FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Reproducible Ranked Lists for Retrieval from Evolving Document Collections

How Column-Store Technology Enhances the Capability of Inverted Indices

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Business Informatics

by

Hannes Bösch B.Sc.
Registration Number 0927746

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber

Vienna, 9th April, 2017

Hannes Bösch — Andreas Rauber

Erklärung zur Verfassung der Arbeit

Hannes Bösch B.Sc.
Holzhausergasse 2/21 1020 Vienna

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 9. April 2017

Hannes Bösch

Danksagung

Mein Dank gilt Herrn Prof. Rauber Andreas für die Anregung zum Thema der Arbeit, der intensiven, professionellen Betreuung, für die sehr kurzen Antwortzeiten und das stets offene Ohr.

Vielen Dank auch meiner Familie und meiner Freundin, die mich auf meinem Weg stets unterstützt haben.

Diese Arbeit widme ich meinem Vater, Werner Bösch!

Abstract

The core structure of (probabilistic) information retrieval systems lacks the ability to make retrieval result rankings reproducible. When the underlying data changes, IR indices change over time and especially the history of *tf-idf* values is hard to preserve. Thus, the same query might produce different results when the collection has been updated in the meantime. Only little research is directed to reproducibility in IR, though it would be desirable in fields of research or patent applications. The first step into this direction is to have subsets of documents in a dynamically evolving data environment unambiguously identifiable. This can be achieved with structured data and a data schema suitable for scalable data citation (cf. section 3). It suggests maintaining a history of evolving data by tagging data records with timestamps and keeping a version history for each update on the collection. Conventional row-stores cannot deal with this large volume data and statistics aggregations, as it would be required for IR applications. Yet, the column-store architecture is designed for analytical workloads and has already been proposed for IR-prototyping (cf. section 4.2), an approach for building retrieval indices on top of RDBMS. This thesis combines the concepts of IR-prototyping with data citation in order to enhance retrieval indices to achieve reproducibility. It addresses questions on how database schemes have to be shaped and if these models are efficient to deal with today's requirements on retrieval systems. The results hold promises for the future.

Contents

List of Figures

List of Tables

SQL Scripts

CHAPTER 1

Introduction

In a dynamic environment where data frequently changes it is challenging to utilize this data for analytics when demanding results to be reproducible. Common strategies are to link it to autonomous static data extracts of the data actually used. This approach becomes increasingly inconvenient and a burden to data management and storage in volatile data environments.

One of these applications is Information Retrieval (IR). The search indices produced by the retrieval system refers to the current state of the data corpora and are updated when data changes. Characteristic examples are web search engines. Result rankings to a search query today would have most likely been different a month ago, presuming new web pages have been added, existing ones were updated or others deleted in the meantime. Common IR systems miss the ability to reconstruct ranked lists. The great challenge is not only to maintain a system state over time but also the relative importance of all terms in the collection as a vital variable to retrieval algorithms and a major part of probabilistic retrieval indices. The strategy to produce snapshots of the entire collection does not scale with the potentially huge quantity of data, the effort to manage the data units and the great demands on retrieval timings retrieval systems have to cope with. While there is a lot of effort directed to build more efficient retrieval systems, only little attention has been paid on how to make retrieval results reproducible. Certainly this is not an important goal to many retrieval applications but there are use cases where reproducible retrieval result rankings are desirable.

- In the scientific domain, for researchers developing new retrieval algorithms where it is important to unambiguously refer to subsets of data being used for experiments.

Researchers in the Digital Humanities may want to extract a subset from a larger corpus of digitized material such as the 15 Mio volumes/666TB HATHI Trust repository 1 receiving documents from Google Books and other major digitization initiatives. Social Scientists may want to base their studies on subsets of documents retrieved from a news portal or social media site frequently updated with new content. Business analytics and intelligence reports are frequently based on collections of documents extracted from sources continuously updated with new content.

- For patent retrieval, where the set of patents consulted during the verification of patent applications has to be documented. The US Patent and Trademark Office Portal Patent Application Information Retrieval (PAIR)[1] attaches Boolean queries reflecting the "Examiner's search strategy" to the application with their timestamp and the number of hits returned. However, this approach is limited to Boolean queries only where result sets are reproducible, given a timestamp and no data has been removed. If there is a way to reproduce ranked lists in dynamic environments, other retrieval models can be applied.

This thesis aims at finding a solution to this problem and first addresses the question, if there is a way to effectively achieve reproducibility of retrieval ranked lists on huge volatile data corpora. We evaluate the system's efficiency to determine if it's applicable at modern hardware.

This research question has been inspired by the idea to use column-store databases for IR prototyping [MSLdV14] as well as the use of structured data to identify arbitrary subsets from a dynamic data collection in order to refer to it like static data. In this approach of data citation [PR13b] actual content is sliced into different views depending on the time data has been valid.
The inventive step is now to combine the two approaches to achieve both a platform for IR prototyping and additionally empowering the system to reproduce retrieval results with respect to the query launch dates.

1.1 Research Question

The research questions that therefore arise are as follows:
Can IR systems be structured/enhanced/modified to allow result rankings to be reproducible in a dynamically changing document-corpus?

[1]PAIR: `http://portal.uspto.gov/pair/PublicPair`

- What is the impact on index storage size in different scenarios of data dynamics?
- What is the impact on query performance in different scenarios of data dynamics?
- How can timestamping and versioning efficiently be applied to a column-store representation of term frequency values?

1.2 Research Methodology

To approach the research questions the following steps are undertaken

- Explore and discuss different versioned Column-Store Data models (VCSM) on top of the free column store MonetDB considering effectiveness and efficiency.
- Create a proof of concepts consisting test set-up to analyze different models to realize versioning.
- Adapt the retrieval query to all data schemes to enable reproducible retrieval rankings incorporating the Best Match Model Nr. 25 (BM25) retrieval model.
- Verify the effectiveness of the retrieval system by checking results to verify functional correctness.
- Perform load and balance tests to check the efficiency of the system.
- Test Settings
  - The English Wikipedia-dumps are parsed and fed into the prepared data schemes.
  - The efficiency is evaluated by comparing performance and storage demand of the models.

1.3 Structure and Overview

The thesis is roughly structured in two different parts.

- In the first part, comprising chapters 2-4, an overview on the latest state of the art is presented. These chapters cover those aspects relevant to the research questions only.
  - Information retrieval foundations.

    - Data Citation on a dynamic document corpus.
    - Column Stores for IR prototyping.

- Chapter 5-7 are dedicated to finding solutions to the actual problem statement.
    - Several data models supporting reproducible retrieval results are explored.
    - All models are benchmarked on a real system and the results are compared to each other.

A summary and outlook is given in chapter 8.

CHAPTER 2

Information Retrieval

IR has become the umbrella term for a research discipline concerned with searching collections of electronic data in order to extract, manipulate and present best matching sub-sets as search results. In general, IR is applicable for all kind of electronic data but most prominently it is associated with **electronic text retrieval** we all know from web search (Google, Yahoo, etc.). The focus of this work is directed to text retrieval thus *IR* is subsequently used as surrogate for text retrieval only. IR is the science of obtaining relevant documents out of a collection of documents according to an information need. Documents are extractable, self-contained units out of a collection of electronic text (e.g. A web page in the World-Wide-Web). Information need denotes the users' wish to have specific questions answered. Those questions are usually packed into a search query to be answered by the IR system. For web searches typical queries are formed by simple terms framing the domain and content of the web pages. The IR system has to process the search query and return those documents out of the document-pool which are most relevant to users' information need. The scoring of the IR system is computed by a ranking-module, an algorithm determining and assigning relevance-measures to documents with respect to the query terms. The documents are then ordered by relevance and a concise representation of the result-ranking is returned to the user. This happens at web-searches like "Google". In return to a users' search term, web pages with a short snippet of the most relevant documents are returned in order of their ranking score. Apparently the document placed on top should by the one satisfying the users' information need best. How well the IR system performs is measured in terms of efficiency (response-time, required space, etc.) and effectiveness (relevance to user). In the following chapters some key aspects of IR are discussed. The work will only scratch the surface of the encompassing field of study and focus mainly on premises, enabling the reader to understand subsequent chapters. Additional literature is quoted if readers want to dig deeper into the topic. Further this thesis discusses the structure, the components, evaluation strategies and the limitations of common IR systems bridging between conventional IR and Relational Database Management System (RDBMS) technology.

2.1 Structure of an IR-System

Figure 2.1 illustrates the architecture of a common IR system.

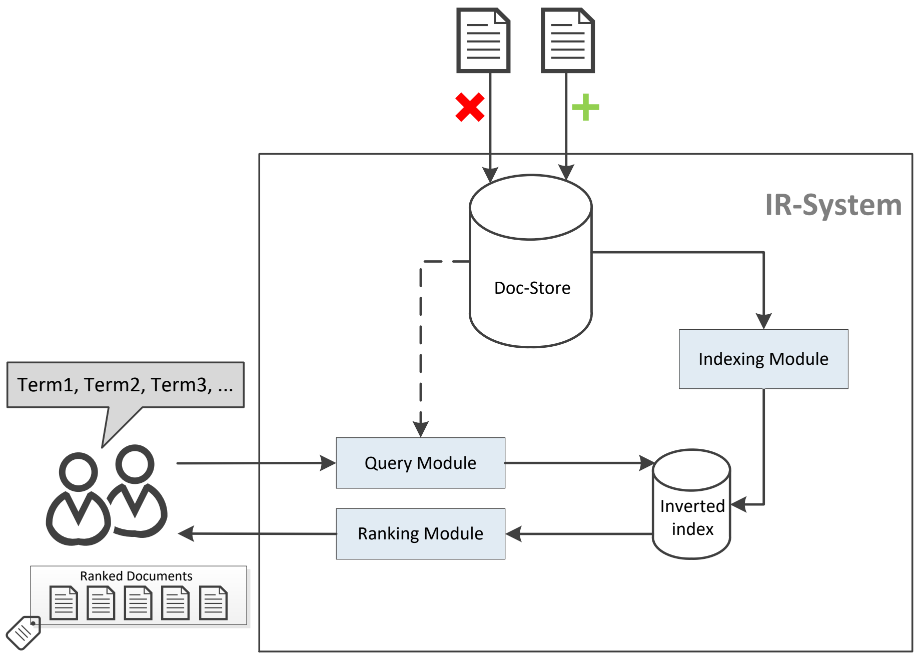
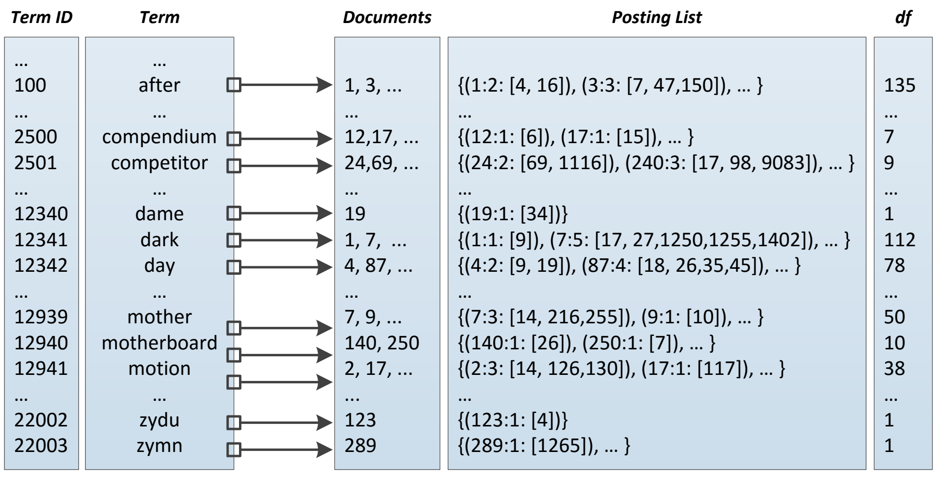


Figure 2.1: Architecture of an IR System

When documents are added, removed or changed, the IR system has to trigger changes in the systems' document store as well as to update the inverted index. Imagine a web crawler detecting updates on a web page in the World Wide Web where an older version is already stored in the IR system. A common solution to deal with changed documents is to remove the old document and re-add the new one. Both the document store and the inverted index need to be updated before a new user query may be launched considering the new document. The system responds to users queries with (usually ranked) result-sets of document sets matching the query terms. In case of web searches, a webpage with documents and document extracts is presented and the returned representatives are ordered by significance. The following subsections explain the components of an IR system in more detail.

2.2 Indexing

A The inverted index is the center-piece of most IR systems. Broadly speaking, it provides an efficient term to document mapping. A set of terms (each term once) of the collection is stored in a dictionary-like data-structure, where each dictionary-entry points to all respective documents the term can be found in. Depending on the application, the index size and available memory, different index types are appropriate [BCC10, pp. 200-210]. Along with sort-based indices, hash-based indices are very common. Here the dictionary terms are stored in the main memory entirely facilitating the system to compute the hash-values pointing at the posting lists on the disc enabling for quick "look-ups" and sequential disk access. Another very important issue to IR not further discussed here is the index compression, a trade-off between memory consumption and computational overhead. The book "An Introduction to Information Retrieval" [CDM+09, pp.67-85] is an excellent starting point to learn more about index construction and compression standards. Figure 2.2 illustrates how a data structure for an IR system can look like.



Term ID	*Term*	*Documents*	*Posting List*	*df*
…	…			
100	after	1, 3, …	{(1:2: [4, 16]), (3:3: [7, 47,150]), … }	135
…	…	…	…	…
2500	compendium	12,17, …	{(12:1: [6]), (17:1: [15]), … }	7
2501	competitor	24,69, …	{(24:2: [69, 1116]), (240:3: [17, 98, 9083]), … }	9
…	…	…	…	…
12340	dame	19	{(19:1: [34])}	1
12341	dark	1, 7, …	{(1:1: [9]), (7:5: [17, 27,1250,1255,1402]), … }	112
12342	day	4, 87, …	{(4:2: [9, 19]), (87:4: [18, 26,35,45]), … }	78
…	…	…	…	…
12939	mother	7, 9, …	{(7:3: [14, 216,255]), (9:1: [10]), … }	50
12940	motherboard	140, 250	{(140:1: [26]), (250:1: [7]), … }	10
12941	motion	2, 17, …	{(2:3: [14, 126,130]), (17:1: [117]), … }	38
…	…	…	…	…
22002	zydu	123	{(123:1: [4])}	1
22003	zymn	289	{(289:1: [1265]), … }	1

Figure 2.2: Inverted Index of an IR System

2.2.1 Token Preprocessing

Prior to index creation each document added to the store has to be preprocessed. Usually a list of words, numbers or other char-sequences is generated from the content of the document. This process is called tokenizing where according to certain rules the document stream is split in small coherent units providing semantic meaning (terms, e.g. words). Typically, separator characters like 'space' or 'new-line' mark the beginning of a new token. When tokenizing is applied to structured data, the meta-tags can be used to further classify the data. Document headings are usually treated more significant whereas the page authors‘ name might be less important with respect to the user‘s information need.
Taking the XML snippet 2.1 an XML-tokenizer can achieve the following list of terms if only the content of the title and text tag is factored into.

Listing 2.1: Exemplary XML-Snippet

```
<page>
  <title>Bayesianism</title>
  <ns>0</ns>
  <id>340</id>
  <redirect title="Atlas Shrugged"/>
  <revision>
    <id>74467467</id>
    <parentid>74467441</parentid>
    <timestamp>2006-09-08T04:22:33Z</timestamp>
    <contributor>
      <username>Rory096</username>
      <id>750223</id>
    </contributor>
    <comment>fix</comment>
    <model>wikitext</model>
    <format>text/x-wiki</format>
    <text xml:space="preserve">
      The probability of any event is the ratio between the value at which an expectation
           depending on the happening of the event ought to be computed, and the value of
           the thing expected upon its happening!
    </text>
    <sha1>rbfotal4dqrib0j7x7co9nid8hbmv74</sha1>
  </revision>
</page>
```

{Bayesianism, The, probability, of, any, event, is, the, ratio, between, the, value, at, which, an, expectation, depending, on, the, happening, of, the, event, ought, to, be, computed,, and, the, value, of, the, thing, expected, upon, its, happening!}
The content of all other tags and attributes are omitted and the 'space' character is used to separate the terms. Still, this list of terms is not well suited for inserting into an inverted index. Some terms start with upper case letters and exclamation marks and commas are still contained in the list. As such two terms with the same semantics are distinguished by the system and bias the results. To avoid this, all terms have to be *normalized*. This is the process of converting all terms to lower case and removal of special characters like punctuation marks. The implementation of normalization algorithms depends on the application and shows clear differences. In this case the exclusive use of lower-case letters and the removal of punctuation mostly results in the following list.
{bayesianism, the, probability, of, any, event, is, the, ratio, between, the, value, at, which, an, expectation, depending, on, the, happening, of, the, event, ought, to, be, computed, and, the, value, of, the, thing, expected, upon, its, happening}
To further improve the IR system, function words such as "and" or "the" shall be removed. A so called *stop-word* list is made available to the system containing all terms on the hit list. Thus, the index size is further decreased while no retrieval-relevant information about the document content is lost. Now the number of terms has been shrunk significantly.
{bayesianism, probability, event, ratio, value, expectation, depending, happening, event, ought, computed, value, thing, expected, happening}
Another improvement to be made is to reduce all words to a root form. Depending on the *stemming-algorithm* applied it does not necessarily need to be the exact linguistic morphological root. The famous "Porter-stemming-algorithm" [Por80] for instance converts the words "applies", "apply", "applying" to the common root "appli". The crucial point is that they all match the same root form and thus the number of different terms can further be reduced. An overview on common "stemmers" are listed in [Jiv11]. Applying a porter stemmer to the words above leaves behind the following list.
{bayesian, probabl, event, ratio, valu, expect, depend, happen, event, ought, comput, valu, thing, expect, happen}
The goal of preprocessing is to reduce the number of dictionary terms and still preserve

Information Entropy. Thus, the overall performance, effectiveness as well as efficiency, might be improved noticeably [BLY14] after these techniques are applied.
After all duplicates have been removed, the terms
{bayesian, probabl, event, ratio, valu, expect, depend, happen, ought, comput, thing}

can be added to the index and linked to the documents in the document-store. For each document a unique document ID (did) is assigned. The number of repetitive appearances of the id in the index equals the total number of terms in the document collection.

2.2.2 Boolean Retrieval

Having term to dictionary mapping in place, simple retrieval algorithms such as Boolean retrieval can already be applied. A Boolean retrieval model represents search queries as cascade of Boolean operations ("and" or "or") and returns a set of documents matching with no specific order. This model treats every document either as a match and takes it in account for the result set if the query term expression evaluates to true or excludes it otherwise. The query $''Boolean'' \wedge (''Retrieval'' \vee ''Expression'')$ would return only documents containing the term *"Boolean"* and in addition either *"Retrieval" OR "Expression"*. All other documents are ignored.
To allow for document ranking, this simple "1/0 - model" has to be extended with some sort of **weighting-score** for the terms. We will see that these scores are mainly influenced by collection properties like term distributions. A data structure extended by a position-list as shown in Figure 2.2 accommodates this demand. Each term refers to a set of values linking it to documents it can be found in (see 2.2.2).

$$TID \rightarrow (did, tf\ \langle pos1, pos2, \ldots \rangle) \tag{2.1}$$

Right after the did, the frequency of terms in the document, (term frequency (tf)) is stated followed by a list of positional offsets within the document. Additionally, for each term the overall number of documents containing the term, document frequency (df), can be stored separately. Such weighting parameters shape the algorithm to calculate document scores that form the basis of document rankings.

