International Journal of Data Science and Analytics (2025) 20:4631–4643
https://doi.org/10.1007/s41060-025-00742-x

REGULAR PAPER

Text reuse in large historical corpora: insights from the optimization of a data science system

Ananth Mahadevan[1] · **Michael Mathioudakis**[1] · **Eetu Mäkelä**[2] · **Mikko Tolonen**[2]

Received: 4 October 2024 / Accepted: 18 February 2025 / Published online: 7 April 2025

Abstract

Text reuse is of fundamental importance in humanities research, as near-verbatim pieces of text in different documents provide invaluable information about the historical spread, evolution of ideas and composition of cultural artifacts. 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 principled and extensive evaluations across different normalization levels, query execution engines, and queries of interest that led to an optimized system—and offers insights from the observed trade-offs and how they were resolved to fit specific requirements. In summary, the paper explains how, for our system, (1) the row-store engine (MariaDB Aria) with denormalized relations emerged as the optimal choice for front-end interfaces, while (2) big data processing (Apache Spark) proved irreplaceable for data preprocessing.

Keywords Data science system optimization · Big data processing · Historical corpora · Text reuse

1 Introduction

Text reuse is an essential methodological element in humanities research, which involves the detection and analysis of near-verbatim 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 [1, 2]. 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] a system for large-scale text reuse analysis developed by a interdisciplinary team of experts in history, NLP, and data science that has already pioneered intellectual history research [4]. `ReceptionReader` is built upon billions of text reuse instances identified over almost all known 18th-century documents (Sect. 2), and provides two main downstream analysis tasks:

1. **Reception Task** (Sect. 3.3.1): Given a document id, this task retrieves all instances of reuses stemming from the document, returning a mapping of its passages to corresponding reused passages in other documents. This

✉ Ananth Mahadevan
ananth.mahadevan@helsinki.fi

Michael Mathioudakis
michael.mathioudakis@helsinki.fi

Eetu Mäkelä
eetu.makela@helsinki.fi

Mikko Tolonen
mikko.tolonen@helsinki.fi

[1] Department of Computer Science, University of Helsinki, Helsinki, Finland

[2] Department of Digital Humanities, University of Helsinki, Helsinki, Finland

[1] See https://receptionreader.com/ for the front-end interface of the system deployed by Rosson et al. [3].

enables users to trace the influence and *reception* of a specific document in the corpora.
2. **Top Quotes Task** (Sect. 3.3.2): Given a set of document ids, this task identifies passages with the highest number of unique reuses across the corpora, along with reuse counts. It provides insights into the most influential passages within the selected documents.

This paper presents the relevant data management considerations, as well as the systematic and extensive evaluation that led to an optimized system. In more detail, the optimization targets simultaneously multiple system performance metrics (Sect. 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 the best 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, namely 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 (Sect. 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 preprocessing 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 for the front-end interface of the pipeline, but `Spark` as an irreplaceable back-end for preprocessing.

2 Description of `ReceptionReader` system

We begin with a brief presentation of the corpora (Sect. 2.1), text reuse detection algorithm (Sect. 2.2), and preprocessing pipeline (Sect. 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) [5–7] and `EEBO-TCP` (Early English Books Online Text Creation Partnership) [8] contain books mainly in English, while `Newspapers` (British Library Newspapers) [9] contain newspaper articles, advertisements, announcements etc. Most documents were published in the 17th and 18th centuries. The corpora were digitized with OCR (optical character recognition) by Christy et al.[10]. While of immense value, the raw OCR'ed texts are noisy, partly due to archaic fonts and layouts [11, 12], leading to varying OCR quality [13].

2.2 Text reuse detection

For the purposes of this paper, we focus on lexical text reuse and define an instance of text reuse as *any pair of near-verbatim pieces of text from different documents*. Due to the presence and variance of noise in the raw texts, standard string matching algorithms perform poorly at identifying text reuse instances. Additionally, vector similarity methods, commonly using models like Sentence-BERT [14] to embed text into dense vectors and then using vector similarity search libraries like FAISS [15], may be suited for identifying semantically similar text, but they are not suited for detecting near-exact phrasing in the presence of OCR noise, which is our purpose here.

Instead of the aforementioned methods, to efficiently identify text reuse instances, the system uses a method developed in a prior phase by Vesanto [16]. This method utilizes BLAST (Basic Local Alignment Search Tool) [17], a fuzzy-string-matching algorithm widely used for DNA sequence matching, and has proved to be more effective than alternatives in finding text reuses with noisy OCR data [18]. BLAST is employed across the corpora in an all-to-all document comparison and outputs '**hits**' which are pairs of text fragments identified as instances of text reuse.[2] We refer to each such text fragment as a **piece**, identified by a (document ID, start-offset, end-offset) tuple.

