

Specifically, the first dataset, Basic, combines ECCO and EEBO-TCP: reuse detection (Section 2.2) returns nearly one billion BLAST hits for it, and pre-processing (Section 2.3) yields 384 million defragmented pieces in 50.3 million clusters. The second dataset, Extended, adds Newspapers: reuse detection returns 6.5× more hits compared to Basic, and pre-processing yields about 2.8× more defragmented pieces and 1.8× clusters.

3.2 Design choices

In optimizing the system, we encountered two choices that raised significant trade-offs that crucially affected performance: execution frameworks and normalization. Different execution frameworks offer different data storage formats (e.g., row-oriented vs column-oriented) and access methods (e.g., distributed reads vs indexed scans), leading to trade-offs between query latency and storage size. Moreover, normalization raises a similar trade-off: typically, on one end, full normalization minimizes redundancy and storage space but complicates queries and increases query latency; on the other end, denormalization (practically, combining data in fewer tables with more attributes) increases redundancy and storage requirements but simplifies queries and decreases latency.

While these trade-offs might be individually understood and anticipated (see Ramakrishnan and Gehrke [15]), what choices lead to their joint optimization is a non-obvious problem, as it also depends on the unique features of the application at hand (in this case, downstream analysis tasks on text reuse and corresponding user requirements) and the computational setting (e.g., the billing cost of the different options in a cloud environment). Therefore, this paper is an applied exercise on optimizing a real-world data science system, aiming to extract insights into how the different choices interact and lead to an optimized configuration.

[3]Same number of authors because Newspapers metadata do not contain authorship.

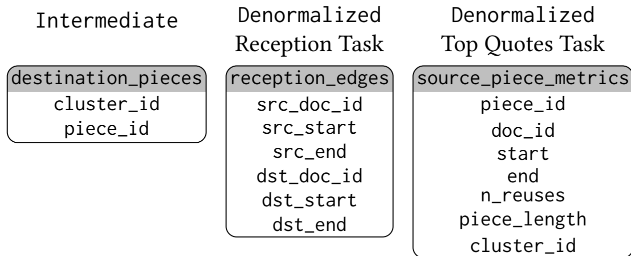

Figure 3: Additional relations materialized for the Intermediate and Denormalized levels

3.2.1 Query Execution Frameworks. Each framework has a unique format to store the data and a corresponding execution engine to answer downstream tasks expressed as SQL queries (Table 3). **Spark** is implemented as a Kubernetes cluster with 38 worker pods and one driver pod. The worker and driver pods each have two cores and 32 GiB RAM. Additionally, the driver pod has 16 GiB persistent storage for code and python environments. The text reuse data are stored as Parquet files on cloud data buckets and Spark loads the data as distributed dataframes to answer downstream queries. **Aria** and **Columnstore** are relational frameworks implemented as MariaDB databases on a cloud Virtual Machine (VM) with has 16 cores, 78 GiB RAM. A persistent cloud storage volume is attached to the VM and the text reuse data are loaded as relational tables. Additionally, we create B+-tree table indexes for Aria.

We use the cloud computing resources provided by CSC[4] to implement each framework. The resource costs for running the system in CSC measured in Billing Units (BU), reported in Table 3.

3.2.2 Normalization Levels. The evaluation includes the following options. **Standard:** Uses the fully normalized tables shown in Figure 2. Downstream analysis queries involve joining multiple tables and filtering for desired results. **Intermediate:** Materializes a task-agnostic relation containing destination pieces in each text reuse cluster shown in Figure 3. The key distinction from the Standard level is that this additional relation is materialized and used in downstream tasks. **Denormalized Level:** Materializes denormalized relations specific to each downstream task as shown in Figure 3. The task-specific denormalized tables are materialized by joining Standard level tables.

3.3 Analysis Tasks

Once design choices are specified according to the options outlined in Section 3.2 above, we consider system performance for two analysis tasks, namely *reception* and *top quotes*. These tasks are selected for the evaluation from field expertise, as they have proved to be central to the research of the historians in our team.

[4]https://research.csc.fi/cloud-computing

Table 3: Execution frameworks. Units for cost rates are CSC Billing Units (BU), tebibytes (TiB) and hours (hr).