2.2.3 tf-idf - Values

Tf-idf is a broader term for a variety of weighting-functions used in practice, determining document scores that reflect the relative significance of a document with respect to the query terms. At least every term from the query, and depending on the ranking model, each term from the documents, must have a weight assigned to. Intuitively the significance of documents strongly relates to the number of occurrences of a term in a document (tf) - the more often a term is found in a document the more important it is - and secondly how many documents a term is contained in (df)[SB88]. Terms found in only few documents are more "specific and elective", i.e. comprise better information entropy [Jon72]. The twofold function tf-idf combines these two aspects. It can be seen as a product of the tf-function and the inverse document frequency (IDF). In most cases the tf value is not simply the term-count within the document. Pretending so, documents with many term repetitions gained importance linearly for each term occurrence in the document. Imagine a document containing a search term only once compared to a document with 3 findings. The overall weighing function would rate the second document 3 times more important compared to the first one. To alleviate this behavior a logarithmic growing term contribution seems more appropriate and have already proved themselves in practice. Hence, a tf-function has the following structure.

$$TF = \begin{Bmatrix} \log(f_{t,d}) & if & f_{t,d} > 0) \\ 0 & & otherwise \end{Bmatrix} \tag{2.2}$$

If we put the document-frequency in relation to the overall number of documents in the collection (N) we receive the probability to pick a document from the collection randomly including the term: $pd_t = \frac{df_t}{N}$ Likewise the Shannon Entropy [Sha48] the information content of the IDF function is expressed as the logarithm of the inverse of the probability function.

$$IDF = \log\left(\frac{N}{df_t}\right) \tag{2.3}$$

Above, these equations are derived from intuition mainly and crystallized out to meet the de facto standard form in practice. There are a variety of other slightly varying functions used in IR though. Apparently, in a dynamic data setting the IR system needs to keep track of the df over time. Otherwise, the entire document corpus has to be scanned to count terms in documents for every query or update of the document corpus, a generally infeasible solution, as results need to be ready in real time for many applications.
Note that the tf-function is document dependent whereas the IDF-function depends on the entire document collection. When the document corpus changes, the document weights

for all terms and documents affected have to be updated as well.

2.2.4 Proximity Measures

Furthermore, the IR architecture in figure 2.1 states a list of positional-offsets of terms within the documents. Some ranking models consider the presence of search-terms in proximity more important than such further apart. Therefore, the list of positional offsets is enough in most cases. This work will consider positional attributes only in one data model and solely take the quantity of offsets to compute the term frequency. Further information about the proximity can be found in [BCC10, pp. 60-63].

2.3 Ranking Models

The ranking models are held responsible for the quality of the retrieval rankings. Their task is to compute scores for the documents with respect to the user query entered and sort the results by score. A distinction of the different models can be drawn based on their mathematical basis. These are set-theoretic, algebraic and probabilistic models. Set-theoretic models deal with documents as sets of ords as binary variables evaluating to "1" if the term is found in the document and "0" otherwise. One representative, is the standard Boolean model introduced in the last section. Algebraic Models such as the vector space model handle queries and documents as term vectors where each dimension represents a specific search term the greater the number of occurrences in the collection and the affected document, the higher its value is. Simple vector multiplication allows for consistent rankings and partial matches of search terms in documents. The third category are the probabilistic models where probabilistic theorems (like Bayes) are used to calculate relevance scores. Well known representatives are the BM25 and all sorts of language models. Probabilistic and algebraic models' performance depend on the corresponding term weights.
All the presented IR models have only limited expressiveness and do not capture meta-data, phrases or document quality. Potential performance improvements coming from text structure (e.g. XML, headers) are ignored in favor of simplicity of the data model. A further discussion is presented in [CDM$^+$09].

2.3.1 OKAPI BM25

Ongoing development and research since the 60s led to the development of the BM25 term-weighting and document-scoring function in the early 90's. It is the best known representative and the fruit of research contributions to the classic Probability Relevance Framework (PRF) [RZ09]. The main concept behind PRF is referred to as the notion of relevance. Each query-document pair is assigned a "non-observable" value that reflects the relative significance (calculated by a probability function) of a document to a user query - the document score. Ordering the documents by their retrieval scores in descending order yields a result ranking. This confirms to the statement of the Probability Ranking Principle [RZ09, p. 338].

> "If retrieved documents are ordered by decreasing probability of relevance on the data available, then the system's effectiveness is the best that can be obtained for the data."

The BM25 can be seen as one manifestation of the tf-idf concept we discussed earlier which has been optimized by experts over a long period and it incorporates common characteristics of retrieval functions. This is why it suits for the query evaluation and experiments in the last chapter.
Consequently, a concise introduction is presented next, demonstrating the functional principle of the BM25 model. In Probabilistic IR models somehow the amplitude of the scoring function must reflect the probability users find a document relevant to a query. This is what Jones [JWR00] labels the basic question.

> "What is the probability this document is relevant to the query?".

We can put the calculation of positive relevance in the following mathematical formulation.

$$p(R = 1|D = d, Q = q) \tag{2.4}$$

or just $p(r|D,Q)$. R is a binary random variable which can take either the value $R=1$ (or simply r) if it is relevant or $R=0$ (or simply $\bar{r}$) otherwise. With D we mean a random variable describing all combinations of possible documents returned by the search engine. The space of all possible queries entered by users is the random variable Q. d and q are the actual query or document rankings respectively. Using the negative probability and applying the Bayes theorem $P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$ to statement 2.4 leads to the following equations of positive and negative relevance equations.

$$p(r|D,Q) = 1 - p(\bar{r}|D,Q) \tag{2.5}$$

$$\begin{aligned} p(r|D,Q) &= \frac{p(D,Q|r) \cdot p(r)}{p(D,Q)} \\ p(\bar{r}|D,Q) &= \frac{p(D,Q|\bar{r}) \cdot p(\bar{r})}{p(D,Q)} \end{aligned} \tag{2.6}$$

To simplify the equation, a monotonic order or rank-preserving log-odd function $logit(p) = \log\left(\frac{p}{1-p}\right)$ is wrapped around and after some algebraic transformation and applied simplifications we get the core of probabilistic retrieval functions.

$$\log \frac{p(D|Q,r)}{p(D|Q,\bar{r})} \tag{2.7}$$

Under the assumption that all terms are statistically independent of each others' and the relevance of a document depends on terms in the query only, a much simplified model, the *'Binary Independence Model'* [BCC10, p. 263], is derived.

$$\sum_{t \in (q \cap d)} \log \frac{p(D_t = 1|r) \cdot p(D_t = 0|\bar{r})}{p(D_t = 1|\bar{r}) \cdot p(D_t = 0|r)} \tag{2.8}$$

In this model only the presence or absence of terms from the query are used for calculations. In other words $V \setminus q$ are not factored in this formula. In this context D is the vector of binary values $\{d_1, d_2, d_3, .., d_n\}$ where each d_i is either 0 or 1. The notion $D_t = d_t$ makes explicit the value d_t conforms to the term t.

Neither of these two assumptions that yield to the concise form of presentation are realistic. The statistical independence of terms does not hold in practice. Taking for instance the two terms *"Alan"* and *"Turing"*, it is much more likely that another term such as *"enigma"* occurs compared to just random choice of the whole vocabulary. Following the same reasoning the assumption that only query terms account for the relevance of the document does also not reflect reality. Of course the existence of the

term *"enigma"* in the above statement should increase the document's relevance score too. It narrows the scope or domain of the documents calculated and refines the results. Retrieval models which factor in relevance feedback (from users or pseudo relevance when feedback is recycled) expand the set of query terms to those terms included in the relevant documents.
Although the two assumptions are unrealistic, the probabilistic models perform well.

Robertson Spark Jones Weighting Formulas

Robertson and Spark Jones [RJ94] derived estimates for the probability functions with respect to the a-priori knowledge.

$$\sum_{t\in(q\cap d)} w_t \to \sum_{t\in(q\cap d)} \log \frac{p_t(1-\bar{p}_t)}{\bar{p}_t(1-p_t)} \tag{2.9}$$

The left part of the equation follows the simple notions of weights (w_t) associated with terms. If the weights are rewritten according to equation 2.3.1 with $p(D_t = 0|r) = 1 - p(D_t = 1|r) = 1 - p_t$ and $p(D_t = 0|\bar{r}) = 1 - p(D_t = 1|\bar{r}) = 1 - \bar{p}_t$ we get the right side. Hence, the sum of logarithmic probability functions makes up the document score. For p_t we have to introduce appropriate estimates from the collection. These are the total number of documents in the collection N, number of relevant documents D_r, number of relevant documents containing term $D_{t,r}$ and actual number of documents containing term D_t.

$$p_t = \frac{D_{t,r}}{D_r} \text{ and } \bar{p}_t = \frac{D_t - D_{t,r}}{N - D_r} \tag{2.10}$$

substituting back

$$w_t = \log \frac{D_{t,r}(N - D_t - D_r + D_{t,r})}{(D_r - D_{t,r})(D_t - D_{t,r})} \tag{2.11}$$

this already reminds of the earlier stated IDF-function (cf. equation 2.3.1) because of the number of documents N in the nominator and the documents containing the term $D_{t,r}$ in the denominator. So far the model is not taking the term frequency t_f into account. Assuming the probability of term occurrence to be *Poisson distributed* [RJ94] the approximation, one of the most famous probability ranking functions - *The OKAPI BM25* - is derived. OKAPI is the name of the retrieval system, "BM" stands for Best

Match and 25 refers to the algorithm number within the family.

$$\sum_{i=1}^{n} \mathrm{IDF}(q_i) \cdot \left[\frac{f(q_i, D) \cdot (k_1 + 1)}{f(q_i, D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{\mathrm{avgdl}})} + \delta \right] \tag{2.12}$$

BM25F denotes an extended version of the algorithm considering the structure of the documents. For instance a query term found in the header of a page of the collection is rated more important than others in the text. A further discussion can be found in [RZ09, pp. 30-39].

2.4 Performance Evaluation

There are two different perspectives when assessing the performance of an IR system. The first one is generally known as the measure of *effectiveness* of a system. This conforms to the user's judgment how relevant the returned documents are according to the information need. These relevance measures are reliant to the power of the applied ranking module. The better the algorithm works the more relevant documents from the collection are detected and the fewer irrelevant documents added to the result set. Mathematically it is expressed as **recall, precision** and **f-measures** in terms and numbers.

The second perspective expresses the requirement to have the IR system implemented *efficiently*. Many design decisions in IR systems are owned to the experience of limited hardware resources. The index construction and maintenance for instance is aligned with the working memory disposable for IR-tasks with respect to the collection size expected. Assumed the number of dictionary terms exceeds the available memory of the IR system for the hash-based indices, disc-based approaches are employed with tremendous impact on query execution time. Such trade-offs were drawn at all component-levels and a variety of standards have emerged for many applications. Ranking models such as BM25 are not only designed to achieve the best effectiveness but also trade in simplifications for the model to reduce computational complexity. So the BM25 model spares term proximity, assumes statistical independence of terms and considers query terms exclusively to improve the model's efficiency while losing only little effectiveness. In this work the effectiveness of the retrieval model is only of peripheral interest. The aim of the work is to reproduce experiments and achieve equal result rankings. Evaluating the efficiency compared to other IR systems will be the crucial point. According to the application and available hardware the system is expected to return the user query in a respective time frame. Hardware demand and query execution time must sustain comparison with other IR systems. For all the experiments conducted the following parameters will be monitored and compared with state-of-the-art retrieval systems (e.g. Apache Lucene) [Fou12]

- Query Execution Time [sec]
- (Index) Storage Size [MB]

2.4.1 Challenges and Limitations of IR-Systems in Data Dynamics

In a static data environment where the number of documents and terms is fixed, the handling of the inverted index is straightforward. The text collection is parsed only once and a corresponding inverted index is created. No special care is needed for potential updates on the index and the efficiency of index construction and maintenance has only little priority because it is only set up once. In most applications the underlying data corpus is subject to change. Wikipedia logs numerous updates and creation of documents every second, libraries update their collection of books and journals and patent documents or accumulate new documents. Thus, IR systems are challenged to employ strategies how to react on document insertions, deletions or updates. Two common strategies to maintain dynamic indices are to

- Rebuild the entire index for the updated corpus.
- Build an index from the updated documents and merge it with the existing index.

If there are a lot of incremental insertions of documents but only a few updates or deletions expected, a merging strategy will be in favor. Yet, many random disc accesses and seeks are required to delete old documents and update the posting lists with new entries the situation can be different. Another aspect is the update interval policy. Web search engines are most likely to collect changed documents and flush the updates all at one go. The costs for update operations can be limited this way. Twitter at the further end strives for index update in the latency range of sub-seconds [Bus10] while dealing with several thousand "tweets" (=index updates) every second [Kir13]. Therefore, they adapted and extended the open source retrieval system *Apache Lucene* [Fou12] system to meet their requirements. Nevertheless, none of the established indexing and searching frameworks do support reproducible query results on dynamically changing document corpora. The results of the queries always reflect the current state of the inverted index. So if there are only minor changes - documents added, deleted or updated - made to the index in between launching two identical user queries the returned result rankings can differ.

The previous state of the index and especially of the term frequency values cannot be restored. This is the desired behavior in most search applications. In online search users are generally interested in up-to-date information. However, IR result rankings attributed to scientific studies need to be reproducible in order to meet foundational scientific standards. Likewise, legal administration could benefit from reproducible domain-specific searches in order to provide proof for the level of awareness - the information that was available at the time the query was launched. The lack of support and research for reproducibility in common IR systems is not entirely coincidental. In fact the way the data is structured and stored in the inverted index, a simple add-on to the system is not apparent. Somehow the IR system would have to keep track of all adaptions been made to the index to be able to restore previous index-states given at a certain point in time. But where should this information be stored if common IR-data structures like the inverted index, suffix-arrays or signature lists are applied? Time spans a new dimension for index growth and query performance. While the dictionary size asymptotically saturates with the cardinality of the vocabulary, the query execution time, memory consumption, disc storage consumption and size of online-indices rise in lockstep with the number of documents added [LZW06, pp.13-16][BCC10, pp. 228-255]. To enable reproducible searches, not only ingest, but also deletion and updates of documents would increase the index size as no previous posting list entries can be updated or deleted. Instead, an updated snapshot of the terms posting list would have to be created. Additionally, collection-related weights such as the term's document frequencies have to be maintained as well for every update on the collection. As an example, we consider a web page with 1000 words having one term from the beginning removed. As opposed to regular indexing the other 999 positional offsets have changed and cannot be overwritten in-place but instead a copy of the index-segment field, labeled with the corresponding timestamp is created. Hence, these 'slices' of fragmented indices needed to be merged with all other deltas of the inverted index in order to avoid exponential growth unlike storing a copy of the entire posting list for every update.

Apparently these extensions yield a gigantic growth of data, additional complexity for index maintenance and do not scale efficiently. In the next sections we discuss patterns how the edit history of a data collection can be versioned and timestamped efficiently. This so called data citation approach offers a solution to the scaling issues.

2.5 Conclusion and Summary

Search Engines like Google are the most famous representatives of IR systems. Their broadly known way of working outlines the characteristics of an IR system. A user's need for information is packed into a query. The IR system's task is to compute a ranking score for all documents (e.g. web pages) known to the system and return the ranking of the top most relevant documents in decreasing order. The heart of the system is formed by the inverted index data structure that interrelates all terms to the documents in the collection and maintains present terms and df. It allows for a direct look up of term and collection statistics facilitating the computation of the ranking scores efficiently. This architecture is matured for most applications but not applicable for reproducible result rankings in a dynamically changing document collection. The storage demand and/or query computation is expected to burst all limits. Another way of structuring data has to be found to realize reproducibility of result rankings.

CHAPTER 3

Reproducible Analytics on Dynamic Data Corpora

A long tradition and key principle of natural science research is to make experiments and published results reviewable. Hence, an experimental setup must be described and documented sufficiently detailed in order for experiments to be repeatable by independent scientists justifying the findings. This also applies to the data used in the experiments. There is evidence computer science is still lagging behind [CPW15][MLS12] because of legal and operational issues. Driven by a *Special Interest Group Management of Data (SIGMOD) Database Conference*[1] initiative in 2008 followed by Very Large Databases (VLDB) with a reproducibility track [2] tools and guidelines have been proposed how to improve repeatability in computer science experiments. Some issues are best practice guidelines how online repositories, built management systems, software tools to create artifacts (e.g. figures), development environment, virtual machines (also Code, Data, and Environment (CDE) [3]), test beds [ELR15] and scientific workflows (eg. VisTrails[4]) are to be used. The European conference on IR *ECIR* launched a new Track [HRF15] in 2015 focusing on reproducing retrieval experiments pointing towards emerging research efforts in *reproducible IR*. A necessary prerequisite for repeatable experiments is to make sure the experiments are launched on the same data basis. Therefore, particular

[1]SIGMOD Repeatability Track: `http://www.sigmod.org/sigmod-pods-conferences/mirrors/sigmod2008/sigmod_research.shtml.html`

[2]Conference on Very Large Data Bases: `http://www.vldb.org/2013/reproducibility_committee.html`

[3]CDE: `http://www.pgbovine.net/cde.html`

[4]VisTrails Scientific Workflow Mgmt. `http://www.vistrails.org/index.php/Main_Page`

attention is devoted the way data is set-up to determine equal laboratory conditions. Some considerations are:

- Source code for data-manipulating scripts or parsers used for the experiments published.
- Tokenization issues, stemming algorithms and stop-word lists applied.
- Precise processing workflow and tools used.
- The order the data is processed and fed into the system preserved.
- Unambiguously identifiable source data sets. (e.g. versioned and hashed database dumps)

In IR research, all these considerations are crucial in order to enable other researchers to precisely reproduce baseline experiments for comparison with own research results. But ranked lists in IR experiments cannot be reproduced accurately, if for instance, setup scripts to obtain inverted index data are not documented in detail, or during data setup slightly deviating steps are applied [YF16], which is, in the light of the complexity and size of data, difficult to avoid. As a result benchmarking the performance of new retrieval models is biased, when being compared to a sloppily reproduced baseline[AMWZ09]. There is striving for a generalisation of retrieval baselines[ACD+15] for better comparison. We are going to show that data citation approaches introduced in section 3.2 offer a solution to this problem.

3.1 Open Data Initiative

IR experiments are often launched on vast data collections or subsets of data collections such as the Text REtrieval Conference (TREC) collection[1], and thus, considerable installation time is needed with respect to the steps above. One way to address this issue is related to Open Science and Open Data initiatives. A central digital IR repository jointly used by IR researchers could spare individual setups. Whether and how central IR repositories [FBS12] could be realized is still subject to ongoing discussions. A public consultation on the European commission on Science 2.0 (also referred to as open science) to understand potential impact and desirable political actions [1] on the changing

[1]Text Retrieval Conference `http://trec.nist.gov/`

[1]Validation of the results of the public consultation on Science 2.0: Science in Transition: `http://www.eesc.europa.eu/resources/docs/validation-of-the-results-of-the-public-consultation-on-science-20.pdf`

modus operandi has addressed challenges to provide open access to research concerned with big data. A suggested action for policy intervention is to develop a interoperable infrastructure for open science in particular for big data solutions with the goal to improve framework conditions on data-driven science. [2] [3]. It concludes building interoperable and standardized European cloud platform for open access to publications and data. The access to research data is less developed across all EU countries than access to research publications [4] but it is acknowledged it is of emerging importance to find strategies tackling the problem of open access scientific data used in publications. Insufficient digital skills and a lack of proper infrastructure have been identified as root causes to the problem. And despite of its positive character, open data initiatives face resistance too. It produces cost to create and maintain public repositories and to format and document data [PV13] in order to fulfill requirements to open data standards. Furthermore, making experiment data publicly available exposes another source for challenging the results or scoop experiments planned in the future. Nonetheless, it has been proved open data improves both, the quality and acceptance of a paper [PDF07]. This is why there is a need to establish an infrastructure to make it just as easy to cite data as it is to cite journal articles or books.