After receiving all the hits from BLAST, the system stores and indexes the raw texts and their location. Moreover, it prepares the hits for querying by passing them through a preprocessing pipeline (Sect. 2.3) that does not require the raw text strings. Then, when users request to see the raw text of a piece, it is received via an API with the piece tuple which returns the page and raw text with the piece highlighted.[3]

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

[3] An example call to the Octavo API can be seen at this link.

Table 1 Historical document corpora. Document length is measured in number of characters

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

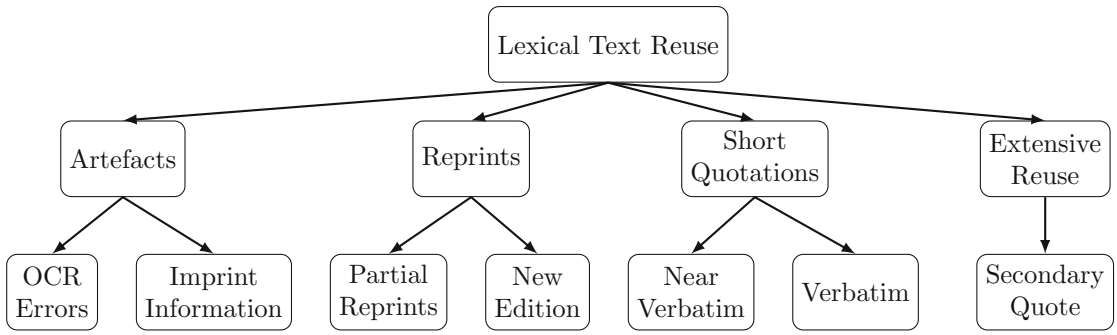

Fig. 1 Taxonomy of lexical text reuses [4]

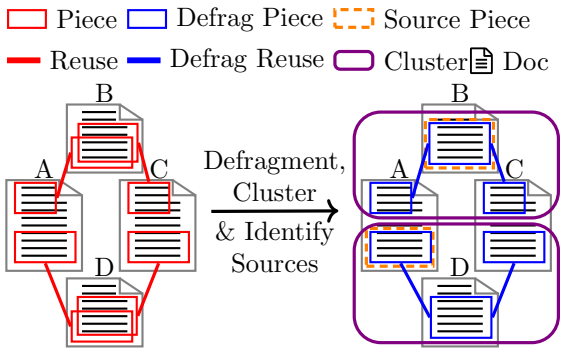

Fig. 2 Preprocessing: BLAST hits (red) are defragmented (blue) and clustered (purple). In each cluster, the earliest piece is identified as the 'source' (dashed yellow)

This BLAST methodology results in a wide spectrum of lexical text reuses ranging from small OCR errors to large passages of similar text across different editions. Figure 1 shows the taxonomy of lexical text reuse from Ryan et al. [4] which highlights the different types of reuses that is present in the raw text reuse data obtained from BLAST. Since each type of lexical text reuse in this taxonomy is valuable in different historical studies and downstream analysis tasks, it is not pre-filtered at this stage. Instead, each downstream task filters instances and materializes data useful in answering queries for that particular task.

2.3 Preprocessing pipeline

To become useful for downstream tasks, the detected text reuses are preprocessed in a three-phase pipeline (Fig. 2).

Phase 1: Defragmentation When a text passage is reused in several documents, its offsets might differ slightly across BLAST hits due to OCR noise and errors (overlapping red boxes in documents B and D in Fig. 2). 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 Fig. 2).

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 [19]). The resulting clusters are groups of similar defragmented pieces from different documents, depicted with rounded purple rectangles in Fig. 2.

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 Fig. 2). Each other piece in the cluster is correspondingly labeled as a 'destination.' This source and destination labeling is performed with the available document metadata.

The preprocessing 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 (Fig. 3).

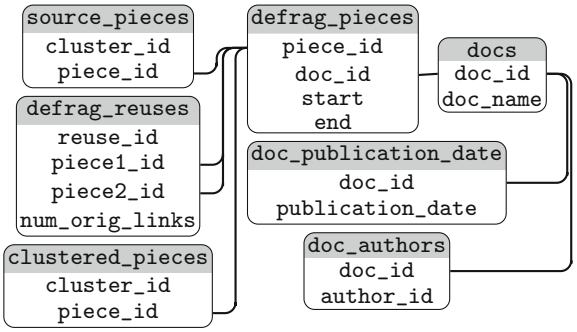

Fig. 3 Simplified schema for preprocessed data. Relations are materialized for the `Standard` normalization level

3 Evaluation setup

This section presents the datasets (Sect. 3.1), design choices (Sect. 3.2), and performance metrics (Sect. 3.4) involved in the evaluation, with the evaluation performed for two analysis tasks, namely *reception* and *top quotes* (Sect. 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 prioritize scalability, as the system is intended to accommodate many additional digital corpora in the future. Toward that end, we evaluated design choices on datasets of varying sizes, with datasets defined as subsets of the corpora described in Sect. 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.