Framework	Data Storage		Query Execution	
	Format	Cost Rate	Engine	Cost Rate
`Spark`	Parquet files	1 BU/hr/TiB	Apache Spark SQL	1254 BU/hr
`Aria`	Indexed Row-store tables	3.5 BU/hr/TiB	MariaDB Aria	24 BU/hr
`Columnstore`	Compressed Columnar tables	3.5 BU/hr/TiB	MariaDB ColumnStore	24 BU/hr

```sql
SELECT dp1.doc_id AS src_doc_id,dp2.doc_id AS dst_doc_id
dp1.start AS src_start, dp2.start AS dst_start,
dp1.end AS src_end, dp2.end AS dst_end
FROM source_pieces
INNER JOIN defrag_pieces dp1 USING(piece_id)
INNER JOIN destination_pieces dsp USING(cluster_id)
INNER JOIN defrag_pieces dp2 ON dsp.piece_id = dp2.piece_id
WHERE dp1.doc_id = D
```

SQL 1: Query for reception task with `Intermediate` level data

```sql
SELECT cluster_id, piece_id FROM source_pieces sp
RIGHT JOIN clustered_pieces cp USING(cluster_id,piece_id)
WHERE sp.piece_id IS NULL
```

SQL 2: Query to materialize the `destination_pieces` relation

```sql
SELECT re.* FROM reception_edges re WHERE src_doc_id = D
```

SQL 3: Query for reception task with `Denormalized` level data

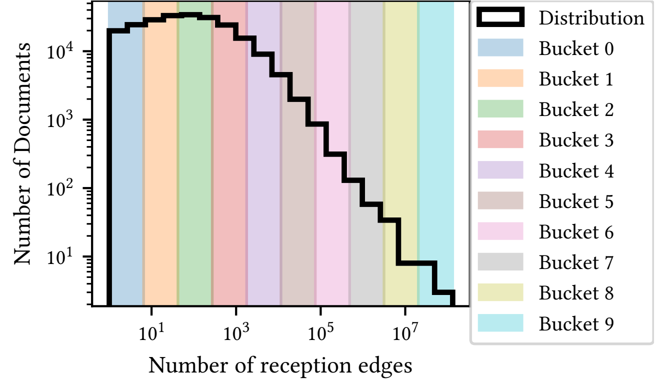

Figure 4: Number of reception edges distribution for the Basic dataset. The log-spaced workload buckets, shown as vertical bands, are used for sampling evaluation queries.

3.3.1 *Reception Task.* *Reception studies* are a sub-field of historical research, where historians study how a document was received (hence, *reception*) after its publication. These studies focus on how the text of a document is reused in other documents (e.g., in book reviews, journals, or newspapers) and analyze the context surrounding these reuses. Therefore, to aid historians with reception studies, our system gathers and displays all the text reuse instances originating from a given document D. We define the task of gathering these reuses as the *reception task*.

The logical steps to complete the reception task with the preprocessed data (Figure 2) are: (1) find all the clusters where the source piece is in D, (2) gather all the destination pieces in those clusters, and (3) create *reception edges* between each source piece and each destination piece in a cluster and return them as the result.

Following the above steps, SQL 1 shows the query for the `Intermediate` level which uses the materialized `destination_pieces` relation. For the `Standard` level, the only difference is that `destination_pieces` is computed using the subquery shown in SQL 2. For the `Denormalized` level, we materialize the `reception_edges` relation containing the reception edges for all documents and then filter for the given D as shown in SQL 3.

The latency of the reception task is largely determined by the number of reception edges for the given document D. The distribution of number of reception edges across documents in Basic dataset is shown in Figure 4, revealing a heavy-tail distribution. Therefore, to effectively evaluate the system for the reception task, we obtain a representative **workload** of queries by sampling 10 documents from each of the 10 log-spaced buckets (vertical bands in Figure 4). Documents in higher workload buckets have more reception edges, and the corresponding queries will have higher latency. This is used in Section 4 to dissect the trade-offs between different normalization levels and frameworks.

3.3.2 *Top Quotes Task.* Another common task for historians is to identify for further study the most influential quotes from a group of documents, with the document group typically belonging to the same author or same category according to some taxonomy. Towards this end, historians define the following: (a) **G** is a group of documents based on certain metadata attributes (e.g., all editions of a book by a specified author), (b) **quote** is a passage of text between 150 and 300 characters from a given document group G, and, (c) **`n_reuses`** for a specific quote is the number of unique documents of other authors that reuse it, and is used to quantify its influence. Based on these definitions, to identify the most influential quotes from a given document group G, we define the *top quotes task* as finding the top $k = 100$ quotes with the highest `n_reuses` from group G.

The query to complete the task with `Intermediate` level data is shown in SQL 4. At a high level, the query searches source pieces for quotes from G, computes `n_reuses` by filtering out destination documents from the same author as G, and returns the top $k = 100$ source pieces as the result. For the `Standard` level, the only difference again is that the `destination_pieces` relation in SQL 4 is computed with the subquery shown in SQL 2. For the `Denormalized` level, we materialize the `source_piece_metrics` relation which precomputes `n_reuses` for each source piece regardless of length. Then, the task query filters the relation for the given G and quote length as shown in SQL 5.