Nevertheless, there is emerging effort introducing global interoperable frameworks for research data access by organizations such as the Research Data Alliance (RDA)[5]. These efforts demonstrate the increasing importance of dealing with data sharing challenges among which the *data citation* as a sub disciplines plays an important part.

3.2 Data Citation

The Future of Research Communications and e-Scholarship (FORCE11) community made a joint declaration of eight data citation principles[6] to understand its specific requirements. They are listed in the following paragraph.

[2] A Digital Single Market Strategy for Europe: `http://eur-lex.europa.eu/legal-content/EN/TXT/?qid=1447773803386&uri=CELEX%3A52015DC0192`

[3] REGULATION (EU) No 1291/2013 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL of 11 December 2013 establishing Horizon 2020 - the Framework Programme for Research and Innovation (2014-2020) and repealing Decision No 1982/2006/EC: `http://ec.europa.eu/research/participants/data/ref/h2020/legal_basis/fp/h2020-eu-establact_en.pdf`

[4] Access to and Preservation of Scientific Information in Europe: `http://ec.europa.eu/research/openscience/pdf/openaccess/npr_report.pdf`

[5] RESEARCH DATA ALLIANCE: `https://www.rd-alliance.org/about-rda/who-rda.html`

[6] Data Citation Synthesis Group: Joint Declaration of Data Citation Principles. Martone M. (ed.) San Diego CA: FORCE11; 2014 `https://www.force11.org/group/joint-declaration-data-citation-principles-final`

1. **Importance:** Data should be considered legitimate, citable products of research. Data citations should be accorded the same importance in the scholarly record as citations of other research objects, such as publications [ODCSPoC13].

2. **Credit and Attribution:** Data citations should facilitate giving scholarly credit and normative and legal attribution to all contributors to the data, recognizing that a single style or mechanism of attribution may not be applicable to all data.

3. **Evidence:** In scholarly literature, whenever and wherever a claim relies upon data, the corresponding data should be cited.

4. **Unique Identification:** A data citation should include a persistent method for identification that is machine actionable, globally unique, and widely used by a community.

5. **Access:** Data citations should facilitate access to the data themselves and to such associated metadata, documentation, code and other materials, as are necessary for both humans and machines to make informed use of the referenced data.

6. **Persistence:** Unique identifiers, and metadata describing the data, and its disposition, should persist – even beyond the lifespan of the data they describe.

7. **Specificity and Verifiability:** Data citations should facilitate identification of, access to, and verification of the specific data that support a claim. Citations or citation metadata should include information about provenance and fixity sufficient to facilitate verfiying that the specific timeslice, version and/or granular portion of data retrieved subsequently is the same as was originally cited.

8. **Interoperability and Flexibility:** Data citation methods should be sufficiently flexible to accommodate the variant practices among communities, but should not differ so much that they compromise interoperability of data citation practices across communities.

To tackle the issue of unique identification a variety of implementations of Persistent Identifier (PID)'s)[HK06] concept have been proposed. In the digital publications the DOI is one manifestation of such a persistent identifier. The basic idea behind is to assign unique IDs to self-contained data units and a central Registration Authority (RA) maintains a repository, linking the requested file for a provided PID. Thus, this way the 'link-rot problem' for static data units is solved, but it increases the complexity in managing huge numbers of data dumps to be persisted over a long period of time, lacking the flexibility desired in highly dynamic research and operational environments.

The specifity and verifiability principle addresses the need to cope with dynamic data. In settings where data evolves dynamically, records are added, amended or deleted, different users can have different views or access rights on the data source. Thus, different strategies how to cite subsets of a collection need to be found. The Research Data Alliance (RDA) has launched a working group[7] in 2014 to overcome the challenge of precisely identifying arbitrary subsets of dynamically changing data to make them citable. It devised 14 recommendations [RAvUP15] [RAvUP16] allowing citing data and reproducing arbitrary views on data as it existed at a particular point in time. According to these recommendations, data should be versioned and stored in a timestamped manner, any change on data must be marked with a timestamp and the PID are assigned to queries instead of to data sets only. Doing so facilitates unambiguous identification of arbitrary subsets of data from the collection. For example, if we assume we had removed only single term from a document at time T_1 from a collection applying principles of data citation, not the entire document needed to be replicated or version controlled, but the term itself is excluded from the result set when at time $T_2 > T_1$ this document is retrieved. Given the original query from the PID repository is applied, the previous document state is retrieved, including the term. A practical industrial implementation of the above concept can be studied at the Virtual Atomic Molecular Data Centre (VAMDC)[ZMD16] where data citation is employed to, inter alia, increase awareness and trace back data changes to employees.

Though the recommendations have been designed for all type of data, RDBMS systems lend themselves providing many requirements stated above out of the box. Data records can easily be enhanced with additional columns for versioned and temporal data. The concept of *primary key (PK)* constraints covers the ability to identify data records unambiguously, and finally, RDBMS offer a powerful query language SQL for selecting unique subsets of the data transparently. The SQL standard (since SQL-92) has strong support for temporal data types [Sno00] and various ways of handling temporal data are familiar. As such Pröll [PR13a] suggests expanding the primary key of every data record with a version attribute representing the temporal validity period and allowing records to evolve over time. All Data Manipulation Language (DML) statements are captured with their issuing timestamp to track back temporal changes in the RDBMS system. RDBMS and SQL query language is highly standardized and not vendor-specific, hence supports

[7]RDA Data Citation Working Group: `https://www.rd-alliance.org/groups/data-citation-wg.html`

the goal of data citation to be interoperable and flexible.

Figure 3.1 illustrates the concept.

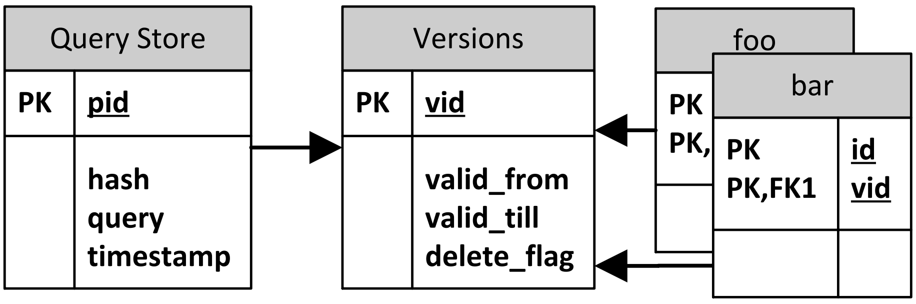


Figure 3.1: Data Citation Model for RDBMS (from [PR13b])

All SQL queries are stored in the query store. In order to provide proof for reproducible result sets a hash value computed on the original result is stored along with the query to verify the results for any later execution. The PID serves as PK for the query store and unique identifier of the sub sets of data to external applications. For any query the issuing timestamp is stored along with a version ID (vid) representing the period it is valid in. The temporal information can be valid or transaction time must be as fine grained as the environment's data update rate requires it to be. The vid of the versions table conforms to the PID and enables to uniquely identify a "slice" of the database records. Three different approaches how to design the operational data base schema to realize scalable data citation are presented in [PR13a].

Prerequisite for the realization is that unique subsets can be expressed (PK), insert, updates and deletes can be handled and nothing is actually deleted from the database unless legal requirements enforce it. A user requests a PID for a defined subset of the data collection. The query of the subset accumulates the issuing timestamp and is stored along with the vid in the versioning table. A hash computation, either of the entire result set, which is very expensive, or of the PK column and the header rows of the result set is also stored along with the query for result verifications. Querying the database with the previously received identifier issues the timestamped query against the system and returns the same result set just like with static data sources.

The versioning data can either be stored in the same table with the production data or at different levels "updated" records can be shifted to a special historized table. The later case is less intrusive because the data model does not have to be changed but more storage or more complex queries are required when data is partly outsourced to a history table.

Another challenge of data citation is to keep the original sort order of the records returned by a query. Records should therefore also have "order by" clauses incorporated on columns with an unambiguous order. Each kind of non-determinism in queries or database procedures is not supported as it makes reproducibility impossible.

The structured way to process data could indeed be the answer to the reproducibility issue in IR from a conceptual point of view. On the downside a massive performance bottleneck caused by the supplementary RDBMS layer and the fashion how the document store is integrated and the vast number of tuples to be processed, is expected. How column-store can obliterate these performance drawbacks is presented in chapter 4.

3.3 Conclusion

Scalable data citation is referred to as a way of unambiguously linking to a subset of a data collection in a dynamic data environment. Structured data sources like RDBMS ease the implementation as such data records can easily be extended by temporal information. In case of the retrieval application, new or changed documents have a new timestamp assigned to the newly created data to reflect the changes. So later queries can extract the most recent result as well as all intermediate historic results.

CHAPTER 4

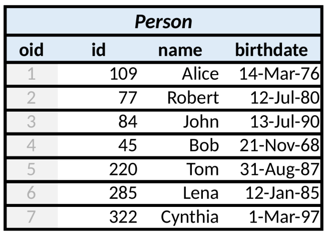
Column-Stores as a Backend to IR

One distinction, that is frequently drawn between column-store and row store is the area of application they are allotted to. Row stores are usually deployed in Online Transaction Processing (OLTP) when frequent updates and insertions of single records occur. One tuple or row is treated as a coherent unit and hence read and processed as unity - the *tuple-at-a-time* pardigm. They are far more common and widespread in operational business. Column-stores have arised from Online Analytical Processing (OLAP) and though first development goes back to the 70's, it has only recently gained more attention. They are optimized for data warehouse applications when a very large number of records are expected and aggregations on single columns are computed frequently. It is also referred to as the *column-at-a-time* processing paradigm as operations process data on one column at a time only. Benchmarks show clear performance advantages of column-stores over row stores in OLAP applications [AMH08]. Section 4.2 points out that there exist structural similarities from IR indices and the OLAP *Star Schema* as both have to deal with aggregation and grouping functions over many records and jointly use fact and dimension tables.

4.1 Column-Store Architecture

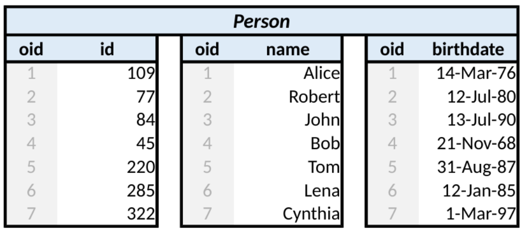
One of the main characteristics of the column-store architecture is the Decomposition Storage Model (DSM) [CK85]. Relational tables are decomposed into their columns (attributes) and stored as 2-ary (surrogate, value) arrays as shown in Figure 4.1.

Figure 4.1: Comparison of Row Store to Column Store Storage Model



Person			
oid	id	name	birthdate
1	109	Alice	14-Mar-76
2	77	Robert	12-Jul-80
3	84	John	13-Jul-90
4	45	Bob	21-Nov-68
5	220	Tom	31-Aug-87
6	285	Lena	12-Jan-85
7	322	Cynthia	1-Mar-97

(a) NSM n-ary array, storing data row-wise



Person					
oid	id	oid	name	oid	birthdate
1	109	1	Alice	1	14-Mar-76
2	77	2	Robert	2	12-Jul-80
3	84	3	John	3	13-Jul-90
4	45	4	Bob	4	21-Nov-68
5	220	5	Tom	5	31-Aug-87
6	285	6	Lena	6	12-Jan-85
7	322	7	Cynthia	7	1-Mar-97

(b) DSM 2-ary array, storing column-wise

The n-ary storage model (NSM) stores values within one row or one tuple block-wise. In contrast, in the DSM, surrogate-value pairs for each attribute are stored. In case of modern column-stores ordinary arrays fill the role where the surrogates are simply arrayed indices [ABH+13] (cf. Sec. 4.2). Array computation facilitates the architecture of modern computers better than the volcano principle [Gra94] of conventional row stores where algebraic operations are implemented as iterators and tuples are processed in a piped fashion one after another. In column-stores all values of the same column are processed in a dense array-like structure streamlining computation on column specific operations. They benefit from high instruction locality of operations. When only one column at a time is processed, values fit in processor cache thus the number of cache misses and far more expensive memory access is reduced. Furthermore, the Memory Management Unit (MMU) provides a constant ($\mathcal{O}(1)$) positional database lookup time when used with arrays.

Generally not all table columns are equally important. Knowing that memory is still the most critical resource in RDBMS, attributes not affected by the query can simply be omitted and are not loaded into memory at all. Let's take a simple *select* statement on a single column as an example. A column-store loops over the values of the column and extracts a subset of matched values and their surrogates but does not take account of other attributes of the relation. But in return re-constructing tuples from surrogate value pairs causes a substantial overhead.

Furthermore, storing values of the same kind within a column gives rise to much more efficient compression algorithms which helps to fruther reduce the amount of memory in use.

4.1.1 MonetDB Architecture

The architecture of the MonetDB system [MKB09] consists of three layers. The front layer compiles the user query language (SQL, XQuery, SPARQL) to assembler language MAL. The queries are translated to MAL and in the first place the execution plan is optimized, e.g., selections pushed before joins. The global cost model utilized in the volcano pipe of traditional row store cannot be applied due to search space explosion caused by vertical fragmentation and is replaced by a more dynamic approach considering local information about the data-collection [IGN$^+$12] during query execution. Depending on the selectivitiy of column data, sortedness, existence of nil values or parallelism MonetDB choses the fastest exuecution strategy on the fly. If, for example, data in a column is sorted then binary search strategy makes data faster accessible. The backend is a virtual machine MAL interpreter that produces a series of simple C-like instructions on so called Binary Association Table (BAT), the equivalent to surrogate-value principle. BAT are direct translations of columns and are stored as memory mapped files in the main memory or on the hard drive. As shown in Figure 4.1 for every relation R with k attributes, k BAT's are created. The object identifier (oid) is the head or surrogate of the memory mapped file (see Figure 4.2). The tail column (value) either contains the value directly, or in case of character data, contains an positional offset of the concatenated string supporting dictionary encoding. When maintenance costs for dictionary encoding become to large the optimizer falls back to non compressed concatenated string. The output of each BAT operation (bulk-operation) is another BAT.

To exploit additional potential of a column-store *MonetDB*[1] reconsidered many design decisions of traditional RDBMS and a new engine was developed incorporating the following features.

- Column-stores are widely used in business analytics applications. In this area millions of records are expected, new records are written in bulks at the end of the tables, the nature or quality of data is priorly unknown and the adaption of regular indices is not satisfactory. Instead, "cracked" column representations [IKM07] of the original columns are created containing subsidiary information about range partitions of the columns. Making use of this *adaptive indexing* the system dynamically extends the knowledge while processing queries. If a query is launched on the cracked database, the tables algorithm knows which clusters of the original column contains the demanded values and which can be skipped. The knowledge about the data in the tables is continuously refined unless the table

[1]MonetDB: `https://www.monetdb.org/`

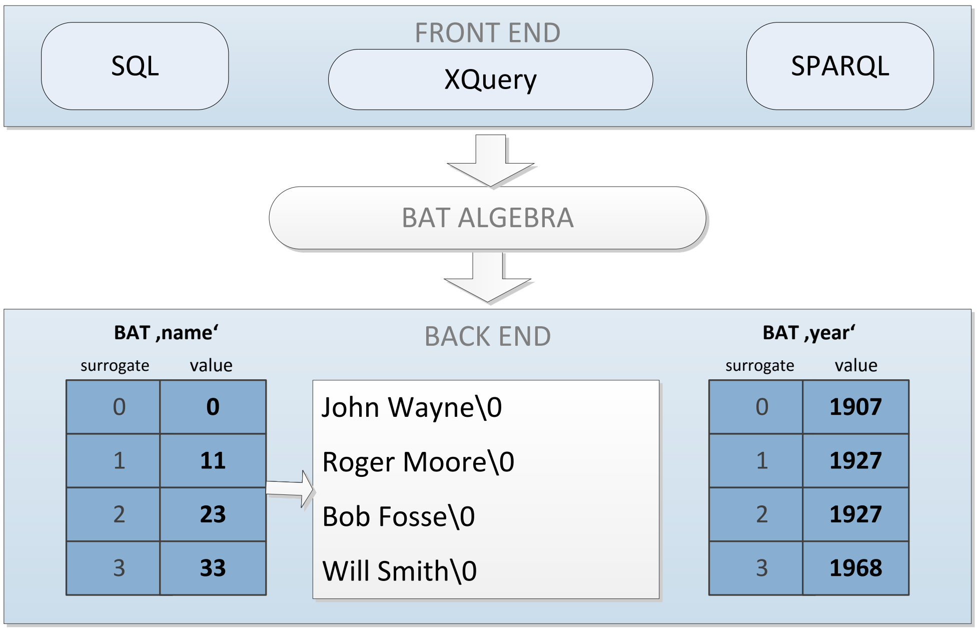


Figure 4.2: MonetDB Architecture (from [MKB09])

content changes and the temporary index is invalidated again. This offers huge performance opportunities on select operations, aggregations, join operations and especially the tuple reconstruction [IKM09] - one of the most frequent operations in column stores.

- *Cache conscious algorithms* are trimmed to exploit the pipelined memory hierarchy of modern computer systems. For join operations a Radix-Cluster algorithm is applied [MKB09] to reduce random access produced by ordinary hash joins. MonetDB clusters values in the join column according to their least significant bits. The goal is to have the values clustered in portions small enough to fit in processor cache. To achieve this, large columns clusters are built on multiple consecutive passes. Values belonging to the same clusters are than subject to the partitioned hash join.

- Complex queries are split into small iterative tasks to be executed in a chain of BAT operations. MonetDB require full materialization of intermediate results, i.e. each operation produces another BAT and the final result is pieced together in the end. If intermediate results exist anyway it is obvious they can be reused for later queries or operations. Therefore, the column-store have to track operations, monitor their lifespan and decide whether query results can be recycled [IKNG09].

4.2 Column-Stores for Retrieval Tasks

There have been approaches to merge IR with traditional RDBMS data structures. A driver is the flexibility of data management in RDBMS systems and the necessity to improve the effectiveness of searches in database systems. A comprehensive overview can be found in [CRW05]. While acceptable results were achieved with set theoretic models (Boolean model, see Sec. 2.3) a successful implementation of ranking models failed due to the inefficient implementation of a inverted-index like data structure. Recent advancement in column-store technology has changed the situation here. BjØrklund [BGT09] underlines existing synergies between inverted indices in IR and the OLAP processing column stores have been optimized for. Mühleisen [MSLdV14] launched IR queries on MonetDB and the commercial spin of Actian Vector [1][ZB12] and compared the performance to established IR systems like Apache Lucene. The benchmark revealed very encouraging results. It turned out to be effective and the latency was only insignificantly worse compared to established IR systems. But in addition it offers a clear separation of concerns between the exploration of retrieval models - the SQL queries - and the actual data layer. Moreover, RDBMS systems are more flexible to explore data offside the given track. For example all kind of statistics can be computed with modest effort because aggregations and groupings are standard operations on RDBMS whereas in many cases additional iterative code is required to achieve the same results in traditional IR systems. The following paragraph is a short recap of the powerful concept introduced in [MSLdAV14] [MSLdV14]. The ER diagram in Figure 4.3 sketches the relations.