Specifically, the first dataset, BASIC, combines `ECCO` and `EEBO-TCP`: reuse detection (Sect. 2.2) returns nearly one billion BLAST hits for it, and preprocessing (Sect. 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 preprocessing 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 the 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.

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

Attribute	BASIC	EXTENDED
Corpora	`ECCO` & `EEBO-TCP`	`ECCO`, `EEBO-TCP` & `Newspapers`
documents	267K	2.04M
BLAST hits	966 M	6.31B
defragmented pieces	384 M	1.10B
avg. defrag. piece length	748	461
defragmented reuses	965 M	5.61B
clusters	50.3M	91.6M
authors[4]	46.0K	46.0K

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

While these trade-offs might be individually understood and anticipated (see Ramakrishnan and Gehrke[20]), 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.

3.2.1 Query execution frameworks

The raw text reuse graph data described in Sect. 2 is preprocessed and stored in the schema shown in Fig. 3. For query execution, we selected three frameworks—`Spark`, `Aria`, and `Columnstore` (Table 3)—to balance scalability, query performance, and compatibility with our relational schema. While the data can be conceptualized as a sparse graph, our tasks focus on retrieval and aggregation rather than graph traversal, making relational query systems more appropriate. Graph databases (e.g., Neo4J) were evaluated during development but were found to be less efficient for prepro-

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

cessing and clustering requirements at our data scale. The selected frameworks provide distinct execution engines and storage formats, enabling a comprehensive evaluation of performance trade-offs for downstream tasks expressed as SQL queries.

`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 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) are reported in Table 3.

3.2.2 Normalization levels

The evaluation includes the following options. **`Standard`:** Uses the fully normalized tables shown in Fig. 3. 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 Fig. 4. 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 Fig. 4. 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 Sect. 3.2, 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.

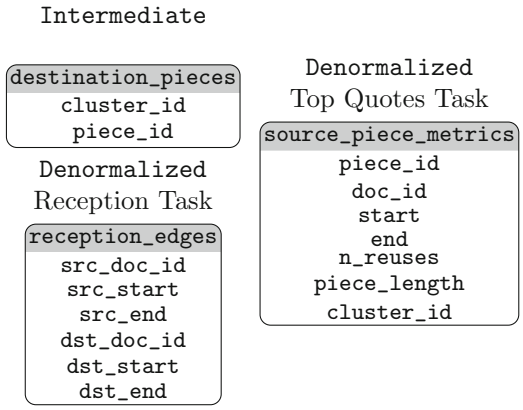

Fig. 4 Additional relations materialized for the `Intermediate` and `Denormalized` levels

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 (Fig. 3) 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

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

```
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 the reception task with `Intermediate` level data

```
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

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

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

`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 Fig. 5, 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 Fig. 5). Documents in higher workload buckets have more reception edges, and the corresponding queries will have higher latency. This is used in Sect. 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. Toward 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**

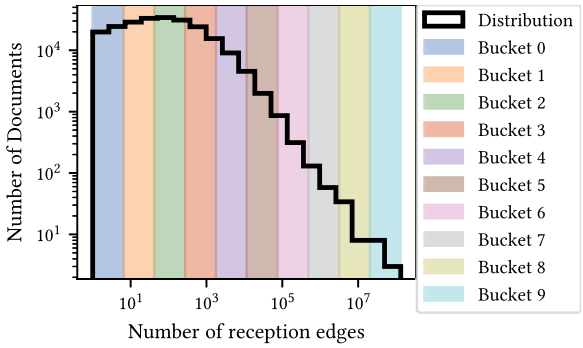

Fig. 5 Number of reception edges distribution for the BASIC dataset. The log-spaced workload buckets, shown as vertical bands, are used for sampling evaluation queries

```
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 the top quotes task with `Intermediate` level data

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`

```
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 the top quotes with `Denormalized` level data