The latency for the top quotes task is largely determined by the `n_reuses` term a quote and the number of quotes in a given G. Therefore, to study the distribution of query latency, we use

```
WITH filtered_clusters AS (
  SELECT *, end - start AS piece_length FROM source_pieces
  INNER JOIN defrag_pieces USING(piece_id) WHERE doc_id
  IN G AND (end - start) BETWEEN 150 AND 300
),group_authors AS ( SELECT DISTINCT author_id
FROM doc_authors WHERE doc_id IN E AND author_id IS NOT NULL
), cluster_stats AS (SELECT cluster_id,
  COUNT(DISTINCT doc_id) as n_reuses FROM filtered_clusters
  INNER JOIN destination_pieces dsp USING (cluster_id)
  INNER JOIN defrag_pieces dp ON dsp.piece_id = dp.piece_id
  INNER JOIN doc_authors USING(doc_id)
  LEFT JOIN group_authors qa USING(author_id)
  WHERE qa.author_id_i IS NULL GROUP BY cluster_id
) SELECT * FROM cluster_stats INNER JOIN filtered_clusters
USING (cluster_id) ORDER BY n_reuses DESC LIMIT 100
```

SQL 4: Query for top quotes with `Intermediate` level data

```
SELECT * FROM source_piece_metrics WHERE piece_length BETWEEN
150 AND 300 AND doc_id IN G ORDER BY n_reuses DESC LIMIT 100
```

SQL 5: Query for top quotes with `Denormalized` level data

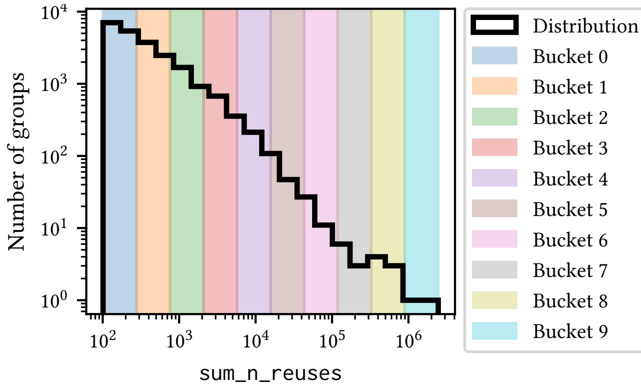

Figure 5: Distribution of `sum_n_reuses` for the Basic dataset. The log-spaced workload buckets, shown as vertical bands, are used for sampling evaluation queries.

`sum_n_reuses`, the sum of `n_reuses` for each quote in G. We compute `sum_n_reuses` for every group in Basic with at least 100 quotes and show the distribution in Figure 5, which, similar to the reception task, reveals a heavy-tail distribution. Here, a group is defined according to a commonly used literature taxonomy, with one group corresponding to books by the same author on the same topic [17]. To effectively evaluate the system for the top quotes task, we obtain a representative **workload** of queries by sampling one group from each of the 10 log-spaced buckets (vertical bands in Figure 4). Queries corresponding to higher workload buckets will have larger `sum_n_reuses` and hence a larger latency. Similarly, we sample 10 groups from the query distribution of the Extended dataset for evaluation.

3.4 Performance Metrics

The following performance metrics are of interest.

Query Latency is the time required to obtain and iterate through the result set for a query. To mitigate the effect of network latency, in the `Aria` and `Columnstore` frameworks, we execute the queries directly on the VM where the data is stored. A low query latency is desirable as it leads to better user experience.

Data Storage Size is the disk space used for the materialized tables necessary for a task, and varies based on the normalization level and execution framework. Smaller storage sizes are preferable due to reduced costs and improved scalability. In our experiments, we measure the storage size in TiB and calculate the total storage cost in BU/hr using the rates for each framework (Table 3).

Query Execution Cost The computational resources required to process a downstream query. We calculate it in BUs using the execution cost rates from Table 3 and the query latency. Lower query execution costs are beneficial, reducing overall system running costs and facilitating scalability to support more users.

4 RESULTS AND DISCUSSION

Our evaluation, outlined in section 3, focuses on two core design choices: normalization levels (`Standard`, `Intermediate`, `Denormalized`) and execution frameworks (`Spark`, `Aria`, `Columnstore`). We assess each combination of these choices based on query latency, data storage size, and execution costs, aiming to optimize system performance for downstream analysis tasks.

The evaluation process involves materializing relations for the chosen normalization level, loading data into the evaluated framework, and executing downstream tasks with queries sampled from workload buckets (see Figures 4 and 5). We set query timeouts of 5 and 15 minutes for the reception and top quotes tasks, respectively, and collect relevant evaluation metrics. The evaluation results are reported and discussed in the following sections: Section 4.1 discusses the practicalities of materializing and loading data. Section 4.2 presents the query latency results across buckets for each downstream task and discusses the impact of each design choice. Section 4.3 discusses each design choice's trade-offs between storage and execution costs.