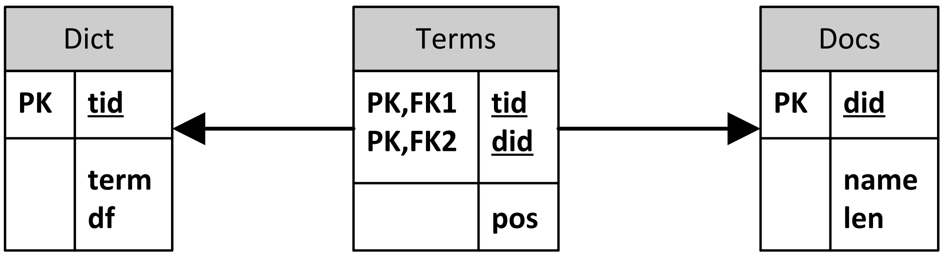


Figure 4.3: The Basic RDBMS-Schema for Ranked Retrieval

The concept of inverted indices was translated to a RDBMS system by creating three tables. The first table is the dictionary and contains the vocabulary of the text collection. The attribute term ID (tid) conforms to a unique term ID, *name* is the actual term name

[1]Actian Vector: `http://www.actian.com/company/news-and-events/press-releases/actian-vsmp-scalemp/`

and df is the number of documents containing this term - the document frequency. The dictionary table conforms to the inverted index in IR systems (see Sec. 2.2). Another table holds the information about the documents in the collection, that is the name of the document, the did and the length. The relation "contains" is represented by the terms table. It interrelates the dictionary with the documents table as the sequence of terms in the document. Thus, for a document of length hundred, a hundred terms with tid, did and the position of the terms within the document *pos* are added to the terms table. This relation between the document collection and the dictionary can be seen as the posting list of the IR system.
Consider the following two documents with only a simple sentence each

- **Document 1**: In Central Asia there exists a wild mouse
- **Document 2**: A mouse is in the box

After the content was copied into the database above the resulting tables look as stated in table 4.1.

Table 4.1: Exemplary Index Containing Two Documents

(a) Dict Table

tid	term	df
10	in	2
20	central	1
30	asia	1
40	there	1
50	exists	1
60	a	2
70	wild	1
80	mouse	2
90	is	1
100	the	1
110	box	1

(b) Terms Table

tid	did	pos
10	1	1
20	1	2
30	1	3
40	1	4
50	1	5
60	1	6
70	1	7
80	1	8
60	2	1
80	2	2
90	2	3
10	2	4
100	2	5
110	2	6

(c) Docs Table

did	name	len
1	Doc 1	8
2	Doc 2	6

This little example with only two documents in table 4.1 indicates how table growth evolves. The dictionary table is restricted to $|V|$ the quantity of vocabulary terms in the document collection and would most likely approach V in logarithmic fashion. The document table grows linearly with the number of documents added. Whereas the terms table grows also linearly but with the average document length. To provide an order of magnitude, the benchmark conducted on 45 million documents from the ClueWeb12 dataset [1] with an average document length of 513 words bloated the terms table to more than 23 billion records.

In the next step the OKAPI BM25 model (cf. Sec. 2.3.1) was translated to a SQL 4.1

Listing 4.1: BM25 Algorithm Translated to Conjunctive SQL (from [MSLdAV14])

```
WITH
qterms AS
 (SELECT termid, docid
 FROM terms
 WHERE termid IN ('tid1','tid2','tid3')),
subscores AS (
 SELECT docs.docid, len, term_tf.termid, tf, df, (log((N-df+0.5)/(df
    +0.5))* ((tf*(1.2+1)/(tf+1.2*(1-0.75+0.75*(len/avg(len)))))) AS
     subscore
 FROM (SELECT termid, docid, COUNT(*) AS tf
      FROM qterms
      GROUP BY docid, termid) AS term_tf
 JOIN (SELECT docid
      FROM qterms
      GROUP BY docid HAVING COUNT(DISTINCT termid) = 3) AS cdocs
 ON term_tf.docid=cdocs.docid
 JOIN docs
 ON term_tf.docid=docs.docid
 JOIN dict
 ON term_tf.termid=dict.termid)
SELECT name, score
FROM (SELECT docid, sum(subscore) AS score
    FROM subscores GROUP BY docid) AS scores
JOIN docs
ON scores.docid=docs.docid
ORDER BY score DESC LIMIT 1000;
```

[1]ClueWeb12 Dataset `http://www.lemurproject.org/clueweb12.php/`

From line 3 to 5 a view of the table is created filtering terms with the specified tid. These tuples in the view *qterms* is further cut down from line 8 to 10 by removing duplicate values tid, did due to applying an aggregate function count. The 'count values' conforms to the term frequency tf (cf. 2.2.3) and could be pre-computed. The benchmark recommends doing so if the applied scoring algorithms do not factor in term offsets or proximity measures. The inner join in line 11 to 13 limits the clause to documents containing exactly the number of terms (three in the present case). By joining the documents with the dictionary table we obtain the actual df value. Having all intermediate results available, the scoring function of BM25 is computed in line seven for every tuple in the record. From line 19 onwards the documents are ordered by their ranking score.

4.3 Conclusion

Conventional row store work on a tuple at a time principle, in other words to reduce IO access relational operators process tuples sequentially. While this is the method of choice in OLTP settings when comparably small number of records are read, deleted or updated, it is slow in OLAP tasks such as aggregation of large volume data. Column-stores are better suited for this purpose. They only share the idea of relational representation of data and the familiar SQL front end. The kernel of column-stores works completely different. Instead of storing rows in their tuple representation block-wise, tables are decomposed into their columns represented by surrogate-value pairs and stored individually. In case of the column-store MonetDB, which proved to work for IR prototyping and will further be used for the experiments, an 2-ary array (BAT) stores the content of a column. An aggregation on a column conforms to a computation of dense array values reducing cache misses and is better suited for modern processor architectures. Data compression is more effective too because data in one column usually has similar data format and coding schemes can more easily be exploited. On the downside decomposed storage of the columns yield to an overhead in tuple reconstruction to reassemble the relational tuples and DML statements are generally more expensive as several array data structures have to be manipulated. All intermediate results like a join of two columns are fully materialized, i.e. BAT memory mapped files are written to the harddisk or just kept in memory. The column-store accumulates knowledge about the value hierarchies in the columns as a by-product to query execution and thus later queries benefit from earlier operations. So column-stores improve query performance automatically whereas row stores mostly benefit from caches and dedicated indices. As retrieval applications on

RDBMS need to compute document statistics, the most expensive operations are column aggregations. It has been shown, three tables are sufficient as the baseline of IR on a RDBMS. The dictionary table represents the inverted retrieval index. The documents table holds all the meta information like name, length and title of the documents. The linking between the two tables is realized by the terms table. It holds the actual document data by referencing the tid with the did values. This setup scales dynamically and can deal with large volume data.

CHAPTER 5

Versioned Column-Store Data Models

To accommodate the data citation requirements we have identified six Versioned Column-Store Models (VCSM) extending the data model from section 4.1, facilitating reproducible results regarding best practice methodology in scalable data citation (cf. section 3.2). Essentially information about the version of the database record and timestamps have to be added as an extension to the unique identifiers for each row. The vital question is how to monitor the tf-idf values in the collection. For every change in the document collection those values are updated but a history of all previous states of theses measures needs to be sustained. In contradistinction to the setting from Mühleisen (see 4.3) the df value cannot be precomputed and stored along with the dictionary term. Instead, they need to be version controlled or computed as part of the query. The tf value depends on the document only and can be computed more simply by counting occurrences within one document. Aside with the retrieval model, adapted SQL queries for IR tasks are presented.

The first two models presented are comparably simple in structure. The basic structure of the IR model presented by Mühleisen [MSLdV14] is preserved and timestamps are added to the documents table to enable document versioning. Only the df value in the dictionary table vanished thus, the df values have to be computed as part of the retrieval query. In the first model the tf value must also be computed as there exists no tf column in the terms table. The third approach reintroduces the written df value in the dictionary by extending the second model with a dictionary history table. The fourth and fifth model aim at eliminating potentially expensive join operations and denormalize document and dictionary tables to the terms table. And finally VCSM6 trades in dedicated version numbers for timestamps. These numbers are maintained and assigned in a separate versions table.

5.1 VCSM1 Versioned Baseline Model

The first approach enriches the basic model by only adding version and timestamping information to the document table and keeping the basic structure of the other tables untouched. A further discussion concerns the df in the dictionary table. The table could hold the latest document frequency value for every term. This would potentially speed up the retrieval for non-timestamped queries but cause significant update costs in return. As the query for the non-timestamped retrieval algorithm is similar to the database schema depicted in figure 5.1. The df would have to be versioned separately and the df value is not persisted but dynamically computed as part of the retrieval query. The values highlighted in bold letters have to be set, other values are nullable. In case of the *removed* column, documents when being inserted do not have a removed timestamp assigned. Only after they get updated the value on the removed attribute is being set.

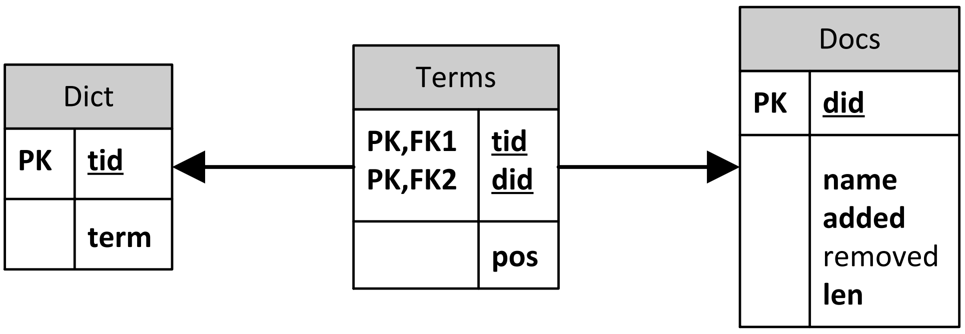


Figure 5.1: Versioned Baseline Model DB Schema (VCSM1)

The dictionary consists now of only two columns, the tid and the *term* column. Technically the dictionary table is not necessary at all if the dictionary term name was used as a unique identifier in the terms table. But the terms table has the most columns and grows faster than the dictionary table and identifiers can be small numerical values whereas string-like representation of the term name are stored using dictionary encoding.

The BAT file keeps pointer implementation to the character offsets within the String "blob". As a result the storage space of strings are relatively low, but significant expenses for maintenance are inevitable when the dictionary becomes vast. Another point for preserving the dictionary table is the reduced costs for dynamic indexing in first place as the dictionary table is expected to be smaller and less volatile making more efficient join operations to the terms table on numerical term identifiers possible.
Three new attributes extend the document table. First, two timestamps are added in order to check the validity period. These are the *added* and *removed* attribute in the documents table. For the *removed* column, *NULL* values are allowed so that the latest records of a document can be found easily. Naturally, unique records of the terms table are identifiable by the tid, did and the *pos* attribute because multiple occurrences of the same term in a document are only distinguishable by their positions.

5.1.1 BM25 Query for VCSM1

To achieve ranked retrieval results according to the BM25 retrieval model, the following adaptions to the query in Figure 4.1 were made. The df value is not in place anymore and is computed dynamically. Furthermore, time components are added to the query. All documents deleted before or not yet existent at query time are not taken into account. In addition, the restriction to conjunctive queries is not carried forward for the following reasons. A disjunctive query approach is more aligned to traditional IR and increases the number of documents for the result. We observed significant performance drawbacks on the baseline experiments, though the disjunctive approach is expected to be slower due to its larger result sets.

Listing 5.1: BM25 Retrieval on the Versioned Baseline Model

```
WITH
/* filter valid documents */
qdocs AS (SELECT * FROM docs WHERE added <= %stamp% AND (removed IS
    NULL OR removed > \%stamp\%)),
/* valid terms containing one of the search strings */
qterms AS (SELECT terms.tid, terms.did, terms.vid, tdic.term FROM
        (SELECT tid, term FROM dict WHERE term IN ('%term1%', '%term1%
           ', ..., %termn%)) AS tdic
        JOIN terms ON terms.tid = tdic.tid
        JOIN qdocs ON qdocs.did = terms.did AND qdocs.vid = terms.vid
           ),
/* average document length (avg(len)) */
```

```
10 len_avg AS (SELECT avg(len) AS anr FROM qdocs),
11 /* total number of documents (N) */
12 doc_nr AS (SELECT count(*) AS tnr from qdocs),
13 /* frequency of terms in documents (tf) = term frequency */
14 term_tf AS (SELECT tid, did, vid, COUNT(*) AS tf FROM qterms GROUP BY
       tid, did, vid),
15 /* compute number of documents containing search term (df) */
16 term_df AS (SELECT tid, count(tid) AS df from term_tf GROUP BY tid),
17 /* compute document term scores */
18 subscores AS (SELECT qdocs.did, qdocs.vid , qdocs."len", term_tf.tid,
       term_df.df, term_tf.tf, (SELECT tnr FROM doc_nr) AS n, (SELECT
     anr FROM len_avg) as av,(log(((SELECT tnr FROM doc_nr) - term_df.
     df + 0.5)/(term_df.df + 0.5)) * term_tf.tf * (1.2 + 1) / (term_tf.
     tf + 1.2 * (1 - 0.75 + 0.75 * ((qdocs."len")/((SELECT anr FROM
     len_avg)))))) AS subscore
19         FROM term_tf
20         JOIN qdocs ON term_tf.did=qdocs.did AND term_tf.vid = qdocs.
              vid
21         JOIN term_df ON term_df.tid = term_tf.tid)
22 /* summing up document scores and order by score descending */
23 SELECT subscores.did, subscores.vid, sum(subscores.subscore) AS rnk
24 FROM subscores GROUP BY subscores.did, subscores.vid ORDER BY rnk
     desc LIMIT 1000;
```

The selection of valid documents according to their time span of validity takes place in line three. This selection is the basis for the average document and number of document calculation in line eleven and 12. Next the tid values are picked from the candidate selection in the dictionary table in line six and joined with the terms and documents table. The **qterms** view in line five represents a list of all timely terms. This list is further refined and statistics about the tf and df are computed in line 14 and 16 respectively. The ranking and ordering from line 18 onwards is only changed for replacing hard coded parameters by dynamic value assignment.

5.2 VCSM2 Tuned Versioned Baseline Model

The second approach can be seen as an enhancement to the minimal model for specific retrieval tasks where term proximity measures are not relevant (e.g.BM25). Here, the *pos* attribute is replaced by the tf value yielding a reduced storage footprint as multiple term occurrences in a document are reduced to a single tuple. Furthermore, the tf is stored and does not need to be constructed by the retrieval query. Apparently this optimization is in favor over the minimal model 5.1 for retrieval algorithms not taking the proximity of terms in documents into account. The database schema is shown in figure 5.2

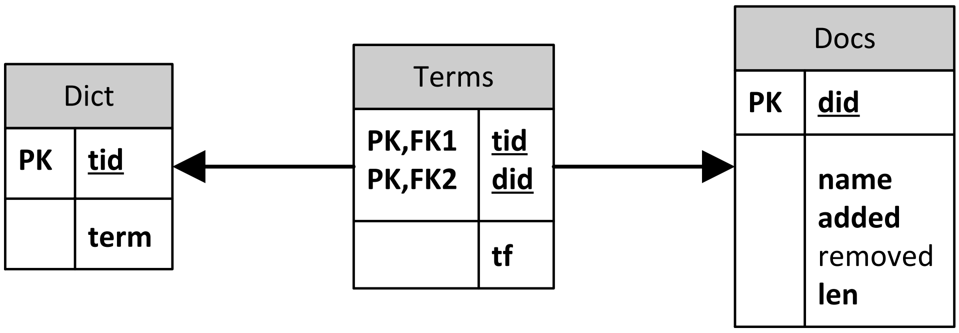


Figure 5.2: Tuned Versioned Baseline DB Schema (VCSM2)

5.2.1 BM25 Query for VCSM2

As a minor adjustment to query 5.1.1 we get the tf value from the terms table directly and line 14 becomes redundant.

Listing 5.2: BM25 Retrieval on Tuned Versioned Baseline Model

```
WITH
/* filter valid documents */
qdocs AS (SELECT * FROM docs WHERE added <= timest AND (removed IS NULL OR removed >= timest)),
/* valid terms containing one of the search strings */
qterms AS (SELECT terms2.tid, terms2.did, terms2.vid, terms2.tf, tdic.term FROM
        (SELECT tid, term FROM dict WHERE term IN ('%term1%', '%term1%', ..., %termn%)) AS tdic
        JOIN terms2 ON terms2.tid = tdic.tid
        JOIN qdocs ON qdocs.did = terms2.did AND qdocs.vid = terms2.vid ),
/* average document length (avg(len)) */
len_avg AS (SELECT avg(len) AS anr FROM qdocs),
/* total number of documents (N) */
```

```
doc_nr AS (SELECT count(*) AS tnr from qdocs),
/* compute number of documents containing search term (df) */
term_df AS (SELECT tid, count(tid) AS df from qterms GROUP BY tid),
/* compute document scores */
subscores AS (SELECT qdocs.did, qdocs.vid , qdocs."len", qterms.tid, term_df.df, qterms.tf, (SELECT tnr FROM doc_nr) AS n, (SELECT anr FROM len_avg) as av,
(log(((SELECT tnr FROM doc_nr) - term_df.df + 0.5)/(term_df.df + 0.5)) * qterms.tf * (1.2 + 1) / (qterms.tf + 1.2 * (1 - 0.75 + 0.75 * ((qdocs."len")/((SELECT anr FROM len_avg)))))) AS subscore
         FROM qterms
         JOIN qdocs ON qterms.did=qdocs.did AND qterms.vid = qdocs.vid
         JOIN term_df ON term_df.tid = qterms.tid
/* suming up document scores and order by score descending */
SELECT subscores.did, subscores.vid, sum(subscores.subscore) AS rnk
FROM subscores GROUP BY subscores.did, subscores.vid ORDER BY rnk desc LIMIT 1000;
```

5.3 VCSM3 Dictionary History

The most expensive operation is the computation of the df value. For the previous two schemes an aggregation over unique documents in the terms table are computed for every term in the query. This approach aims at eliminating this expense while causing as little storage increase as possible. The idea is to introduce an additional table which stores the tid along with the df value at the time located by two timestamps. This way only a single tuple qualifies for a query request making range queries and aggregation superfluous. Unfortunately, the dictionary history table grows as fast as the terms table for incremental document insertions or updates, giving rise to performance drawbacks caused by search space explosion and additional storage required for the history table. Figure 5.3 illustrates the schema.

5.3.1 BM25 Query for VCSM3

Like with query 5.2 the tf is stored in the terms table. In addition, the df is derived from joining the dictionary history table with the dictionary table and sorting out outdated values in line nine by having applied the same timestamp as for the document selection.