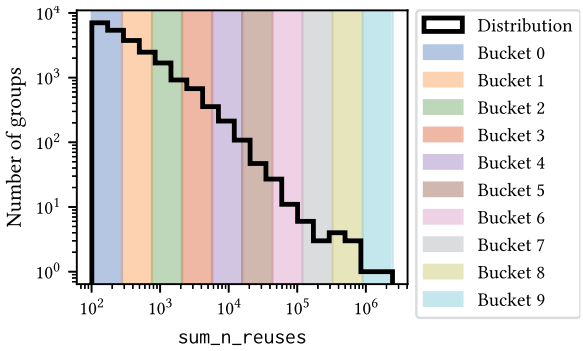

Fig. 6 Distribution of `sum_n_reuses` for the BASIC dataset. The log-spaced workload buckets, shown as vertical bands, are used for sampling evaluation queries

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 `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 Fig. 6, 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 [3]. 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 Fig. 5). 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 are 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 is 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. For each downstream task, we compute the weighted average of the query execution cost, where the weights correspond to the proportion of queries for a given workload bucket, and report it as the expected query execution cost. 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 Sect. 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 Figs. 5 and 6). 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 (Fig. 4) to be materialized before answering queries for downstream tasks. The number of rows in these additional materialized relations are shown in Table 4. We attempted materializing these additional relations

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

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

directly in each evaluated execution framework. However, in practice, the `Denormalized` tables failed to materialize after 24 h 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.

We report the materialization and loading times, and the execution cost in BUs for different normalization levels of the EXTENDED dataset in Table 5. The bulk-loading execution cost for the `Aria` framework is computed as a sum of the loading costs from `Spark` to `Aria` and then the indexing of assets in `Aria`. For the `Columnstore` framework, the bulk-loading cost contains the additional cost of loading from indexed `Aria` tables into `Columnstore` tables.

The `Spark` framework takes 2, 11 and 24 minutes to materialize the `destination_pieces`, `reception_edges` and `source_piece_metrics` relations, respectively. 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 half that time.

In the bulk-loading, the materialized relations into the `Aria` framework 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 different normalization levels. In addition, indexing the tables takes 51 minutes, 4.3 hours and 7.1 minutes, respectively. For the `Columnstore` framework, we used the `Aria` relations to bulk-load data, thereby taking longer times consistently.

4.2 Latency results

The materialization and loading discussed in Sect. 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 Figs. 7 and 9. 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 (Figs. 7 and 9 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. This 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 Fig. 7 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 (Fig. 7(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 Fig. 8. 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 Fig. 9, 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

Table 5 Materialization times and execution costs in the `Spark` framework and loading times to other frameworks for different normalization levels of the EXTENDED dataset. Each cell shows the time in HH:MM:SS format, and the execution cost in BUs is shown in parantheses

Normalization Level	Materialization in Spark	Bulk-loading to Aria	Bulk-loading to Columnstore
`Intermediate`	00:02:02 (42.50)	01:15:52 (537.50)	01:31:59 (543.94)
`Denormalized` (Reception Task)	00:11:30 (240.35)	06:49:10 (3298.67)	14:30:2 (3483.47)
`Denormalized` (Top Quotes Task)	00:24:32 (512.75)	00:10:39 (77.42)	00:15:5 (79.50)

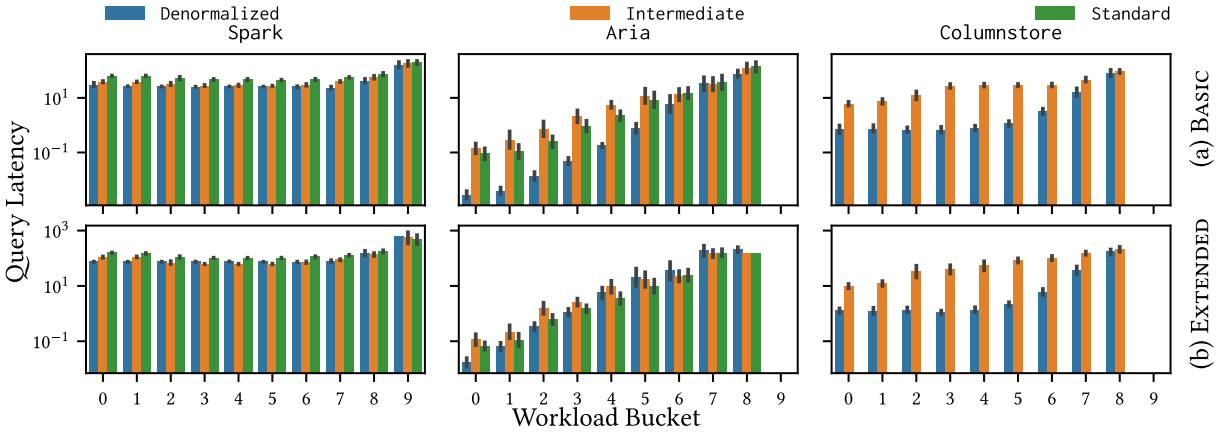

Fig. 7 Latency for reception task queries from different workload buckets (see Fig. 5). Columns and rows correspond to frameworks and datasets, respectively. Missing results indicate that the queries did not finish after 5 min