4.1 Materialization and Loading

The `Intermediate` and `Denormalized` levels require additional relations (Figure 3) to be materialized before answering queries for downstream tasks. We attempted materializing these additional relations directly in each evaluated execution framework. However, in practice the `Denormalized` tables failed to materialize after 24 hours in the `Aria` framework, and the `Columnstore` framework crashed for both normalization levels due to a memory error from large table joins. Therefore, materializing directly in the `Aria` and `Columnstore` frameworks is infeasible. Consequently, we use the `Spark` framework to materialize the additional relations and bulk-load them into the relational frameworks.

For the Extended dataset, the `Spark` framework takes 2, 11 and 24 minutes to materialize the `destination_pieces`, `reception_edges` and `source_piece_metrics` relations, respectively. The number of rows in these materialized relations are shown in Table 4. Note that while the `source_piece_metrics` relation has fewer rows, materializing it takes the longest because the `n_reuses` term for every source piece is computed using a computationally expensive aggregation. In contrast, `Spark` is able to efficiently materialize the billions of `reception_edges` rows in a half that time.

Table 4: Number of rows in the additional materialized relations for the Intermediate and Denormalized levels

Materialized Relation		Dataset	
Name	Normalization	Basic	Extended
destination_pieces	Intermediate	324M	985M
reception_edges	Denormalized	1.28B	4.74B
source_piece_metrics	Denormalized	56.1M	90.1M

The bulk-loading the materialized relations into the relational frameworks is directly proportional to the number of rows reported in Table 4, taking 24 minutes, 2.5 hours and 3.5 minutes, respectively for the Extended dataset. Indexing the tables in Aria takes an additional 51 minutes, 4.3 hours and 7.1 minutes, respectively.

4.2 Latency results

The materialization and loading discussed in Section 4.1 are typically performed offline and therefore do not affect the end-users of the system. However, the queries for the downstream analysis tasks are executed online and the query latency plays an important role in determining the quality of the user experience. In this section, we report the latencies from executing queries sampled from different workload buckets for each downstream task in Figures 6 and 8. In each plot, the x-axis corresponds to the different buckets and the colors correspond to different normalization levels.

For both downstream tasks, the queries for the Standard level fails to complete in the Columnstore framework (Figures 6 and 8 right), even for the lowest workload bucket. This is because the Columnstore framework uses a slower disk-based hash join when an in-memory hash join cannot be performed. Therefore, when the destination_pieces relation is not materialized and has to be computed on-the-fly using several joins with large relations, the slower disk-based hash joins are used. The results in large latencies rendering the Columnstore framework infeasible with the Standard normalization level.

Now we share more insights from the task-specific results.

Reception Results. At the highest workload bucket, queries of all normalization levels fail to complete in both Columnstore and Aria (see Figure 6 left and middle). Specifically, the failure of the query for the Denormalized level indicates the relational frameworks are unable to fetch and iterate over the large result set within the time limit. On the other hand, the distributed Spark framework succeeds due to its ability to pre-fetch from different partitions and efficiently iterate over large result sets.

For low workload buckets (0, 1 and 2), with the smallest number of reception edges, the Aria framework with Denormalized data has the lowest query latency due to the indexed and materialized reception_edges relation. In comparison, the Columnstore and Spark frameworks with no indexed relations have query latency one and two orders of magnitude larger.

Interestingly, for high workload buckets, the query latency for the Intermediate and Denormalized levels are higher than that of the Standard level for the Aria framework (Figure 6(b) middle). These higher latencies were due to page faults from a cold system cache during evaluation. To remedy this, we re-ran each query twice, recorded the latency from the second run, and presented the hot-cache results for the Extended dataset in Figure 7. These results show that with a hot-cache, all queries have an overall lower query latency, and, the queries for the Intermediate and Denormalized levels are faster than that of the Standard level.

Top Quotes Results. Across all the frameworks in Figure 8, the Denormalized level has a significantly lower latency than the Standard and Intermediate levels. Specifically, in the Aria framework the Denormalized level on average has 3000× lower latency than the Intermediate level due to the materialized and indexed source_piece_metric relation with pre-computed n_reuses.

When we scale to the Extended dataset, previously successful Intermediate level and Standard level queries in the Aria framework fail at higher workload buckets (7, 8 and 9). This failure highlights the limitation of Aria which is not optimized for the aggregations queries required to compute n_reuses using larger relations. In contrast, the Intermediate level in Spark and Columnstore are successful and have consistent query latency across workload buckets due to their execution engines being optimized for such large-scale aggregation queries.

4.3 Data Storage and Execution Cost Results

Moving on from the latency which is important for end-users to billing metrics which are important for system maintainers. The size of each normalization level's materialized relations (Section 4.1) affect the storage costs and the query latency of the execution framework (Section 4.2) affects the execution cost. Therefore, as discussed in Section 3.4 we compute the billing costs for each design choice using the cost rates from Table 3. The trade-offs between these costs for the different downstream tasks and dataset are visualized in Figure 9. In each plot, the y-axis is the execution cost measured in CSC Billing Units (BUs) and the x-axis is the data storage cost measured in billing units per hour (BU/hr). The execution cost variance is from the variance in query latency from different workload buckets. The desirable region is the bottom-left of the plot, which corresponds to both low storage and execution costs.