Listing 5.3: BM25 Retrieval on the Dictionary History Model

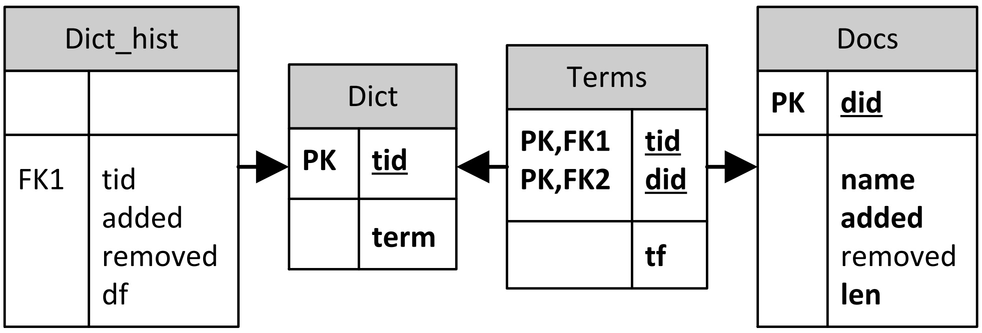


Figure 5.3: Dictionary History DB Schema (VCSM3)

```
WITH
/* filter valid documents */
qdocs AS (SELECT * FROM docs WHERE added <= timest AND (removed IS
    NULL OR removed > timest)),
/* filter terms containing at least one of the search strings */
qterms AS (SELECT terms2.tid, terms2.did, terms2.vid, dict.term,
    terms2.tf, dict_histu.df, qdocs."len" FROM dict
   JOIN dict_histu ON dict_histu.tid = dict.tid
   JOIN terms2 ON terms2.tid = dict.tid
   JOIN qdocs ON qdocs.did = terms2.did AND qdocs.vid = terms2.vid
   WHERE dict_histu.added <= timest AND (dict_histu.removed is null
       OR dict_histu.removed > timest) AND term IN ('%term1%', '%term1
       %', ..., %termn%)),
/* average document length (avg(len)) */
len_avg AS (SELECT avg(len) AS anr FROM qdocs),
/* total number of documents (N) */
doc_nr AS (SELECT count(*) AS tnr from qdocs),
/* compute document scores */
subscores AS (
SELECT qterms.did, qterms.vid , qterms."len", qterms.tid, qterms.df,
    qterms.tf, (SELECT tnr FROM doc_nr) AS n, (SELECT anr FROM len_avg
    ) as av,
 (log(((SELECT tnr FROM doc_nr) - qterms.df + 0.5)/(qterms.df + 0.5))
    * qterms.tf * (1.2 + 1) / (qterms.tf + 1.2 * (1 - 0.75 + 0.75 * ((
    qterms."len")/((SELECT anr FROM len_avg)))))) AS subscore FROM
    qterms)
```

```
/* suming up document scores and order by score descending */
SELECT subscores.did, subscores.vid, sum(subscores.subscore) AS rnk
FROM subscores GROUP BY subscores.did, subscores.vid ORDER BY rnk
    desc LIMIT 50;
```

5.4 VCSM4 Persisting df

Because of the rapid growth of the dictionary history table in lockstep with the terms table for incremental updates the storage footprint is considerable in VCSM3 (cf. 5.3). Four additional values are stored for every term and update in the dictionary history table. Two of the attributes, namely the *added* and the tid attribute could be spared if the df value was stored in the terms table instead. The tid is already there and the timestamp of addition can be derived from the document table. Only the invalidation time *rem_df* is needed to infer the valid df value. Figure 5.4 presents the according DB schema. The composition of the relation is not intuitive because the actual df value is independent of the terms relation itself. The design aims solely at storage space reduction due to exploitation of attribute sharing potentials and avoiding additional join operation as necessary with VCSM3 (cf. section 5.3).

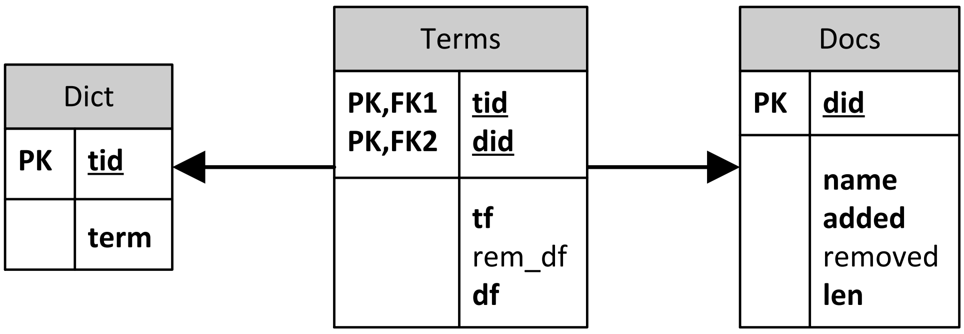


Figure 5.4: Persisting df DB Schema (VCSM4)

5.4.1 BM25 Query for VCSM4

The main difference to the historized dictionary model 5.3 is the computation of the df value. Instead of consolidating a separate table, the intermediate results of *qterms* can be recycled to gain access to the document frequency. First the candidate selection for qualified terms takes place in line #5 just like in the previous approaches. The attribute *rem_df* filters out now, by a given timeframe, the remaining tuples to a single match on all tids. These values are joined for scoring in line 18.

Listing 5.4: BM25 Retrieval on the Persisting df Model

```
1  WITH
2  /* filter valid documents */
3  qdocs AS (SELECT * FROM docs WHERE added <= timest AND (removed IS
      NULL OR removed > timest)),
4  /* filter terms containing at least one of the search strings */
5  qterms AS (SELECT terms3u.tid, terms3u.did, terms3u.vid, terms3u.
      added, terms3u.removed, terms3u.tf, qdocs."len", terms3u.df as df
6  FROM terms3u
7     JOIN qdocs ON qdocs.did = terms3u.did AND qdocs.vid = terms3u.vid
8     JOIN dict ON terms3u.tid = dict.tid WHERE dict.term IN ('%term1%',
          '%term1%', ..., %termn%)),
9  /* compute document frequency */
10 term_df AS (SELECT tid, df FROM qterms WHERE qterms.rem\_df IS NULL
      OR qterms.rem\_df > timest),
11 /* average document length (avg(len)) */
12 len_avg AS (SELECT avg(len) AS anr FROM qdocs),
13 /* total number of documents (N) */
14 doc_nr AS (SELECT count(*) AS tnr from qdocs),
15 /* document term scores */
16 subscores AS (SELECT qterms.did, qterms.vid , qterms."len", qterms.
      tid, term_df.df, qterms.tf, (SELECT tnr FROM doc_nr) AS n, (SELECT
       anr FROM len_avg) as av,
17 (log(((SELECT tnr FROM doc_nr) - term_df.df + 0.5)/(term_df.df + 0.5)
      ) * qterms.tf * (1.2 + 1) / (qterms.tf + 1.2 * (1 - 0.75 + 0.75 *
      ((qterms."len")/((SELECT anr FROM len_avg)))))) AS subscore FROM
      qterms
18    JOIN term_df ON qterms.tid = term_df.tid)
19 /* suming up document scores and order by score descending */
20 SELECT subscores.did, subscores.vid, sum(subscores.subscore) AS rnk
21 FROM subscores GROUP BY subscores.did, subscores.vid ORDER BY rnk
      desc LIMIT 50;
```

5.5 VCSM5 Single Table

In analytical business processing database schemes are often denormalized[KR02] to improve performance of warehousing tools. The schema from 5.5 carries this concept to the extremes. In fact only a single table remains if the tid can be used for the query directly. As all attributes are transferred to the terms table the expenses for join operations are saved. In return the tuples belonging to the document relations are attached to every record of the term's relation. The storage demand to document related tuples increases approximately by a factor of the average document length. The rest of the table stays unchanged compared to schema 5.4.

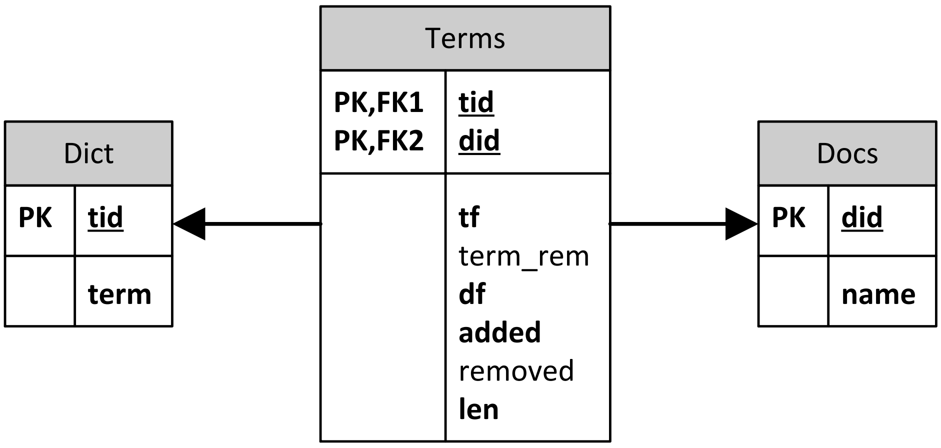


Figure 5.5: Single Table DB Schema (VCSM5)

5.5.1 BM25 Query

As the documents table is effectively excluded from the schema, all operations are conducted on the terms table. Thus, the filtering of qualified documents is deferred to line five. Further, the validity period is checked on the terms table directly in line three and only after the qualified docs are ready, non-relevant terms in line seven are filtered out. Succeeding calculations are analogous to the previous models.

Listing 5.5: BM25 Retrieval on the Single Table Model

```
1 WITH
2 /* filter valid terms containing one of the search strings */
```

```
qterms AS (SELECT terms4.tid, terms4.did, terms4.vid, terms4.tf,
    terms4."len" FROM terms4 WHERE added <= timest AND (removed IS
    NULL OR removed > timest)),
/* derive unique groups of documents */
qdocs AS (SELECT distinct did, vid, len FROM qterms),
/* filtering all documents containing the values */
filt_terms AS (SELECT qterms.tid, qterms.did, qterms.vid, qterms.tf,
    qterms."len" FROM qterms JOIN dict ON qterms.tid = dict.tid WHERE
    dict.term IN ('%term1%', '%term1%', ..., %termn%)),
/* compute document frequency */
term_df AS (SELECT tid, count(*) AS df FROM filt_terms GROUP BY tid),
/* compute average document length (avg(len)), and nr of documents in
     collection N */
param AS (SELECT avg(len) AS anr, count(*) as tnr FROM qdocs),
/* compute document term scores */
subscores AS (
SELECT filt_terms.did, filt_terms.vid , filt_terms."len", filt_terms.
    tid, term_df.df, filt_terms.tf, (SELECT tnr FROM param) AS n, (
    SELECT anr FROM param) as av,
(log(((SELECT tnr FROM param) - term_df.df + 0.5)/(term_df.df + 0.5))
     * filt_terms.tf * (1.2 + 1) / (filt_terms.tf + 1.2 * (1 - 0.75 +
    0.75 * ((filt_terms."len")/((SELECT anr FROM param)))))) AS
    subscore
FROM filt_terms JOIN term_df on term_df.tid = filt_terms.tid)
/* suming up document scores */
SELECT subscores.did, subscores.vid, sum(subscores.subscore) AS rnk
FROM subscores GROUP BY subscores.did, subscores.vid ORDER BY rnk
    desc LIMIT 50;
```

5.6 VCSM6 Versions IDs

The model represents a shift in paradigm to the suggested prototype for data citation in [PR13b]. The information about the range of validity is encoded in the vid column along with the documents instead of timestamps in the other models. Depending on the kind of implementation the vid value is incremented for updates, deletions and insertions right after query executions. The idea is to describe data states with the vid attributes unambiguously. The versions column maps non-overlapping time periods to corresponding vids. As such all revisions belonging to a document can easily be retrieved with the static dids. There is no additional logic in place to determine new dids, if documents change. Simple increments on the revision ID (revid) are sufficing. This 2-fold document revision is aligned with the Mediawiki (cf. table 7.1) way of storing documents. Apparently with the column we also introduced a new join attribute which is suspected to slow down the join operation between the documents and terms table. All records affected by document deletion are explicitly tagged with a deleted flag. Such tagged documents are excluded from the result set during candidate selection. For any other document only the latest vid is considered for retrieval rankings. The final schema is presented in figure 5.6.

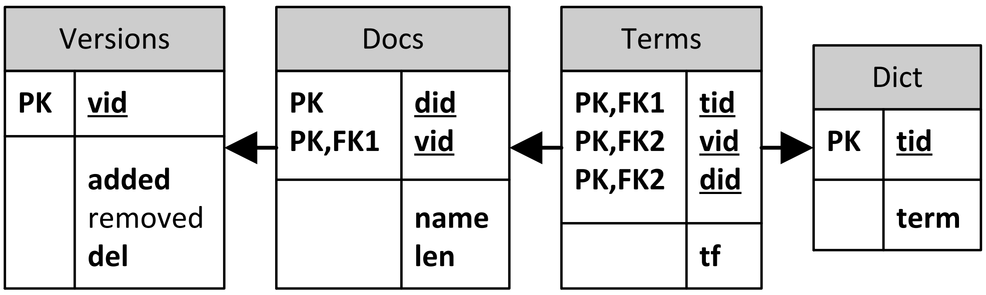


Figure 5.6: Version IDs DB Schema (VCSM6)

5.6.1 BM25 Query for VCSM6

At first the query spots the relevant vid at query execution time in the versions table in line three. In line five to eight the docs table is filtered by the latest document versions. If the latest version conforms to the *"deleted state"*, the document is excluded from the selection. The remaining portion of the query is similar to VCSM2 (cf.section 5.2). The only difference is the expansion of the PK to did and vid.

Listing 5.6: BM25 Retrieval on the Versions IDs Model

```
1 WITH
2 /*valid document terms */
3 qvers AS (SELECT vid, del from versions WHERE added <= timest AND (
     removed IS NULL OR removed > timest)),
4 /* selecting the latest document versions of documents, excluding
     removed documents */
5 qdocs AS (SELECT b.did, b.vid , b.len, b.name FROM
6 (SELECT did, max(vid) as vid FROM docs WHERE vid <= (SELECT vid from
     qvers) GROUP BY did) as ldocs
```

```sql
JOIN docs b ON ldocs.did = b.did AND ldocs.vid = b.vid
JOIN versions ON versions.vid = ldocs.vid WHERE versions.del = false),

/* selection of qualified terms */
qterms AS (SELECT terms.tid, terms.did, terms.vid, tdic.term, terms.tf FROM
(SELECT tid, term FROM dict WHERE term IN (%term1%, %term2%, %term3%, ..., %termx%)) AS tdic
JOIN terms ON terms.tid = tdic.tid
JOIN qdocs ON qdocs.did = terms.did AND qdocs.vid = terms.vid ),

/*extracting collection statistics from document list*/
stats AS (SELECT avg(len) AS anr, count(*) AS tnr FROM qdocs),

/* computing number of documents containing search term (df) */
term_df AS (SELECT tid, count(tid) AS df from qterms GROUP BY tid),

/* computing document term sub scores */
subscores AS (
SELECT qdocs.did, qdocs.vid , qdocs."len", qterms.tid, term_df.df, qterms.tf, (SELECT tnr FROM stats) AS n, (SELECT anr FROM stats) as av,
(log(((SELECT tnr FROM stats) + 0.0)/(term_df.df)) * qterms.tf * (1.2 + 1) / (qterms.tf + 1.2 * (1 - 0.75 + 0.75 * ((qdocs."len")/((SELECT anr FROM stats)))))) AS subscore
FROM qterms
JOIN qdocs ON qterms.did=qdocs.did AND qterms.vid = qdocs.vid
JOIN term_df ON term_df.tid = qterms.tid)

/* sum up document scores and order by score descending */
SELECT subscores.did, subscores.vid, sum(subscores.subscore) AS rnk
FROM subscores GROUP BY subscores.did, subscores.vid ORDER BY rnk desc LIMIT 50;
```

5.7 Limitations

One must take into account that the last approach is the only model that has the original did preserved and solely the expansion of the did by the vid attributes makes the document distinguishable from earlier versions of the same document. The other models require unique did values and as a consequence at another point the knowledge about the relation between the did development (respectively updates) needs to be maintained. This is an important finding considering the following implications. When a document with a certain didis added to the database where another document with the same ID already exists, an update is expected in case the document is different from the existing. Therefore, with respect to the *unique* constraint another ID has to be assigned to the document for model one to five. But taking no additional arrangements, the link between the different document versions is about to be lost. Somehow actions need to be taken to sustain links in both directions so that holding the new ID allows for finding the previous document states when otherwise the document entries cannot be found or updated in the database. The discussion of proper solutions on how to circumvent this issue efficiently goes beyond the scope of this thesis. Anyhow a practical approach is presented in the efficiency evaluation in section 7.

5.8 Conclusion

In this chapter we introduced six database schemes allowing retrieval results to be reproducible, even when the data changes dynamically. The models are aligned to retrieval data models presented in chapter 4.2 enriched with temporal data following the data citation approaches in chapter 3. The simplest models are VCSM1 and VCSM2. They are equivalent to the IR prototyping approach but instead of reading the df value from the dictionary table it is computed dynamically. The only difference between VCSM1 and VCSM2 is that the term's position within the document is preserved in the first model and replaced by the tf in the second one. In addition to the model one and two, VCSM3 keeps the history of df values in the dictionary table and has therefore to introduce a dictionary history table capturing all data states to be reproduced. Model four and five shift attributes from the dictionary and documents table to the terms table to reduce additional joins costs. And finally model six swaps all temporal information to a dedicated versions table which keeps track of the system state by defining vid. In place of timestamps it acts as revision ids in the documents table. The BM25 retrieval is adapted to all data models and we notice the query gets more complex the higher the number of the model is. As the first two data models are the simplest, we expect them to be easiest to maintain and to perform best on updates in the data collection. Query latency is supposed to be shorter in VCSM3 because the df value is selected from the dictionary history table and must not be computed on the fly. VCSM4 and VCSM5 are more expensive to maintain cause huge storage overhead and require rather complex queries. Their reason for existence is entirely based on the hope for achieving faster query execution times when join operations become no longer necessary. And last, VCSM6 offers a higher flexibility to manage data states and the overhead introduced by the versions table is rather low.

CHAPTER 6

Effectivity Evaluation

After the theoretical models have been introduced in the previous section a small example will demonstrate their functionalities. In the first place the schemes are filled step by step with small test samples. After each DB operation the results are presented and checked against expected results. Once the models stood up to scrutiny, in the second part the experiments are extended to medium-size data samples to assess the correctness of retrieval results on all schemes including the basic model. Equal result rankings must be achieved for all variants. Finally, the experiments are enlarged and the performance of the system is tested in a variety of different setups.

6.1 Introductory Example

The effectiveness of each of the six models is evaluated according to the following steps.

1. Insert two short documents.
    - Document D_1 *"Alan Turing"* at T_1.
    - Document D_2 *"Aileen Kay"* at T_1.
2. Execute Query Q_1 {'Alan', 'Mathison', 'Turing'} at T_2.
    - Compare retrieval result scores with expected results.
3. Add third document.
    - Document D_3 *"Alen Mycroft, Alan Turing"* at T_3.

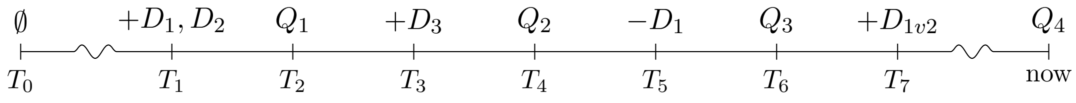


Figure 6.1: Experiment Timeline

- Reconstruct and reevaluate the query Q_1 with corresponding timestamp T_2, the results must coincide with last execution.

4. Launch new query Q_2 {'Alan', 'Mathison', 'Turing'} encompassing D_3 at T_3.
5. Remove document D_1.
   - Rerun query Q_2.
6. Launch new query Q_3 {'Alan', 'Mathison', 'Turing'} at T_6. D_1 must be excluded from the ranking.
7. Re-add document D_1 with alterations and new id at T_5 to simulate an update.
   - Document D_{1v2}- *"Alan Mathison Turing"*.
   - Reconstruct results from query Q_1, Q_2 and check if Q_3 has changed.
8. Run query Q_4 and check final scores on the entire collection.

The sequence of actions is demonstrated in figure 6.1.
In step two and three the general effectivity of the retrieval model is checked. The scores computed by the system are compared to the expectations. In step the system's behavior on updates is assessed. Retrieval results prior to the insertion of document 3 must be restorable. Non-time-stamped query scores must comprise all three documents. Similar to step five in step seven the system's behavior on deletions is reviewed. Step eight checks for system's capability to take account of document updates. Furthermore, the evaluation pays particular attention to the boundary values at time T_1, T_3 and T_5. The computed query scores fall into four different time intervals at $R_a \rightarrow [T_0, T_1)$, $R_b \rightarrow [T_1, T_3)$, $R_c \rightarrow [T_3, T_5)$ and $R_d \rightarrow [T_5, now]$ and the score must correspond for every point in time within the interval.