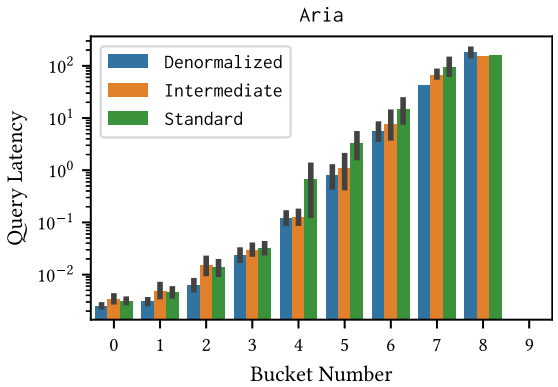

Fig. 8 Reception query latency for `Aria` framework with hot cache for EXTENDED dataset

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

Let us now move on from the latency considerations, which are important for end-users, to resource and billing metrics, which are important for system maintainers. The size of each normalization level's materialized relations (Sect. 4.1) affect the storage costs, and the query latency of the execution framework (Sect. 4.2) affects the execution cost. Therefore, as discussed in Sect. 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 Fig. 10. In each plot, the y-axis is the expected 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 desirable region is the bottom left of the plot, which corresponds to both low storage and execution costs, and due to the multiple optimization criteria, the Pareto frontier is shown in each plot as the gray dashed line.

From the *reception task* plot in Fig. 10 (top row), we see that the `Aria` framework with `Denormalized` relations has the lowest expected execution cost of 4e-3 BU and the highest storage cost of 0.2 BU/hr for the BASIC dataset due to its low latency and large materialized relations, respectively. Contrastingly, `Spark` for the same normalization level has a higher expected execution cost of 9.3 BU and one of the lowest storage costs of only 0.04 BU/hr due to its expensive

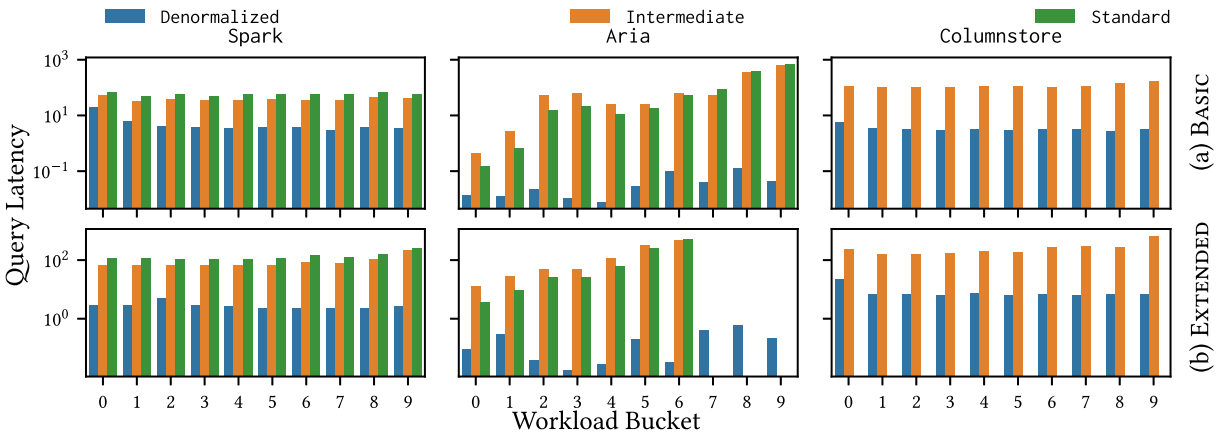

Fig. 9 Latency for top quotes queries from different workload buckets (see Fig. 6). Columns and rows correspond to frameworks and datasets, respectively. Missing results indicate that the query did not finish after 15 min