From Figure 9 reception task, the Aria framework has the lowest execution cost of 0.01 BU and the highest storage cost of 0.2 for the Basic dataset due to its low latency and large materialized relations, respectively. Contrastingly, Spark has the highest execution cost of 50 BU and the lowest storage costs of only 0.01 BU/hr due to its expensive distributed processing and efficient storage format, respectively. We notice that when scaling to the Extended dataset, the Columnstore framework for Denormalized data starts offering a low storage cost of 0.1 BU/hr for only a slightly higher execution cost of 0.13 BU.

For the top quotes task (Figure 9 right), the smaller Denormalized tables offer the lowest storage and execution costs across frameworks. Furthermore, the Aria framework with indexed tables has the overall lowest query execution cost. However, due to the indexing, Aria has the highest storage costs for Intermediate and Standard data. Therefore, Columnstore framework offers a better trade-off for the Intermediate level and Spark is the only feasible option for the Standard level.

4.4 Discussion

There are three main takeaways from the results presented above.

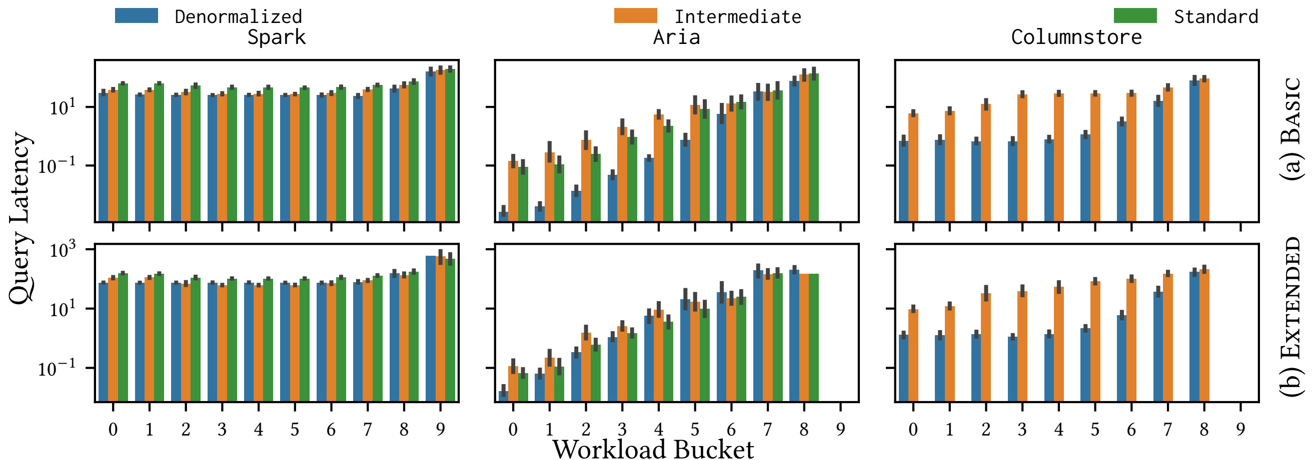

Figure 6: Latency for reception task queries from different workload buckets (see Figure 4). Columns and rows correspond to frameworks and datasets respectively. Missing results indicate that the queries didn't finish after 5 minutes.

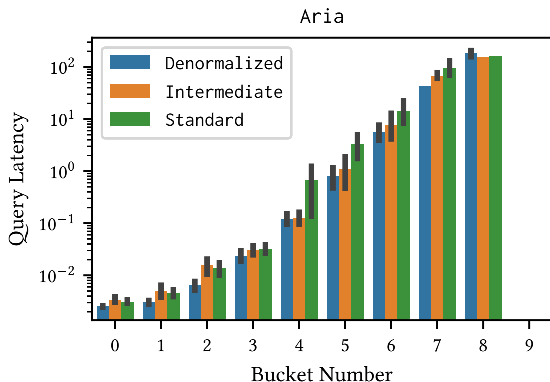

Figure 7: Reception query latency for `Aria` framework with hot cache for Extended dataset.

`Aria` with `Denormalized` data is optimal due to the significantly lower latency from materializing and indexing task-specific data, especially for queries from lower order buckets. While `Denormalized` data has materialization and data storage costs, especially for the reception task, query latency is more critical for end-users.

`Spark` is irreplaceable in the large-scale pre-processing of text reuse data (Section 2). Furthermore, Section 4.1 shows that `Spark` is the only viable framework for the materialization and loading data required for the `Intermediate` and `Denormalized` levels. Lastly, for both tasks, `Spark` is the only framework to consistently answer `Standard` queries from the highest order bucket.