6.1.1 VCSM1

After the tables have been created with the scripts provided in the appendix B.2, two documents with time-stamp T_1 are created and added to the database (cf. appendix

B.3). The following simplified BM25 scoring function is used to rank the documents: $Score(D,Q) = \sum_{t \in Q} \ln(\frac{N}{d_f}) \frac{t_f(k_1+1)}{t_f+k_1(1-b+b\frac{len}{avg(len)}})$ - where D is the respective document, Q the query, t a query term, tf the term frequency, df the document frequency, $k_1 = 0.75$ and $b = 0.75$ free parameters, len the length of the document and $avg(len)$ the average document length with regard to the entire collection. To prevent negative scores, for this small sample, the IDF function is cut short to $\ln \frac{N}{d_f}$. The adjusted ranking function from section 5.1.1 produces the sub-scores in table 6.1 for the query terms *'Alan', 'Mathison', 'Turing'*.

did		len	tid	d_f	t_f	N	avg(len)	subscore
100	2	1	1	1	2	2	0.69314718055994529	
100	2	2	1	1	2	2	0.69314718055994529	

Table 6.1: BM25 Retrieval, Query Terms *{'Alan', 'Mathison', Turing'}* at Time T2

Every row in the sub scores represents a match of a query term in the document. Term one and two are both present in document 100. Both terms are only contained in the collection once and in the first document only. Therefore, the score for both matches are equal and sum up to the cumulative score.

$$\begin{gathered} Score(D,Q) = \sum_{t \in Q} \ln(\frac{2}{1}) \frac{1(1.2+1)}{1+1.2(1-0.75+0.75\frac{2}{2}}) \\ \rightarrow 0.693147 + 0,693147 = 1.386294 \end{gathered} \tag{6.1}$$

Next D_3 is inserted at time T_3 (cf. appendix B.8) and the query re-executed at time T_2 and T_3. The intermediate results for T_3 are stated in table 6.2. Document D_1 is still ranked first with 0.90 followed by D_3 with 0.82. The timestamped query at T_2 produces the expected ranking from table 6.1.

did		len	tid	d_f	t_f	N	avg(len)	subscore
100	2	1	2	1	2	$2.6\overline{6}$	0.45165733561415777	
100	2	2	2	1	3	$2.6\overline{6}$	0.45165733561415777	
300	4	1	2	2	3	$2.6\overline{6}$	0.48877985634956811	
300	4	2	2	1	3	$2.6\overline{6}$	0.33661254258036294	

Table 6.2: The Sub-Scores from BM25 Retrieval with the Query Terms *'Alan', 'Mathison', Turing')* on the Table with Three Documents at Time T_3

Apparently the system is able to handle insertions according to plan in this small sample. In the next phase the document D_1 is deleted by setting the removed timestamp in the docs table to T_5. The ranking score for D_3 goes up to 1.48 and the document D_1 vanishes. The result rankings produced by Query Q_2 have not changed. In the last step now, the document D_{1v2} is added at the same time D_1 was deleted in order to enforce an update of the document. Since the updated document contains all the query terms exclusively, the ranking increases. The intermediate sub-score results are depicted in table 6.3 and in table 6.4 an overview of the final data records is given.

did	len	tid	d_f	t_f	N	avg(len)	subscore
300	2	1	2	2	3	3	0.50972756447883527
300	2	2	2	1	3	3	0.35680929513518472
101	3	1	2	1	3	3	0.40546510810816438
101	3	6	1	1	3	3	1.0986122886681098
101	3	2	2	1	3	3	0.40546510810816438

Table 6.3: The Sub-Scores from BM25 Retrieval with the Query Terms *'Alan', 'Mathison', Turing')* on the Final Table

Apparently D_{1v2} is now in front with non-timestamped queries. According to our calculations the results are correct and further do query Q_2 and Q_3 still produce consistent results. The first model is encouraging and further examination will be undertaken on larger data samples.

6.1.2 VCSM2

Model two varies only little from the previous model and is discussed in short by comparing the retrieval results. When executing the scripts provided in appendix B.3, the queries Q_1 to Q_4 produce the same result scores. The terms table 6.5 illustrates the only difference to the previous model. As opposed to VCSM1 the term 1 which is contained in document D_3 twice, is written only once and the other record is dropped. The quantity is declared in the term frequency column tf.

6.1.3 VCSM3

For VCSM3 the insert scripts provided for VCSM2 are re-used. In addition, updates are reflected in the dictionary history table. Each distinct term of newly inserted, deleted or updated documents (or document batches) triggers a new history record with the term's df value deduced by the delta to the previous state. Exemplary scripts for this experiment are provided in B.4. Like with inserts, document deletion of D_1 requires not only to set the *removed* timestamp to invalidate the document. In addition, all terms

Table 6.4: VCSM1: Status of the Index after all DML-Statements

(a) Docs Table

#	did	name	len	added	removed
1	100	Doc 1	2	T_1	T_5
2	200	Doc 2	2	T_1	NULL
3	300	Doc 3	4	T_3	NULL
4	100	Doc 1	2	T_5	NULL

(b) Terms Table

#	tid	did	pos
1	1	1	1
2	2	1	2
3	3	2	1
4	4	2	2
5	1	3	1
6	5	3	2
7	1	3	3
8	2	3	4
9	1	1	1
10	6	1	2
11	2	1	3

(c) Dict Table

#	tid	term
1	1	Alan
2	2	Turing
3	5	Aileen
4	3	Kay
5	4	Mycroft
6	5	Mathison

Table 6.5: VCSM2: Status of the Final Terms Table

#	tid	did	tf
1	1	1	1
2	2	1	1
3	3	2	1
4	4	2	1
5	1	3	**2**
6	5	3	1
7	2	3	4
8	1	1	1
9	6	1	2
10	2	1	3

of the deleted candidates are read from the terms table to extract the term ID's of the terms affected by the document deletion. The corresponding df values are read from the history table, are decreased by one and written back. A similar procedure is performed on document updates. In addition to the document deletion, a comparison and merging of the different documents is required since document changes would impact the dictionary history table. Table 6.6 shows the final table content of the experiments on model three. The *NULL* values mark the latest entries in the table. Periods are non-overlapping so each tid has exactly one df value assigned for each point in time a query is run.

The model produces the same result rankings for query $Q1$-Q_4 just like the previous models.

6.1.4 VCSM4

To store the df value in the terms table, operations similar to VCSM3 are necessary. Here the two attributes *added* and *removed* account for the df in the query time period. Updates and insert functionality can be achieved by adding new documents and updating the df values of the previous periods. However, deletions become more complicated. The query in appendix B.5.6 factors in a *del* flag to the documents table which marks documents that are deleted but no new version of the document is added (document updates). In case of deletions only the terms table can now contain two occurrences of the same term (tid and did) belonging to the same document due to the fact df values have to be updated anyway. So terms tuples replications is enforced for document deletions too. The document table holds information about the temporal validity. In case of deletions it must be marked as deleted in addition to setting the *removed* timestamp. The query needs to distinguish between the two applicable candidates at query time. Documents already deleted at query time, are excluded from the retrieval results. Values from three different columns are required to distinguish the df in the terms table. These are the actual added and removed timestamps contained in the documents table and the *rem_df* column of the terms table. Due to the size of the terms table and rapid growth, tuple reconstruction and subsequently query processing is potentially expensive. Table 6.7 shows the records in the terms table after document one has been deleted.

After inserting document D_3 the df of tid one and two are updated in line five and seven. The previous values are invalidated by setting the "removed flag", *rem_df*, to the insertion timestamp in line one and two. As there are no records for update in case of document removal, in line eight and nine dummy records are added only to decrease the df values. These records are filtered out during query processing. Yet the update of document 1 can be handled like ingest. The table 6.8 shows the final state after all documents have been updated. The dictionary does not change in this scenario either. When applying the scripts from appendix B.5 the same retrieval results as in the previous

Table 6.6: VCSM3: Status of the Index after all DML-Statements

(a) Docs Table

#	did	name	len	added	removed
1	100	Doc 1	2	T1	T2
2	200	Doc 2	2	T1	NULL
3	300	Doc 3	4	T2	NULL
4	100	Doc 1	2	T2	NULL

(b) Terms Table

#	tid	did	tf
1	1	1	1
2	2	1	1
3	3	2	1
4	4	2	1
5	1	3	2
6	5	3	1
7	2	3	1
8	1	1	1
9	6	1	1
10	2	1	1

(c) Dict Table

#	tid	term
1	1	Alan
2	2	Turing
3	3	Aileen
4	4	Kay
5	5	Mycroft
6	6	Mathison

(d) Dict_Hist Table

#	tid	added	removed	df
1	1	T1	T2	1
2	2	T1	T2	1
3	3	T1	null	1
4	4	T1	null	1
5	1	T2	T2	2
6	2	T2	T2	2
7	5	T2	null	1
8	1	T2	null	2
9	2	T2	null	2
10	6	T2	null	1

models are achieved.

Table 6.7: VCSM4: Status after Deletion of Document 1 at T_5

#	tid	did	tf	rem_df	df
1	1	100	1	T2	1
2	2	100	1	T2	1
3	3	200	1	NULL	1
4	4	200	1	NULL	1
5	1	300	2	T2	2
6	5	300	1	NULL	1
7	2	300	1	T2	2
8	1	100	2	NULL	1
9	2	100	1	NULL	1

Table 6.8: VCSM4: Status of the Index after all DML-Statements

(a) Docs Table

#	did	name	len	added	removed	del
1	100	Doc 100	2	T1	T2	false
2	200	Doc 200	2	T1	NULL	false
3	300	Doc 300	4	T2	NULL	false
4	100	Doc 101	2	T2	NULL	false

(b) Terms Table

#	tid	did	tf	rem_df	df
1	1	100	1	T2	1
2	2	100	1	T2	1
3	3	200	1	null	1
4	4	200	1	null	1
5	1	300	2	T2	2
6	5	300	1	null	1
7	2	300	1	T2	2
8	1	101	1	null	2
9	6	101	1	null	1
10	2	101	1	null	2

6.1.5 VCSM5

A huge advantage of the “one table” approach presented next, is the possibility to omit join operations. The only task of the documents and terms table is to maintain the names of the entities but no joins are required to determine the query scores if the system is aware of the dids and tids. To maintain the df value for deletions, dummy entries are added for all terms in the document. A range-scan on the terms tables is performed to select the documents and ranking weights. In the first place the time validity restrictions on all tuples are checked. This subset including the dummy records serves as input to df selection according to the validity period. Like with the previous models a single and unambiguous record shall identify the df value at any point in time. The grouping of dids in the terms table opens access to a intermediate document list - formerly supplied by the documents table - comprising all document attributes, but the name. Self-joins of the downscaled views on the terms table wipe out the dummy lines and fetches in scoring weights. Table 6.9 marks the final filling of the table after the deletion of D_3. Two dummy lines represent the new df status as opposed to the update performed in table 6.10 where the latest document frequency is stored along with the actual terms of the new document D_{1v2}.

Table 6.9: VCSM5: Status after Deletion of D_1

#	tid	did	added	removed	tf	rem_df	df
1	1	100	T1	T2	1	T2	1
2	2	100	T1	null	1	T2	1
3	3	200	T1	null	1	NULL	1
4	4	200	T2	null	1	NULL	1
5	1	300	T2	null	2	T2	2
6	5	300	T2	null	1	NULL	1
7	2	300	T2	null	1	T2	2
8	1	0	T2	null	0	NULL	1
9	2	0	T2	null	0	NULL	1

Table 6.10: VCSM5: Status after Update of Document D_1 with D_{1v2}

#	tid	did	added	removed	tf	rem_df	df
1	1	100	T1	T2	1	T2	1
2	2	100	T1	null	1	T2	1
3	3	200	T1	null	1	NULL	1
4	4	200	T2	null	1	NULL	1
5	1	300	T2	null	2	T2	2
6	5	300	T2	null	1	NULL	1
7	2	300	T2	null	1	T2	2
8	1	101	T2	null	1	NULL	2
9	6	101	T2	null	1	NULL	1
10	2	101	T2	null	1	NULL	2

6.1.6 VCSM6

A special characteristic of the model is the implementation of the vid increments. The model can be as fine grained as the previous models if any change in the collection is reflected in the vid increment. With emphasis on query reconstruction only, document ingest does not affect the vid value as long as no new query has run against the system in the meantime. In the present case $vid = 1$ covers the period $[T_0, T_3)$, $vid = 2$ the interval $[T_3, T_5)$ and $vid = 3$ the period $[T_5, now]$. Document updates or deletions are accompanied by vid incrementation compulsorily. After executing the insert scripts B.7 the table fillings of the docs, versions and terms tables is now as shown in table 6.11.

Table 6.11: VCSM6: Status of the Index (Terms, Docs and Versions) after all DML-Statements

(a) Docs Table

#	did	vid	name	len
1	100	1	Doc 1	2
2	200	1	Doc 2	2
3	300	2	Doc 3	4
4	100	3	Doc 1	2

(b) Terms Table

#	tid	did	vid	tf
1	1	100	1	1
2	2	100	1	1
3	3	200	1	1
4	4	200	1	1
5	1	300	2	2
6	5	300	2	1
7	2	300	2	1
8	1	100	3	1
9	6	100	3	1
10	2	100	3	1

(c) Versions Table

#	vid	added	removed	del
1	1	T1	T2	1
2	2	T2	T2	1
3	3	T2	null	1

The query provided in the appendix B.7.6 achieves the same scoring values for all terms as the previous models. Just like with the extended models two to five, for VCSM1 further amendments to the model are conceivable but are not discussed here. The reason for this is by using the *vid* column to identify records instead of timestamps in the data we lose flexibility to access data. Lets for instance discuss the realization of VCSM3 (cf. Section 6.1.3) making use of the *vid* concept from VCSM6. Instead of having *added*

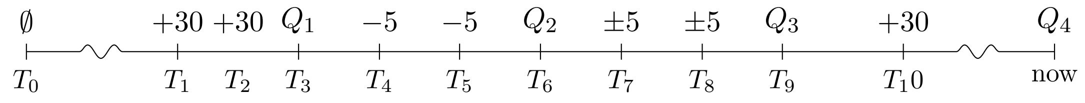


Figure 6.2: Proof of Concept Evaluation Timeline

and *removed* timestamps to identify the df value in the dictionary history table, only the vid column would be introduced. But then, as we have to select the *max(vid)* to identify the latest valid df value from the dictionary history table, an additional expensive grouping operation is required causing additional performance drawbacks. Hence, further observations of such approaches is not carried on.

6.2 Proof of Concept

In the last section the underlying theory was elaborated with a small example and a few inserts. In this section we step further and use generated data to assess the correctness and efficiency of the retrieval results more trustworthily. The sequences of actions are chosen to cover a wide variety of possible combinations of transitions. The case of multiple consecutive inserts, updates and document deletions is covered as well as other critical transitions. Timeline 6.2 demonstrates the instruction sequences for the experiment. $+$ denotes document inserts, $-$ deletions and $\pm$ updates. In total 90 documents are added, 10 deleted and 10 updated, which sums up to a total of 100 individual documents.

6.2.1 Data Set

The Vocabulary V consists of $|V| = 100$ individual random latin terms. The population of terms is expanded to 685 terms so that 50 appear exactly ten times, 25 five times, 15 three times, six twice and three only once. Out of the pool of 685 terms, 100 documents are created by randomly and uniformly picking 50-150 terms with the help of the Benerator Test Data Generator [1] The scripts are added in appendix A.1 whereas the particular documents source files can be found online [1]. The documents have distinguishable names assigned and are stored as single plain text files in alphabetic order. All terms are written in separate lines. For the retrieval queries we have used terms from three different frequency classes with respect to the entire collection, *'cursus'* (df=41), *'augue'* (df=25) and *'viverrra'* (df=14).

[1]Databene Benerator `http://databene.org/databene-benerator`

[1]Experiment files: `https://www.dropbox.com/sh/pw7t8e9vmwpa44e/AADK6XX4xMXTvETt9mJhUQWja?dl=0`

6.2.2 Experiment Methodology

The experiment is designed to assess if the introduced data models

- are fit for data updates. Are suitable insert, update and delete strategies available and how do they look like?
- achieve retrieval result rankings conforming to the BM25 algorithm.
- can reproduce past query results on data dynamics.
- are exchangeable amongst themselves.

6.2.3 CRUD operations for VCSM1

Initially, the first 60 documents are flushed to schema one (cf. section5.1) in two batches, both having distinguishable transaction timestamps [Sno00, pp.12-18](as opposed to valid time) T_1 and T_2 assigned, to improve traceability. When this is done, the query Q_1 is executed and the results stored for later comparing. The same way further deletes, updates and queries are run alternately. In the end document scores listed in figure 6.12 are achieved consistently and reproducibly for query Q_1 to Q_4 running the query from section 5.1.1. The experiment timeline is depicted in 6.2. In order to prepare the data models the files are parsed (cf. parser in section 7.2) and stored in the different models (section5.1.1). The parser reads and stores the 100 files sequentially. The tids are assigned by the RDBMS on inserts in the dictionary table. By all means new document inserts do not affect existing records in the database and the newly parsed documents can simply be appended to the end of the tables. Nonetheless, the system hast to cope with updates and deletion requests. In VCSM1 deleting documents results in setting the *removed* stamp in the documents table only for the affected tuple. Updates require setting identical timestamps for removal of the old did and addition of the new document. All scripts used for the setup are quoted in the appendix B of this work.

As expected the scores for the different queries vary. Document '53000' returns the highest score in the time interval observed. Generally the computed scores for the other documents in the results on Q_2 diminish compared to Q_1. Due to the document deletion, the power of the collection is reduced whereas the df values for the query term remains unchanged (cf. table 6.13). This causes the IDF ratio $\frac{N-d_f+0.5}{d_f+0.5}$ to decrease.

Table 6.12: Document Scores for Query Terms 'augue', 'cursus' and 'viverra'

	(a) Q_1		(b) Q_2		(c) Q_3		(d) Q_4	
rank	did	score	id	score	id	score	id	score
1	53000	2.81784	53000	2.36619	53000	2.49475	53000	2.97579
- 2	21000	2.50779	21000	2.17149	21000	2.40499	45001	2.61828
3	45000	2.19949	45000	1.96270	45001	2.23243	21000	2.58172
4	23000	2.02789	23000	1.80884	49000	2.02383	86000	1.94763
5	49000	1.92718	49000	1.77643	25001	1.76684	25001	1.92892

Table 6.13: Tf-idf Parameters for Retrieval

(a) Q_1

var	measures	Q_1	Q_2	Q_3	Q_4
N	60	50	50	80	
avg(len)	97.4	96.7	98.2	100.1	
term	$d_{f_{Q_1}}$	$d_{f_{Q_2}}$	$d_{f_{Q_3}}$	$d_{f_{Q_4}}$	
augue (6)	19	18	17	23	
cursus (58)	28	23	21	38	
viverra (98)	7	7	7	10	

(b) term frequencies t_f

tid	did	53000	45001	21000
6		2	1	1
56		0	0	1
98		1	1	1

CRUD operations for VCSM2

There are several ways to transfer the previous model to VCSM2. If built from the scratch, the parser either needs to process and compute term frequencies before storing, or read and where appropriate update existing records. Alternatively VCSM1 can simply be transformed to VCSM2 by exporting an aggregate function on tids for all document ids in VCSM1 to VCSM.. As a consequence the number of records in the terms table decreases from 9908 to only 5634 terms whereas the query (cf. appendix B.3.6) produces the same rankings as in the previous models. As the terms table is not affected by updates and deletions the strategies from VCSM1 are taken over.

CRUD operations for VCSM3

In VCSM3 (cf. 5.3) the dictionary, terms and documents table are processed in the same manner. A new challenge is to keep the dictionary history table updated. Here all data updates are way more expensive. The previous models required to set timestamps in the rather small documents table only once per document. If all documents are processed sequentially and have differing timestamps assigned, each distinct term in the document requires to have an updated df record set in the history table. A naive approach where the df value of every term is individually read and updated, leads to escalating update times. Let's take for example a document with 1000 terms added to a database with 1000 documents. Therefore, a 1000 look-ups and single document insertions on a table with 10^6 tuples would be required. One solution proposing itself is, to collect all potential update terms in memory and extract the latest df values with a single query. The updated values can be handled by a single transaction that helps to improve insertion speed substantially. The following two solutions are encouraging.