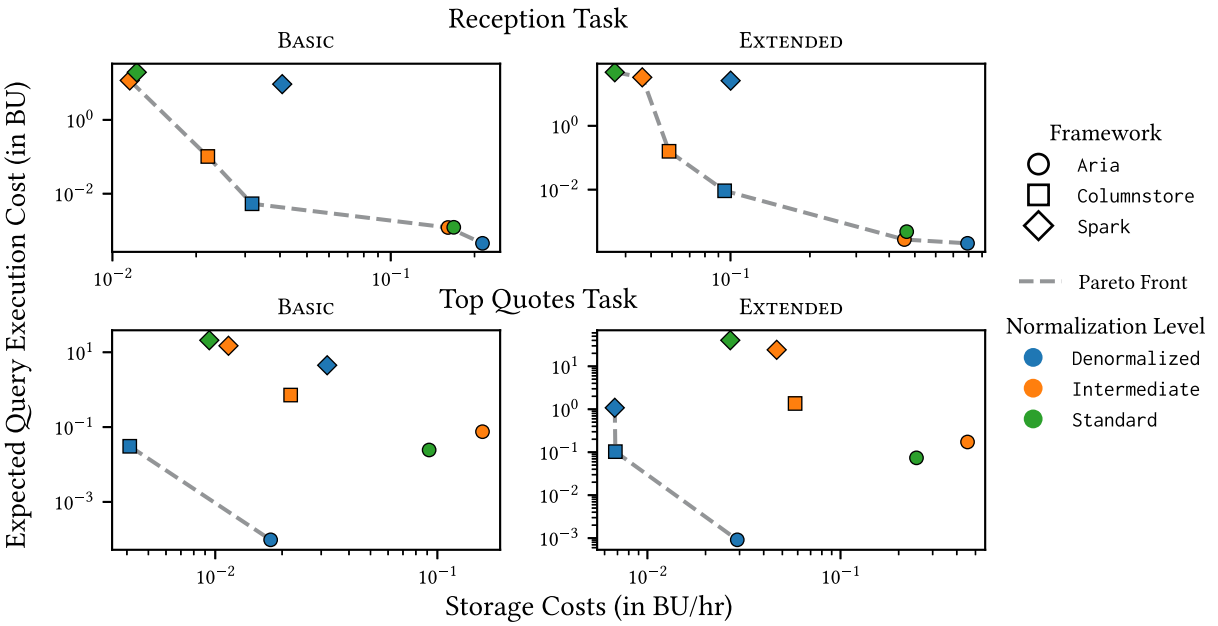

Fig. 10 Trade-off plots between expected query execution and data storage costs for the Reception and Top Quotes tasks. Markers and colors correspond to framework and normalization levels, respectively. Lower is better for both metrics, and the Pareto frontier is indicated by the dashed gray line

distributed processing and efficient storage format, respectively. We notice that when scaling to the Extended dataset, the `Columnstore` framework with `Denormalized` data starts offering a low storage cost of 0.1 BU/hr for only a slightly higher execution cost of 9e-3 BU compared to the 0.8 BU/hr and 2e-4 BU of the `Aria` framework with the same normalization level.

For the *top quotes task* plot in Fig. 10 (bottom row), 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. Consequently, the `Columnstore` framework has higher expected execution costs with lower storage costs for the `Intermediate` and `Denormalized` levels. Lastly, the `Spark` framework is the only feasible option for the `Standard` level.

4.4 Discussion

The previous sections outlined the trade-offs between the key metrics of query latency, data storage costs and expected query execution costs for the different execution frameworks and data normalization levels. In this section, using the

insights from these various trade-offs, we proceed to synthesize a solution that is optimal for our requirements.

First, humanities researchers regularly use a front-end interface connected to the system to run both the downstream analysis tasks (reception and top quotes). Due to the large volume of user queries and to ensure the best user experience, we require the lowest expected query execution cost and latency overall. Therefore, to interact with the front-end interface, we opt to use the `Aria` framework with `Denormalized` relations which offers the lowest query execution cost and expected latency for both tasks, as seen in Fig. 10 and Sect. 4.2, respectively.

Second, the system must be flexible to upstream changes in data or logic, such as the addition of additional historical text corpora or modification to preprocessing steps. Furthermore, the system must be able to efficiently materialize and load the `Denormalized` level relations needed for the previous requirement. To satisfy these requirements, we opt to use the `Spark` framework for the large-scale preprocessing of text reuse data (Sect. 2) into the `Standard` relations, and to materialize and load `Denormalized` relations into `Aria` platform for the front-end interface.

Apart from the `Aria` and `Spark` frameworks that we opt to use due to our application requirements, we observe that `Columnstore` is a middle ground between the other 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. Due to these drawbacks, and given our requirements of the system, we opt not to use `Columnstore` for our system. However, the results indicate that `Columnstore` would be a potential solution for systems that require minimal use of storage.

We further summarize our concluding insights in Sect. 6.

5 Related work

Detecting plagiarism in documents is similar to identifying text reuses in historical documents [21]. These plagiarism detection methods typically narrow the search space of all documents using vector similarity and then performing lexical analysis on the smaller subset of documents [22]. However, such methods are only viable when there is a single target document and requires on-demand processing to find reuse fragments. Additionally, it has been shown that vector space models perform poorly in the identifying identical texts due to the variance in OCR quality present in historical corpora [23].