`Columnstore` is a middle ground between the `Spark` and `Aria` frameworks. Compared to `Aria`, it has a higher but consistent query latency. And similar to `Spark`, the `Columnstore` framework has very low storage costs, even with `Denormalized` data. The two main drawbacks are inability to run `Standard` queries and the high materialization and loading costs.

5 RELATED WORKS

Several systems for exploring text reuse exist [6, 11, 16, 22]. While some of them use different approaches for identifying text reuses from noisy OCR data, such as n-grams [19], they have similar pre-processing phases, such as filtering, merging and clustering the identified text reuses. However, many of these systems focus primarily on historical newspaper corpora and are built upon only a few million text reuses, at most. In contrast, `ReceptionReader` is built upon billions of text reuses from large heterogeneous corpora and, based on our results, the best-performing configuration that emerged from the evaluation is anticipated to scale comfortably to corpora larger than the current ones. Moreover, `ReceptionReader` is now optimized for complex downstream analytical tasks such as reception and top quotes, while previous systems simply provide an interface to view the pre-processed text reuses.

Several studies compare row-store and column-store database management systems [1, 4]. However, these studies use synthetic schema, data, and queries from benchmarks like SSB [14] to analyze the generic trade-offs offered by different database systems. In contrast, we explore trade-offs and extract insights in the context of a specific application (analyzing large-scale text reuse data), from a combination of execution frameworks and normalization levels. The purpose of this paper is not to propose and evaluate novel algorithms or DBMSs, but rather to explore, dissect, and resolve the data management challenges related to a real data science system.

6 CONCLUSION

In this paper, we presented insights from `ReceptionReader`, our system for analyzing text reuses in large historical corpora. We studied the impact of different data normalization and execution frameworks on the performance of the system for the two downstream analysis tasks of reception and top quotes. Specifically, we considered three options for normalization (`Standard`, `Intermediate`, and `Denormalized`) and execution frameworks (`Aria`, `Columnstore`, and `Spark`). Each normalization had distinct materialized tables and downstream queries, while each framework had its own storage format and execution engine. We evaluated the system on each combination of options with datasets containing billions of text reuses for downstream tasks over several sampled

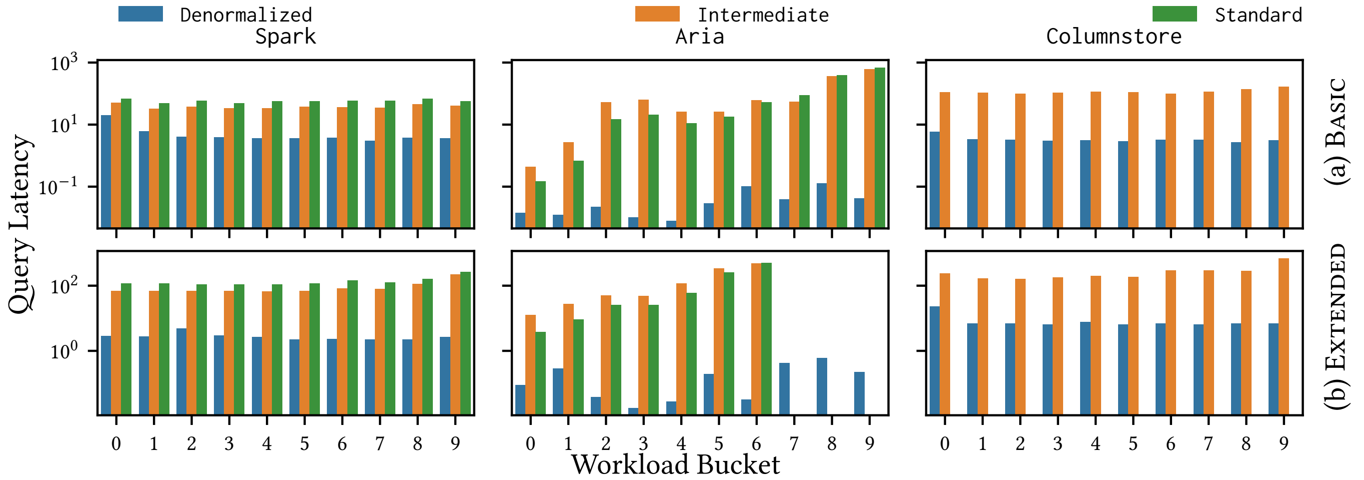

Figure 8: Latency for top quotes queries from different workload buckets (see Figure 5). Columns and rows correspond to frameworks and datasets, respectively. Missing results indicate that the query didn't finish after 15 minutes.