1. Keep the entire dictionary and the latest df value in working memory of an external application. This way a simple update on the dictionary history table to invalidate the old records and a new insert with the updated document frequency values is enough.

2. Alternatively, database procedures could ensure the posterior update of the df values. All documents and terms are stored according to VCSM2 and a database procedure is run right after the last term insertion. So the updates and inserts into the history table are conducted by joining the document and terms table with the history table in order to receive and update the values collectedly.

The procedure can further be optimized, if documents are collected in batches first, and the database is jointly updated when the max batch size is reached. As such the expense for the dictionary history batch update is equal to single document updates while the table size decreases dramatically. The df is now stored for any distinct term in the batch instead of a per document basis. In case the entire dictionary is kept in memory the relative frequency is traced and written out. In the latter case a temporary table can store all dids for updates, inserts and deletions. The time the update procedure is executed is called transaction time.

The *update* table stores the dids that are subject to change. In the *odid* column the ID'S of document's to be deleted. The *ndid* column states values to be added. If both columns are set an document update is performed where the old value document is tagged as 'removed' and the new document as 'added'. The figure 6.3 and algorithm 6.1 shows an exemplary implementation of the second strategy. The view *uv* builds the bridge between the update table and the document table. Two aggregations on the terms table for both, all terms regarding to the *odid* and did are computed and brought together by a self-join, making the new df value available. This intermediate results are

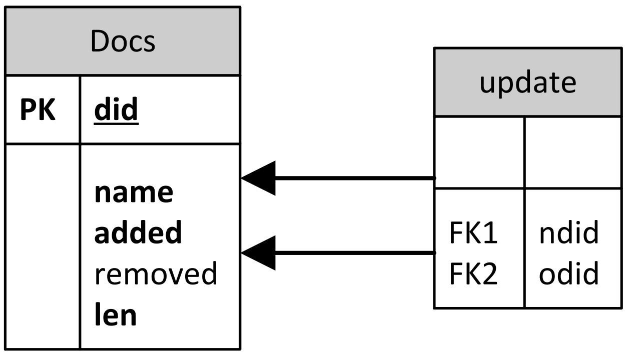


Figure 6.3: Exemplary Document Frequency SQL-Update Strategy for Document Batches

inserted into the dictionary history table using the transaction timestamp declared at the beginning. The same timestamp is used to invalidate the predecessors and for document addition in the documents table. In operation, the documents added to the system must not be considered in the retrieval query, as long as the update procedure has not been completed. Because of the long time span of update procedures, atomic transactions are unsuited. A workaround is to spare out the *added* timestamp during the initial document insertions. As such the documents do not qualify for any queries. As part of the update procedures, the *added* timestamp is then set for little additional expense at the end of the update procedure.

Listing 6.1: Exemplary SQL-Procedures to Update Records

```

CREATE view uv AS
SELECT COALESCE(udid_df.tid, ddid_df.tid) as tid, COALESCE(udf,0) -
    COALESCE(ddf,0) as ndf
from (select tid, count(*) as ddf FROM terms2 where terms2.did in (
    select updates.odid from updates) group by terms2.tid) as udid_df
full join
(select tid, count(*) as udf FROM terms2 where terms2.did in (select
    updates.ndid from updates) group by terms2.tid) as ddid_df
on udid_df.tid = ddid_df.tid;

CREATE PROCEDURE updateM2()
BEGIN
```

```
  declare ts TIMESTAMP;
  SET ts = current_timestamp;
  INSERT INTO tdict_hist select uv.tid, ts as added, null as removed,
     coalesce(df,0)+ coalesce(ndf,0) as df from tdict_hist RIGHT JOIN
      uv on tdict_hist.tid = uv.tid where removed is null;
  UPDATE tdict_hist SET removed = ts where tdict_hist.tid IN (select
     tid from uv) and added < ts and removed is null;
  UPDATE tdocs set removed = ts where removed is null AND did in (
     select odid from updates);
  UPDATE tdocs set added = ts where did in (select ndid from updates);
  DELETE FROM updates;
END;
```

For both strategies the same retrieval results like in the previous models are produced. The database solution may be easier to implement and requires less memory. Except for small batch sizes, having the application storing the df value is expected to outperform update times of the database only approach.

CRUD operations for VCSM4

In the VCSM4 the two strategies are translated to the new data models. The general requirements to external applications do not change. Still, the crucial part is to maintain the document frequency in memory to prevent frequent database searches and then write it to the terms table instead of the history table. The update triggered from the database involves only two tables now. As the terms table contains the df value, all aggregates are run on the same table with potential performance gain. This attribute sharing advantage can only be exploited for specific settings, where only single document inserts or updates occur. In case many updates are executed and the number of tuples in the terms table is fixed, the df column contains many redundant variables that have to be filtered during the query processing, or an arbitrary representative value is set qualified exclusively. Both implementations are neither clean or elegant nor do they bring any advantages over the approach in the third model. Furthermore, document deletions cause a duplication of storage space for the affected documents as all terms need to be reinserted to the same table for df updates only! The growth of the terms table *rem_df* column outweighs the advantages of having a single column by far, when being compared to the two columns tuple identification in the history table where only actual changes are stated. And last, df value updates are way more expensive than with the history approach. Instead of simply inserting the latest df values, existing term records are updated. To wipe out this advantage update terms are collected in the terms table - such as for VCSM2 - and the update procedure appends it to the latest column. This way documents do not appear in the result sets before the procedure is completed and update time can be improved. The procedure from section 6.1 populates the tables from the model which is capable of constructing the expected results with a slightly altered query. For the reasons stated above, this approach will not be considered a serious IR aspirant for a setting in data dynamics where collective updates occur or where documents are likely to be removed or updated.

CRUD operations for VCSM5

For the simple reason that VCSM5 is a further extension to VCSM4 the previous arguments become even more important. The two more timestamp columns cause additional tuple overhead for every insertion. For instance, if a single document is deleted, $|D| = len$ additional records are added to the terms table despite of redundancies of term frequency, removed timestamps or length attributes columns for terms not even contained by the document affected by the deletion. For this reason and without further demonstration it is concluded this model is impractical for both, batch insertions and frequent updates. *The models 4 & 5 will be evaluated in the light of incremental collection growth by document insertions exclusively.*

CRUD operations for VCSM6

In **VCSM6** a new perspective is opened. It is the first model to enable the preservation of the original document id for document updates by expanding the primary key with the version attribute. So document updates spare additional inserts for the deletion of the predecessors into the documents relation. Further the model is closely related to the second approach, and so are the SQL statements. Of particular interest is the management of the vid. The question at issue is, if solely the retrieval queries shall be reproducible or all data updates should be traceable as well. In the first case two consecutive operations do not necessarily initiate an increment of the vid attribute. According to the decision matrix in Figure 6.4 new vids are generated with respect to the preceding operation. For instance, queries always refer to the last state of the data whereas DML statements followed by a query yields an increment of the vids attribute.

A simple query store where a history of all SQL statements is preserved as proposed by Pröll [PR13b, p2] could do the trick. Thus, according to the last operation new vid

$$\begin{array}{c|cccc} & Q & I & U & D \\ Q & 0 & 1 & 1 & 1 \\ I & 0 & 0 & 1 & 1 \\ U & 0 & 0 & 1 & 1 \\ D & 0 & 1 & 1 & 0 \end{array}$$

Figure 6.4: Decision Matrix for *vid* Increments on Queries(Q), Inserts(I), Updates(U) and Deletions(D)

can be assigned when necessary.
The implementation of VCSM6 produces exactly the same results as all the other models and does therefore fulfill the requirement.

6.3 Conclusion

In this chapter we demonstrated the model's ability to exactly reproduce probabilistic retrieval result rankings, though documents were updated, deleted or new documents added. We achieved reproducibility of ranked retrieval results by reusing the timestamp of issuing the original query, given that all updates on the database preserve the edit history.

CHAPTER 7

Efficiency Evaluation

As the concept has passed all theoretical examination, the real significance of the findings are derived from the actual practical performance of the system in real world scenarios. In order to operate as an IR system it must be capable of dealing with large scale data and provide reasonable response times. Furthermore, the required hardware resources should be kept within acceptable limits. For evaluating system performance the retrieval database must meet practical requirements, i.e. proof that it can cope with high volume data. Therefore, the evaluation in this thesis is focused on retrieval on the entire English online encyclopedia *Wikipedia*.

7.1 The Wikipedia Experiment

There existed more than 5 million english wikipedia articles in August 2016 [1], 800 documents are added daily and each article is edited 90 times per month on average. This highly volatile data environment suits the performance benchmark requirements. It is the dynamically changing document corpus that allows assessing not only the response time on launched queries but also the impact of document updates, ingests or deletions on performance and storage growth.
The goal is to store all articles in the presented database models and work out, whether IR experiments can be run on top of them, and how they compare to each other. If the system can handle this large number of records it might be applied to other applications as well.

[1]Mediawiki table description: `https://stats.wikimedia.org/EN/TablesWikipediaEN.htm`

7.2 Test Data

One way to get the entire Wikipedia data is through database dump files (static XML files) downloadable from the Wikimedia page and mirrors[2]. In Mai 2016, first, the English dump files (enwiki) including only the latest article pages had a size of 12594780 KBs. After downloading the compressed '.bz2' files, a way had to be found how to extract the article data from the structured text files, transform the document stream and insert it into the database. This task turned out to be non-trivial since we had to omit traditional existing approaches for the reasons discussed in section 7.4 in favour of a parsing and ingest benchmarking tool developed for this purpose - the *Wiki2DB*[1].

In the particular case of Wikipedia article pages, some article pages do not contain exploitable data and are filtered out from processing.
Such articles are

- *Redirect Articles*:[1] Label article pages providing a link to another synonymous term. The article *'Binary Synchronous Transmission'* for instance, consists of a link to the page *'Binary Synchronous Communications'*. This prototype cannot make use of complex linking structure and the page itself does not contain valuable information and is thus skipped from parsing.

- *Stub Articles*:[2] These are articles tagged as not adequately explaining the matter or lacking information. This mostly applies to short articles. This sort of articles are excluded from processing as well.

- *Disambiguation Pages*:[3] Denotes articles solely aiming at resolving conflicts on ambiguous term definitions. Taking for instance the term *'Bavarian'*. It refers to not only the people of ancestry in Bavaria but also a village in Iran, Fars Province. This kind of pages start with a general term definition statement and a list of references pointing at contesting term definitions. This is very convenient for browsing through pages, but may bias our retrieval results. The pages are usually short and contain all key terms which may yield a high rating score owed to length normalization mechanism in the *bm25 formula*. In order to limit index size and avert disambiguation pages to be ranked first in all probability the pages are ignored by the parser.

[2]Wikipedia DB Dump Files: `https://dumps.wikimedia.org/`
[1]Parsing util Wiki2DB: `https://github.com/Merthyra/WikitoDB`
[1]mediawiki redirect `https://www.mediawiki.org/wiki/Help:Redirects/en`
[2]mediawiki stub `https://en.wikipedia.org/wiki/Wikipedia:Stub`
[3]mediawiki disambiguation `https://en.wikipedia.org/wiki/Wikipedia:Disambiguation`

- All kind of *Special Documents*: Documents not complying with regular encyclopedian style. These can be pages about other articles, article collection pages, aso.

7.2.1 Parsing Tool

The tool is implemented in Java programming language because of the interoperability and rich library support for parsing and text processing tasks.
The requirements are:

- Parsing XML-structured data.
- Efficient transformation of the document stream into list of terms (=tokens). This step includes the removal of stop words as well as meta tags and stemming.
- Detecting and filtering relevant documents.
- Logging and performance monitoring functionalities.
- Enable following-up previous parsing runs. The parser must be aware of the existing data in the database before the parsing run starts or continues.
- Incorporating behavioral logic on updates. In particular the tool must detect if articles already exist and initiate a different document update routine otherwise.
- Strategies to preserve the document frequencies.
- Having a simple architecture and being performant. Single stream processing makes it easier to monitor the performance and track back the actions.

For the Wikipedia part a simple SAX-parser extension WikiXMLJ[3] is forked and adapted to the project requirements. It is configured to extract the *did, revid, document title, document timestamp and document text* from every page and creating a page object for further processing. With the help of Apache Lucene's Analyzer package[4] the document stream is transformed going through the following pipeline,

- *tokenized* using the Wikipedia tokenizer, having all meta tags and wikipedia language constructs removed.

[3]WikiXMLJ Java Library: `https://github.com/synhershko/wikixmlj`
[4]Apache Lucene Analyzer `https://lucene.apache.org/core/5_3_1/core/org/apache/lucene/analysis/Analyzer.html`

- *latin folded*, transformation of alphabetic, numeric and symbolic Unicode characters to the basic latin (127 ASCII) equivalents where this is not otherwise the case.
- *lower case* transformation of all characters.
- *elision* filter, the removal of elision characters from the token stream.
- *stop-word* removal, providing an advanced stop-word list.
- *length* filter, removing all tokens with one or zero, or exceeding 100 character length.
- *stemming* filter, reducing all documents to a word stem, according to section 2.2.1.

The resulting list terms are further collected and organized by batch objects and handed over to the respective DAO's to write it to the database. Each transaction is monitored by a performance proxy class handled by Spring AOP [4].
Many settings can be configured manually. Most importantly the data model and the number of documents written to the database simultaneously (= batch size) must be configured upfront in a properties file or by passing command line parameters.
The parser reads and keeps all dids in the main memory. Thus, it can decide more quickly and with no additional DB access whether documents are newly inserted or update operations are required. In addition, collections statistics and in particular the latest df value can be stored along with the dictionary terms. As opposed to common uninformed parsers, the project aims at limiting expensive DB read operations often also involving costly term aggregations to access these collection statistics parameters.

7.2.2 Parsing Strategy

Once all database files are stored locally, they are processed using the *Docs2DB*-Parser in alphabetical order. Two different snapshots of the *enwiki* dumps are processed in two independent runs. The first run fills the empty database with all insert operations and monitors the insert performance. The second run targets at evaluating update operations on the existing and previously filled database. Only documents are added having an earlier representation in the database and consequently sharing the same did while having a different revid. As a consequence the performance of the two different operational modes is evaluated and compared separately.

[4]Spring AOP: `http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html`

7.2.3 Data Format

The Mediawiki project defines used data types in Wikipedia pages such as the range of integer values applied to dids and revids. This project follows this structure in order to keep the space requirements as low as possible. The table 7.1 lists the per row storage requirements for all columns and data types in the project.

Table 7.1: Wikipedia Experiment Data Types

typ	**Wiki Specs**[2]	**MonetDB type**[3]
did	UNSIGNED INTEGER (10 digits)	SIGNED INTEGER (32BIT)
revid	UNSIGNED INTEGER (10 digits)	SIGNED INTEGER (32BIT)
vid	n.a.	SIGNED INTEGER (32BIT)
added	CHAR/BINARY(14)	TIMESTAMP (64BIT)
removed	n.a.	TIMESTAMP (64BIT)
term	n.a.	VARCHAR 100 ($\leq 10032\,BIT$)
name	n.a.	VARCHAR 100 ($\leq 10032\,BIT$)
len	UNSIGNED INTEGER (10 digits)	SIGNED INTEGER (32BIT)
pos	n.a.	SIGNED INTEGER (32BIT)
t_f	n.a.	SIGNED INTEGER (32BIT)
d_f	n.a.	SIGNED INTEGER (32BIT)
pos	n.a.	SIGNED INTEGER (32BIT)

7.2.4 Test Environment

All experiments are conducted on a virtual server system [1]. The test system is configured as listed in table 7.2. Most DML statements are run in the server environment directly whereas the queries are executed remotly using MonetDB's JDBC drivers and the SQL-Client *SQuirreL SQL* [4].

7.3 Baseline Comparison

To get a better understanding of how the IR performance of our timestamped appproach is to be classified we compared the VCSM1 (cf. section 5.1) performance to the Lucene [Fou12] index and the BL (cf. section 4.2). Figure 7.1 shows the comparison of the three IR approaches on the entire data collection of 3034603 million articles. All collection statistics such as the average document length, the total number of documents and the

[1]Hetzner VServer: `https://www.hetzner.de/de/hosting/produkte_vserver/cx60`
[4]SQuirreL SQL: `http://squirrel-sql.sourceforge.net/`

CPU-Type	Intel Xeon (Sandy Bridge)
Cores	8
CPU MHz	2100
Memory	32GB
OS	Ubuntu 16.04 (Xenial)
DB	MonetDB v11.23.3
Storage	500GB SSD
Filesystem	ext4

Table 7.2: Test System Configuration

tids are pre-computed and given. The queries in appendix C.2.1 are executed 50 times using the search terms 251-300 from the TrecWeb[5]. For the Lucene benchmarks all documents where stored within one folder on the filesystem containing the same terms as the database separated by new line characters. For indexing and retrieval a slightly adapted version of the Lucene demo[6] was used applying the BM25 similarity[7] scores. Lucene Index creation on the test system (cf. table 7.2) took 10021 seconds in total for all articles.

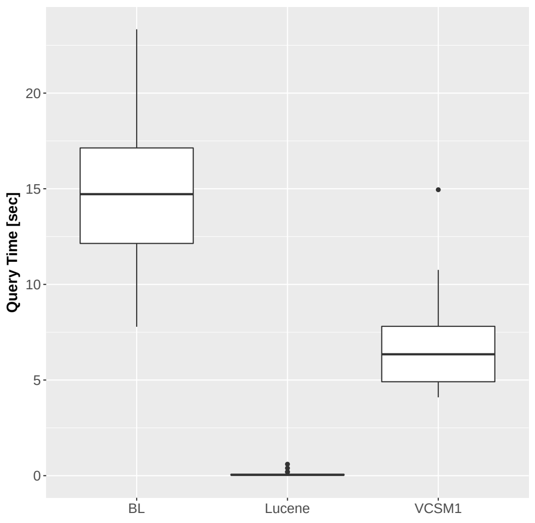


Figure 7.1: Comparing Query Response Times of Lucene Index with BL and the Revised Timestamped VCSM1 on the Wikipedia Dataset on 50 Consecutive Queries Returning the First 50 Documents with a Dictionary Size of 28.8 Million Terms

[5]Trec Web Test Queries: `http://trec.nist.gov/data/web/2014/web2014.topics.txt`

[6]Lucene Demo: `https://lucene.apache.org/core/2_9_4/demo.html`

[7]Lucene BM25 Similarity: `https://lucene.apache.org/core/4_0_0/core/org/apache/lucene/search/similarities/BM25Similarity.html`

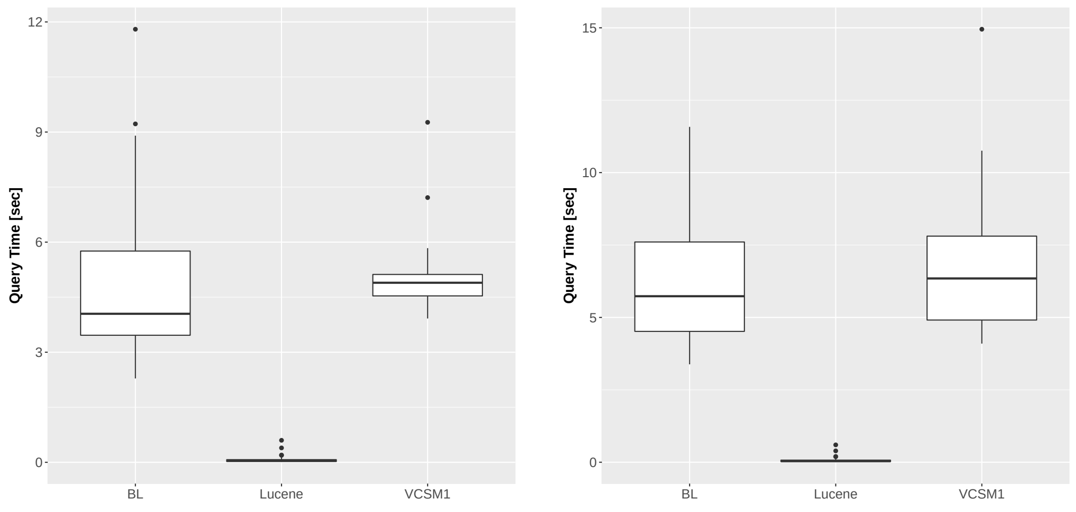
The results indicate Lucene is in front with average query response times of 200ms. While we were expecting this, we were surprised to find a huge gap in performance to the test setup BL used by Mühleisen [MSLdV14] who used the first disc of the ClueWeb2012 data set[8] with approximately 45 million documents. Rather than query latencies of one second and less our setup required 15 seconds though we used similar queries on a more powerful hardware with a smaller number of documents. Further differences in the setup listed next cannot be made accountable for the large discrepancies found.