Several systems exist specifically for exploring text reuse in historical documents [24–27]. While some of them use different approaches for identifying text reuses from noisy OCR data, such as n-grams [28], many of these systems focus on specific datasets, such as historical newspapers. In contrast, our approach is designed to operate across multiple archival collections, handling heterogeneous corpora such as books, articles and newspapers simultaneously. Furthermore, existing systems are typically built on smaller-scale datasets or focus primarily on interface-level exploration, rather than providing robust scalability to billions of text reuse instances. Our `ReceptionReader` system uniquely addresses the challenges of large-scale text reuse analysis by integrating cross-archival capabilities with scalable preprocessing and query execution frameworks, enabling us to analyze significantly larger datasets than previous attempts.

Several studies [29, 30] compare row- and column-store systems in analytical workloads, particularly in benchmarks like Star Schema Benchmark (SSB) [31]. These findings highlight the advantages of column-store systems for generalized business analysis workflows characterized by uniform query patterns and data schemas. However, our application of large-scale text reuse analysis presents unique challenges that differ significantly from those modeled by benchmarks. Specifically, our denormalized schema is for task-specific queries, such as retrieving text reuse instances and aggregating statistics by document. These queries require efficient access to complete records, favoring row-store systems. In addition, our pipeline introduces preprocessing and materialization requirements which change the workload and shift the trade-offs observed in generic scenarios. These differences highlight the importance of studying these trade-offs for the requirements of the application domain and using the insights to align system design choices.

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 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 preprocessing 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. As discussed in Sect. 4.3, there is no single best framework/normalization level to minimize costs of storage and execution, evidenced by the multiple design choices along the Pareto frontier for both downstream tasks. The second is that because different execution frameworks and database designs may perform differently for different computation tasks (e.g., preprocessing 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 preprocessing 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.

Acknowledgements This work was funded by the Academy of Finland under grants 347706, 347709, and 347747. Additionally, the authors wish to acknowledge CSC—IT Center for Science, Finland, for generous computational resources.

Funding Open Access funding provided by University of Helsinki (including Helsinki University Central Hospital).

References

1. Thompson, M.P.: Reception theory and the Interpretation of Historical Meaning. History Theory **32**(3), 248–272 (1993)
2. Spencer, M.G., Tolonen, M.: In: Skjönsberg, M., Waldmann, F. (eds.) The Reception of Hume's Essays in Eighteenth-Century Britain. Cambridge Critical Guides, pp. 15–35. Cambridge University Press (2025)
3. Rosson, D., Mäkelä, E., Vaara, V., Mahadevan, A., Ryan, Y., Tolonen, M.: Reception reader: exploring text reuse in early modern British publications. J. Open Hum. Data (2023). https://doi.org/10.5334/johd.101
4. Ryan, Y., Mahadevan, A., Tolonen, M.: A Comparative text similarity analysis of the works of Bernard Mandeville. Digital Enlightenment Studies 1, 28–58 (2023) https://doi.org/10.61147/des.6
5. Tolonen, M., Mäkelä, E., Lahti, L.: The anatomy of eighteenth century collections online (ECCO). Eighteenth-Century Stud. **56**(1), 95–123 (2022)
6. Gale: Eighteenth Century Collections Online (2003). https://www.gale.com/intl/primary-sources/eighteenth-century-collections-online
7. Gregg, S.H.: Old Books and Digital Publishing: Eighteenth-Century Collections Online. Elements in Publishing and Book Culture. Cambridge University Press, Cambridge (2021)
8. Univ. Michigan: Early English Books Online - Text Creartion Partnership (EEBO-TCP) (2009). https://quod.lib.umich.edu/e/eebogroup/
9. Gale: British Library Newspapers. https://www.gale.com/intl/primary-sources/british-library-newspapers
10. Christy, M., Gupta, A., Grumbach, E., Mandell, L., Furuta, R., Gutierrez-Osuna, R.: Mass digitization of early modern texts with optical character recognition. J. Comput. Cult. Herit. **11**(1) (2017) https://doi.org/10.1145/3075645
11. Gupta, A.: Assessment of ocr quality and font identification in historical documents. Master's thesis, Texas A & M University (2015)
12. Ye, P., Doermann, D.: Document image quality assessment: A brief survey. In: 2013 12th International Conference on Document Analysis and Recognition, pp. 723–727 (2013). https://doi.org/10.1109/ICDAR.2013.148
13. Hill, M.J., Hengchen, S.: Quantifying the impact of dirty OCR on historical text analysis: Eighteenth Century Collections Online as a case study. Digit. Scholarsh. Humanit. **34**(4), 825–843 (2019). https://doi.org/10.1093/llc/fqz024
14. Reimers, N., Gurevych, I.: Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In: Inui, K., Jiang, J., Ng, V., Wan, X. (eds.) Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019, pp. 3980–3990. Association for Computational Linguistics, New York, NY, USA (2019). https://doi.org/10.18653/V1/D19-1410
15. Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P., Lomeli, M., Hosseini, L., Jégou, H.: The Faiss library. CoRR **abs/2401.08281** (2024). https://doi.org/10.48550/ARXIV.2401.08281 arXiv:2401.08281
16. Vesanto, A.: Detecting and Analyzing Text Reuse with BLAST. Master's thesis, University of Turku (2018)
17. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. J. Mol. Biol. **215**(3), 403–410 (1990). https://doi.org/10.1016/S0022-2836(05)80360-2
18. Vesanto, A., Nivala, A., Rantala, H., Salakoski, T., Salmi, H., Ginter, F.: Applying BLAST to text reuse detection in Finnish newspapers and journals, 1771-1910. In: Bouma, G., Adesam, Y.