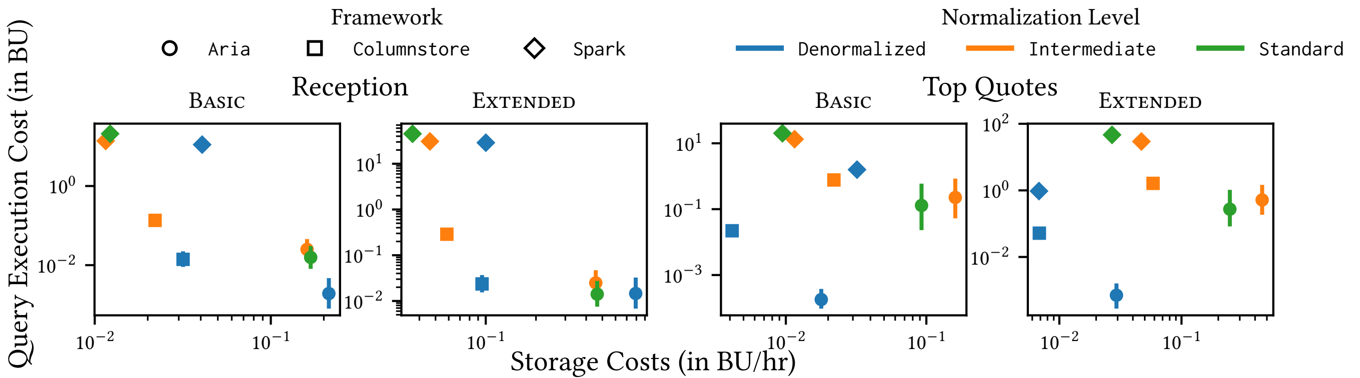

Figure 9: Trade-off plots between query execution and data storage costs for the Reception and Top Quotes tasks. Markers and colors correspond to framework and normalization levels respectively. Lower is better for both metrics.

queries. Each combination offered non-obvious trade-offs between metrics like query latency, data storage costs and execution costs.

First, the `Columnstore` framework was infeasible with `Standard` data, but with `Denormalized` data offered scalability along with low storage and execution costs. Next, the `Spark` framework is irreplaceable for the back-end large-scale pre-processing of text reuse data. Additionally, it was the only feasible option for materializing `Intermediate` and `Denormalized` data for other frameworks. Finally, the `Aria` framework with the indexed `Denormalized` data had higher materialization and data storage costs. However, it also had the lowest latency, proving optimal for user-end applications.

In conclusion, there are two high-level insights that we hope the readers take from this study. The first is that, when considering the design of a data science system, resolving the performance trade-offs that arise from different choices for data management may depend crucially on the resource constraints and user requirements rather than isolated system performance. Indeed, while for our resource costs and availability `Aria` proved to be the best option for resolving the data storage-execution cost at the user-end of the processing (Section 4.3, different settings could have led to a different outcome. The second is that because different execution frameworks and database designs may perform differently for different computation tasks (e.g., pre-processing vs. user-query processing), it may be worth using a different framework at different processing stages, even at the additional cost of transforming data from one framework to another between processing stages. Indeed, for `ReceptionReader`, it proved that a combination of `Spark` for pre-processing and intermediate materialization stages with `Denormalized` `Aria` for user-query processing stages was practically a necessity, as the relational engines could not handle the former stages and `Spark` under-performed in query latency for the latter.

REFERENCES

[1] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. 2008. Column-Stores vs. Row-Stores: How Different Are They Really?. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 967–980. https://doi.org/10.1145/1376616.1376712

[2] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. 1990. Basic local alignment search tool. *Journal of Molecular Biology* 215, 3 (1990), 403–410. https://doi.org/10.1016/S0022-2836(05)80360-2

[3] Chris Biemann. 2006. Chinese Whispers - an Efficient Graph Clustering Algorithm and its Application to Natural Language Processing Problems. In *Proceedings of TextGraphs: the First Workshop on Graph Based Methods for Natural Language Processing*, Rada Mihalcea and Dragomir Radev (Eds.). Association for Computational Linguistics, New York City, 73–80. https://aclanthology.org/W06-3812

[4] Hichem Chaalal, Mostefa Hamdani, and Hafida Belbachir. 2021. Finding the Best between the Column Store and Row Store Databases. In *Proceedings of the 10th International Conference on Information Systems and Technologies* (Lecce, Italy) *(ICIST '20)*. Association for Computing Machinery, New York, NY, USA, Article 40, 4 pages. https://doi.org/10.1145/3447568.3448548

[5] Matthew Christy, Anshul Gupta, Elizabeth Grumbach, Laura Mandell, Richard Furuta, and Ricardo Gutierrez-Osuna. 2017. Mass Digitization of Early Modern Texts With Optical Character Recognition. *J. Comput. Cult. Herit.* 11, 1, Article 6 (dec 2017), 25 pages. https://doi.org/10.1145/3075645