(eds.) Proceedings of the NoDaLiDa 2017 Workshop on Processing Historical Language, pp. 54–58. Linköping University Electronic Press, Gothenburg (2017). https://aclanthology.org/W17-0510
19. Biemann, C.: Chinese whispers - an efficient graph clustering algorithm and its application to natural language processing problems. In: Mihalcea, R., Radev, D. (eds.) Proceedings of TextGraphs: the First Workshop on Graph Based Methods for Natural Language Processing, pp. 73–80. Association for Computational Linguistics, New York City (2006). https://aclanthology.org/W06-3812
20. Ramakrishnan, R., Gehrke, J.: Database Management Systems. Mc Graw Hill, USA (2003)
21. Büchler, M., Burns, P.R., Müller, M., Franzini, E., Franzini, G.: In: Biemann, C., Mehler, A. (eds.) Towards a Historical Text Re-use Detection, pp. 221–238. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-12655-5_11
22. Foltýnek, T., Meuschke, N., Gipp, B.: Academic plagiarism detection: A systematic literature review. ACM Comput. Surv. **52**(6) (2019)https://doi.org/10.1145/3345317
23. Gladstone, C., Cooney, C.: Opening new paths for scholarship: Algorithms to track text reuse in eighteenth-century collections online. Digitizing Enlightenment. Digital Humanities and the Transformation of Eighteenth-Century Studies. Oxford University Studies in Enlightenment, 353–371 (2020)
24. Düring, M., Romanello, M., Ehrmann, M., Beelen, K., Guido, D., Deseure, B., Bunout, E., Keck, J., Apostolopoulos, P.: impresso text reuse at scale. an interface for the exploration of text reuse data in semantically enriched historical newspapers. Frontiers in big data 6, 1249469 (2023). https://doi.org/10.3389/fdata.2023.1249469
25. Harris, M., Levene, M., Zhang, D., Levene, D.: Finding parallel passages in cultural heritage archives. J. Comput. Cult. Herit. **11**(3) (2018). https://doi.org/10.1145/3195727
26. Vesanto, A., Ginter, F., Salmi, H., Nivala, A., Salakoski, T.: A system for identifying and exploring text repetition in large historical document corpora. In: Tiedemann, J., Tahmasebi, N. (eds.) Proceedings of the 21st Nordic Conference on Computational Linguistics, pp. 330–333. Association for Computational Linguistics, Gothenburg, Sweden (2017). https://aclanthology.org/W17-0249
27. Roe, G., Gladstone, C., Morrissey, R., Olsen, M.: Digging into ecco: Identifying commonplaces and other forms of text reuse at scale. In: Digital Humanities 2016: Conference Abstracts, pp. 336–339 (2016)
28. Smith, D.A., Cordell, R., Mullen, A.: Computational Methods for Uncovering Reprinted Texts in Antebellum Newspapers. Am. Lit. History **27**(3), 1–15 (2015). https://doi.org/10.1093/alh/ajv029
29. Abadi, D.J., Madden, S.R., Hachem, N.: Column-stores vs. row-stores: How different are they really? In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. SIGMOD '08, pp. 967–980. Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1376616.1376712
30. Chaalal, H., Hamdani, M., Belbachir, H.: Finding the best between the column store and row store databases. In: Proceedings of the 10th International Conference on Information Systems and Technologies. ICIST '20. Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3447568.3448548
31. O'Neil, P., O'Neil, E., Chen, X., Revilak, S.: The star schema benchmark and augmented fact table indexing. In: Nambiar, R., Poess, M. (eds.) Performance Evaluation and Benchmarking, pp. 237–252. Springer, Berlin, Heidelberg (2009)