[6] Marten Düring, Matteo Romanello, Maud Ehrmann, Kaspar Beelen, Daniele Guido, Brecht Deseure, Estelle Bunout, Jana Keck, and Petros Apostolopoulos. 2023. impresso Text Reuse at Scale. An interface for the exploration of text reuse data in semantically enriched historical newspapers. *Frontiers in big data* 6 (2023), 1249469. https://doi.org/10.3389/fdata.2023.1249469

[7] Gale. [n.d.]. British Library Newspapers. https://www.gale.com/intl/primary-sources/british-library-newspapers

[8] Gale. 2003. Eighteenth Century Collections Online. https://www.gale.com/intl/primary-sources/eighteenth-century-collections-online

[9] Stephen H. Gregg. 2021. *Old Books and Digital Publishing: Eighteenth-Century Collections Online*. Cambridge University Press.

[10] Anshul Gupta. 2015. *Assessment of OCR Quality and Font Identification in Historical Documents*. Master's thesis. Texas A & M University.

[11] Martyn Harris, Mark Levene, Dell Zhang, and Dan Levene. 2018. Finding Parallel Passages in Cultural Heritage Archives. *J. Comput. Cult. Herit.* 11, 3, Article 15 (aug 2018), 24 pages. https://doi.org/10.1145/3195727

[12] Mark J Hill and Simon Hengchen. 2019. Quantifying the impact of dirty OCR on historical text analysis: Eighteenth Century Collections Online as a case study. *Digital Scholarship in the Humanities* 34, 4 (04 2019), 825–843. https://doi.org/10.1093/llc/fqz024 arXiv:https://academic.oup.com/dsh/article-pdf/34/4/825/33046904/fqz024.pdf

[13] University of Michigan. 2009. Early English Books Online - Text Creartion Partnership (EEBO-TCP). https://quod.lib.umich.edu/e/eebogroup/

[14] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Performance Evaluation and Benchmarking*, Raghunath Nambiar and Meikel Poess (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 237–252.

[15] Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database Management Systems*. Mc Graw Hill.

[16] Glenn Roe, Clovis Gladstone, Robert Morrissey, and Mark Olsen. 2016. Digging into ECCO: Identifying Commonplaces and other Forms of Text Reuse at Scale.. In *Digital Humanities 2016: Conference Abstracts*. 336–339.

[17] David Rosson, Eetu Mäkelä, Ville Vaara, Ananth Mahadevan, Yann Ryan, and Mikko Tolonen. 2023. Reception Reader: Exploring Text Reuse in Early Modern British Publications. *Journal of Open Humanities Data* 9 (2023). https://doi.org/10.5334/johd.101

[18] Yann Ryan, Ananth Mahadevan, and Mikko Tolonen. 2023. A Comparative text similarity analysis of the works of Bernard Mandeville. *Digital Enlightenment Studies* 1 (12 2023), 28–58. Issue 1. https://doi.org/10.61147/des.6

[19] David A. Smith, Ryan Cordell, and Abby Mullen. 2015. Computational Methods for Uncovering Reprinted Texts in Antebellum Newspapers. *American Literary History* 27, 3 (06 2015), E1–E15. https://doi.org/10.1093/alh/ajv029 arXiv:https://academic.oup.com/alh/article-pdf/27/3/E1/194881/ajv029.pdf

[20] Martyn P. Thompson. 1993. Reception Theory and the Interpretation of Historical Meaning. *History and Theory* 32, 3 (1993), 248–272.

[21] Aleksi Vesanto. 2018. *Detecting and Analyzing Text Reuse with BLAST*. Master's thesis. University of Turku.

[22] Aleksi Vesanto, Filip Ginter, Hannu Salmi, Asko Nivala, and Tapio Salakoski. 2017. A System for Identifying and Exploring Text Repetition in Large Historical Document Corpora. In *Proceedings of the 21st Nordic Conference on Computational Linguistics*, Jörg Tiedemann and Nina Tahmasebi (Eds.). Association for Computational Linguistics, Gothenburg, Sweden, 330–333. https://aclanthology.org/W17-0249

[23] Aleksi Vesanto, Asko Nivala, Heli Rantala, Tapio Salakoski, Hannu Salmi, and Filip Ginter. 2017. Applying BLAST to Text Reuse Detection in Finnish Newspapers and Journals, 1771-1910. In *Proceedings of the NoDaLiDa 2017 Workshop on Processing Historical Language*, Gerlof Bouma and Yvonne Adesam (Eds.). Linköping University Electronic Press, Gothenburg, 54–58. https://aclanthology.org/W17-0510

[24] Peng Ye and David Doermann. 2013. Document Image Quality Assessment: A Brief Survey. In *2013 12th International Conference on Document Analysis and Recognition*. 723–727. https://doi.org/10.1109/ICDAR.2013.148