- Our setup is not built on top of a compressed file system, but as similar studies suggest[SBH14] is the impact of compressed file systems rather small.
- We use disjunctive queries as opposed to the original setup but cross checking with the conjunctive approach showed no significant differences in the given setting.

Because we do use the same sample queries for our estimations, the only explanation left is the structure of the test collection. The average size of the documents between the collections does not differ significantly so we isolated the dictionary size - the actual cardinality of the vocabulary - to be the crucial factor. Figure 7.2b and 7.2a demonstrate the impact of the dictionary size on the query performance.
This is also an explanation why the BL approach performs worse than the VCSM1 approach. The average response time from the BL approach is 15 seconds whereas the timestamped solution does not even take one third of the time to compute the results. In this setting it seems less effort to compute the document frequency parameter out of the intermediate query terms table rather than looking it up in the dictionary table, which is the first unexpected finding. According to figure 7.2, storing df in the dictionary pays only off for smaller dictionaries. Defining a threshold on the vocabulary terms to exclude terms which are contained in more than 25% of all documents (low entropy), or vice versa, unique within the entire corpus, shrinks the vocabulary to fewer than 3 million documents. At this magnitude the BL outruns the VCSM1 approach on average response times. We also found the variance of response times in the VCSM1 to become much smaller with decreasing vocabulary size. This is because the search space on the tid column in the terms table becomes smaller as the number of allocated bins of equal tid is reduced.

On the bottom line this first experiment raises expectations for versionable indices being a useful amendment to the c-store retrieval index.

[8]ClueWeb12: `http://lemurproject.org/clueweb12/index.php`



(a) Dictionary Size 1 Million Terms (b) Dictionary Size 5 Million Terms

Figure 7.2: Query Latency Compared to Figure 7.1 having the Size of the Dictionary Table Reduced

7.4 Ingest Performance Evaluation

To assess the time required for building a retrieval index for the suggested models we measure the relative ingest speed with the parser introduced. Therefore, we had to figure out a realistic setup how data can be imported from the ".bz2" dump files into the database. Among many alternatives, the purported fastest way would be a bulk import of a text file into the database after the input file was translated to a machine-readable format (e.g. Map-Reduce jobs) by the databases' bulk importer. For the stipulated data schemes this is hard to achieve because we generate several tables from one file including interdependencies between the records of the tables. Further, new inserted data depends on data which has been persisted earlier. From this finding we concluded ingest can be achieved most efficient from an application to cache intermediate index statistics and prevent unnecessary database access. We have observed MonetDB is slow in constructing single tuples from database queries, thus whenever ingest involves read access prior to the actual insert statement the overall performance is in free fall and hence the overall approach highly impractical. With increasing size of the database the effect gets even worse and additionally, running in ingest mode yield performance gains from column-store dynamic indexing to vanish since dynamic indices have to be rebuilt each time data is updated. Let's take the simple VCSM1 for example. The documents and terms can easily be inserted without any further checks, provided document novelty is ensured beforehand. However, in the dictionary table we demand a single instance of a term with a given unique ID by the database and the assigned dictionary ID is required to be written to the terms table, substituting the actual term. If we requested the tid from the database for every document term, a thousand of read operations would be conducted causing a severe performance overhead. Another option with *unique* database constrains and triggers assigned to the term name do not work out either for the same reason of causing a huge performance impact.

Therefore, we equipped the parser with a dictionary and document buffer, simple in memory HashMaps, to reduce the discussed overhead on ingests. We assume any application using this database would have to rely on caching mechanisms to keep data update times within feasible limits.

As pointed out above, the overall data ingest performance of the introduced data models is predominantly subject to the applied database caching strategy of the ingestion-tools' persistence layer. However, discussions how to reduce database access with sophisticated caches are not part of this thesis and so there arises a challenge of not comparing apples with oranges on ingest performance benchmarks. Given the fact that ingest cannot efficiently be handled by the database alone and ingest logic optimization is put one layer up into a persistence algorithm, makes it hard to assess the performance of data model's contribution on data ingest performance isolated. Nonetheless, we give it a try.

It has proved the database performance on ingest changes dramatically once data records were deleted from the tables again. Consequently, on each DML statement database tables have to be rebuilt from the scratch to reproduce its' exact state for the tests. It is not enough to just delete recent changes before new experiments can be launched in order to be comparable. Because parsing and tokenization of the Wikimedia dump files and writing them to the database over JDBC is a very time consuming (cf. table 7.3) task we decided to first import all files into the database for VCSM1 and shorten the import procedure for the other models by leveraging the bulk processing power of the column store to copy and translate retrieval data to the remaining models. The applied scripts can be found in appendix C. This course of actions allows conducting query benchmarks on top of the full-blown Wikipedia database for all models. Otherwise, inserting the entire collection with the introduced parser to the suggested schema and employed hardware would probably have taken month for VCSM3, VCSM4 & VCSM5. For VCSM1 the parsing tool introduced in the last section ran more than 36 hours to tokenize the XML data and store the document data in the column store. The batch insert operations alone account for approximately 29.7 hours and is divided between the different tables as listed in the comparison view (cf. table 7.3). The ingestion performance of the other models are evaluated on a smaller subset of the data in section 7.4.1.

table	rows	ingest time [min]
terms	2 727 855 619	29h 37min
dict	28 588 030	23 min
docs	3 034 603	5 min

Table 7.3: Required Time to Ingest 3 Million Documents on VCSM1 Using a Batch Size of 500 Documents

Approximately 98% of the time the system is busy with building the terms table. This share is reduced linarly for VCSM2 where the terms table shrinks to approximately 1/3 of the original table size. Nevertheless, it remains the focus of performance considerations. While ingest on VCSM1, VCSM2 and VCSM3 takes almost constant time, ingest on VCSM3 takes much longer when the collection grows. The size of the dictionary history and the batch size are the main drivers influencing the ingestion times. The bigger the historized dictionary table becomes, the longer each ingest takes. When comparing table 7.4 and Figure 7.3 the average insert speed on VCSM3 in an empty table is about 12 docs/second and declines to five docs/second on the corpus filled with three million documents even though there is a overhead to populate the dictionary table involved when the index is empty.

7.4.1 Ingest Performance Comparison

The ingestion time measurements of the different approaches as listed in table 7.4 are so far apart that they are hardly comparable. At the beginning, when all tables are empty, it takes more time to build up the dictionary table for each insert, all new terms introduced are also subject to amendment in the dictionary table. With increasing maturity this overhead vanishes. On the downside, the larger the collection gets, the slower insert statements on huge tables become. Furthermore, the models VCSM3, VCSM4 and VCSM5 keep track of the df values, thus old validity periods have to be updated on each update of the document corpus. Apparently this impacts the ingestion times to a large extend as depicted in figure 7.4. The huge gap in ingest times on pre-filled tables between the VCSM1, VCSM2 and VCSM6 as opposed to VCSM3, VCSM4 and VCSM5 is entirely attributed to the need to update existing data in the table, i.e. to reset invalidation timestamps. Table 7.4 underlines this finding by showing, when the tables are rather small and hence fewer records are to be considered for invalidation, the picture changes completely and VCSM4 becomes even the best performing schema.
We have explored several strategies to most efficiently perform updates to all models. In

any case, to achieve acceptable latencies, we had to completely omit intermediate read operations as they had slowed down ingest performance in a way it became unusable. Hence, the dictionary term-tid mapping, did set, tid-tf mapping for the current document and global tid-df mapping values are all kept in memory to speed up inserts. For all models but VCSM1, VCSM2 and VCSM6 the timestsmaps marking the validity time-frame for the last df-values need to be updated additionally. Therefore, we introduced an intermediate table where those tids are persisted that are subject to the current update operation. So we were able to facilitate faster SQL batch update operations by replacing the two-fold read and succeeding update operation by a single statement.

BL	VCSM1	VCSM2	VCSM3	VCSM4	VCSM5	VCSM6
33.9	32.5	10.4	13.8	7.6	35.8	12.1

Table 7.4: Indexing Time for Wikipedia Corpus [min] (first 10,000 Documents, Batch Size = 100)

Not surprisingly ingest in the first approach takes three times longer than for the second scenario. Allowing multiple equal terms in one document yields an explosion of the terms table by the factor of 2.7. For VCSM3 the batch size is the crucial issue. If df updates can be accumulated in a document batch, multiple records in the dictionary history table can potentially be spared. Approach VCSM4 and VCSM5 do not profit from fewer actual columns, compared to the VCSM3 approach. Here the batch configuration gains only little performance benefits. The drawback of blowing up each term entry with additional timestamp attributes outnumbers the savings for the *removed* column.

Figure 7.3 shows the degree to which the ingestion performance depends on the batch sizes. When documents are inserted one by one, i.e. batch size is one, ingest speed is significantly worse in all models. Here VCSM2 performs best with 13 docs/second followed by VCSM6 with eight docs/second. These two approaches hold the lead and we were able to index 20 docs/second using a batch size of 1000 documents. On the other end those models with more columns or records in the terms table fall behind. In particular VCSM5 performs the worst on all different batch settings. The BL and VCSM1 have two write more term entries, thus is slower but ingest speed is stable among all batch sizes at around five docs/second. VCSM3 and VCSM4 benefit from larger batch sizes as they do yield fewer updates on the collection. Applying larger batch sizes facilitates acceptable performance of ten docs/second on these approaches.

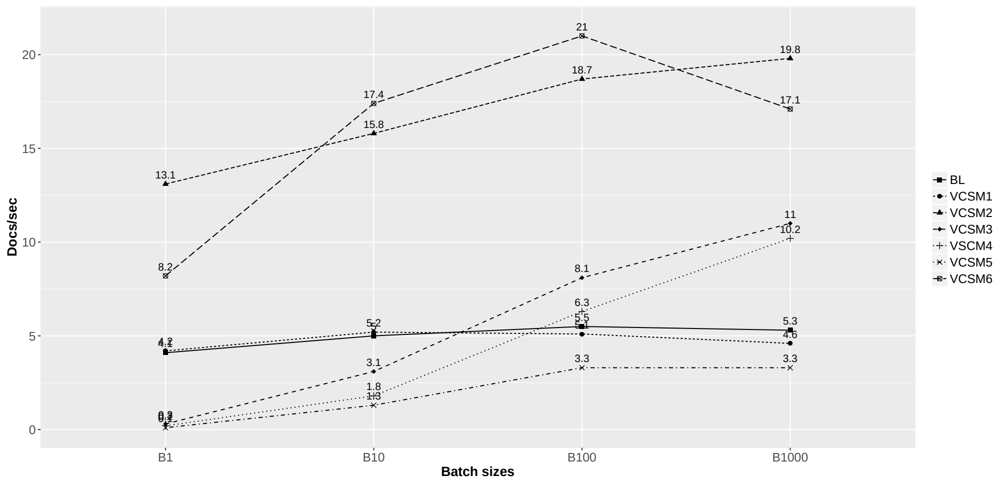


Figure 7.3: Ingest Speed for the Models on Top of a Corpus with 3 Million Documents as a Function of the Batch Size

7.4.2 Conclusion

Ingest performance to a large extend depends on whether only large bulks are appended to the existing data structures or whether intermediate updates and read operations are conducted. Thus, caching of intermediaries such as the current tid to term mapping plays a crucial part to keep ingest times within reasonable limits. It also means the ingestion strategy or algorithm accounts for the actual ingest performance, rather than just the simple DB schema. From the benchmarks it is self evident all approaches except for VCSM1, VCSM2 and VCSM6 are not well suited for data environment with frequent changes and updates. VCSM2 is the fastest and the simplest architecture at the same time. VCSM1 and VCSM6 are computationally more expensive but offer other opportunities discussed in the next sections. While there may exist better strategies to ingest, which may change the picture here, it is still to be expected that VCSM3, VCSM4 and VCSM5 will perform significantly worse because in any case there are more DML statements required.

7.5 Retrieval Performance

As the database is meant to cope with retrieval queries, the focus of performance evaluation is subsequently directed to query timings and VCSM1 to VCSM6 are benchmarked according to different parameters being suspected to impact the overall retrieval latency. Among all the discussed models VCSM1 is given a special status. It preserves the position

of all terms by allowing a term to occur repeatedly, which is crucial for certain retrieval models like language models. The number of records grows with a factor of 2.7 to the other models which does certainly affect query timings.
These selections of parameters for evaluation is derived from prior experience with the system or common sense. Like with the correctness evaluation, we also stick to the BM25 retrieval model (cf. section 2.3.1) in order to be comparable with Mühleisen's [MSLdV14] findings. Therefore, we have compared the retrieval of the BL with the versionable (timestamped) models.

table	rows
terms1	2 228,170 055
terms2	859 926 093
dict	2 350 829
docs	3 325 892
batches	6652

Table 7.5: Table Length According to the Dictionary Threshold After Having Removed the *tids* Contained in 25% of all Documents or Occurring Only Once in the Collection

7.5.1 Disjunctive Retrieval Timings involving Dictionary Lookups

In this first evaluation we determined the query performance on the 50 queries used for the baseline comparison in section 7.3 but using the term string inside the query forcing the database to look up the actual tid in the dictionary table first. The import batch size is set to 500, i.e. 500 documents do have the same timestamp. In total *3325892* documents and 6652 batches are stored in the database. In order to achieve a more realistic setup we defined a dictionary threshold (cf. 7.5). We purged all vocabulary which was contained only once and those terms contained in more than 25% of all documents. Thus, the size of the dictionary could be reduced to 2 350 829 entries. Evidently we examined the query ranking scores to be identical for all approaches as an additional validation. The first subquery calls the dictionary table to extract the tids of the search terms. This is tantamount to an constant time added to each of the approaches. Thereafter, the result set of documents is shrunk to those being valid at the given point in time. In all approaches from one to five this is a simple full table scan on the respective document table and therefore does have equal effort. In VCSM6 a preprocessing step is required to obtain the latest valid vid which is not of removal type and adds up here. Certainly the toughest nut to crack is joining the document candidates with the allocated terms. Here the VCSM1 has the biggest number of terms, almost double the size of the other models. Computing the collection's statistics - such as the total number of documents and the average length of documents - is similar in all approaches once the set of qualified documents has been extracted from the database. All the remaining gaps in processing times are now to be explained with the differences in the strategies how the actual document frequency is extracted.

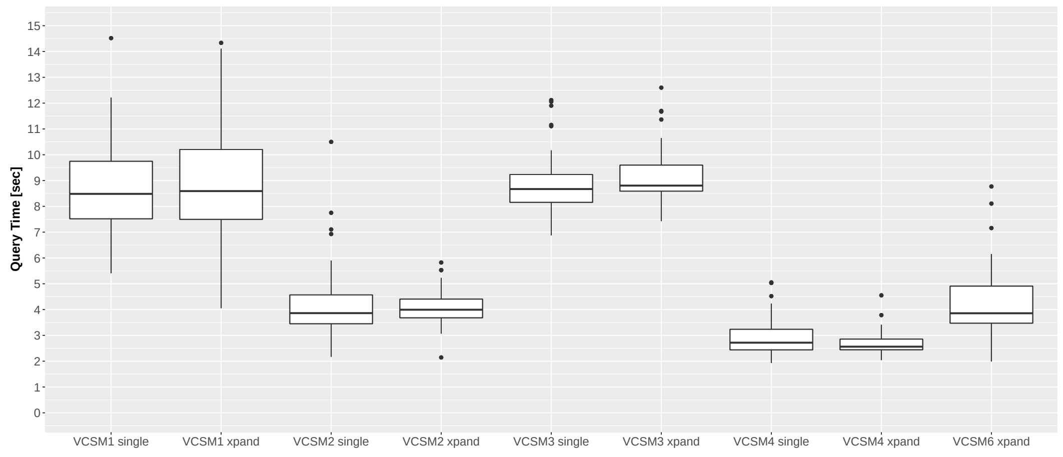


Figure 7.4: Hot-Run query response times [seconds], 50 queries from Trec-collection one with only a single column and the second with two columns

In figure 7.4 the average warm runs query response times are listed for all scenarios. MonetDB reorganizes and refines table indices from the knowledge it gains from query executions. Consecutive warm runs are usually faster by magnitudes. The applied dynamic column index refinement introduced in section 4.1 is the key driver behind the improvements in query response times.
The figure contrasts *single column ID's* with the concept of *extended document ID's*. In the later case we have expanded the dids primary key by a second revid column (xpand). It is a common requirement to keep ID numbers stable to keeping track of one document's history. Hence, we examined the impact on query time a supplementary revid column has in the given setting.
Due to response times beyond acceptable borders and limited spaces on the plot we had to remove VCSM5 from the graph. The average response time of VCSM5 was 7.2 minutes for the two column approach and 5.3 minutes for the single column approach. The calculation of the document statistics from the terms table turned out to be computationally expensive. We found VCSM2 and VCSM4 to perform best in the given setting. It takes only little additional time in VCSM6 to look up the version id in the dictionary table. Demanding a secondary column for the vid values is not a burden

when comparing to other models and the negligible differences between the one and two column ID approaches. The time required to extract the df value from the dictionary history table adds up to the time of VCSM2 in the VCSM3 benchmark. There we found another clear indication, the additional effort for storing the df along with the tid in a versioned way does not pay off. Another outcome of the query experiment is that schemes VCSM2, VCSM3 and VCSM4 hardly show any differences when an additional revid column is introduced. The column stores query execution pipeline factors in the join column only when necessary. So the little differences in warm run executions times are attributed to the 10% share of updated documents, which share the same did and can only be distinguished by the revid. Only warm run times on the VCSM1 diverge significantly. This is again owed to the fact that the two huge columns cannot entirely be kept in memory.

7.5.2 Disjunctive Retrieval Timings without Dictionary Lookups

Figure 7.5 shows the warm run benchmark assuming the term id is already known, i.e. does not need to be looked up in the dictionary table first. It turns out the dictionary lookup plays a big part in the overall query computation. The performance gain in all data models is significant. The huge performance boost in VCSM1 and VCSM2 suggests that the selection of strings on the comparably small dictionary table does indeed produce a performance drawback of two seconds. Interestingly the markup time is not constant for all approaches. The memory critical first setup does benefit even more from skipping the first lookup to the dictinary table. All the other setups remain the same and the query results do not scatter as indicated by the low standard errors. As long as the data is not beeing updated and thus, the BAT caches and indexes are invalidated, warm-run response times are remarkably stable.

7.5.3 The Impact of Batch Sizes

For the upcoming evaluations we have ceased the threshold on the dictionary table and reestablished the original dictionary table from 7.3. The batch size of inserts has been identified as one of the key parameters to influences the query performance. It is the number of documents that have the same ingest timestamp and can, from the column store perspective, be seen as the number of ordering relationships on the table. It potentially speeds up the selection on the documents and versions table but does directly have an impact on the number of records in the versioned dictionary table. The

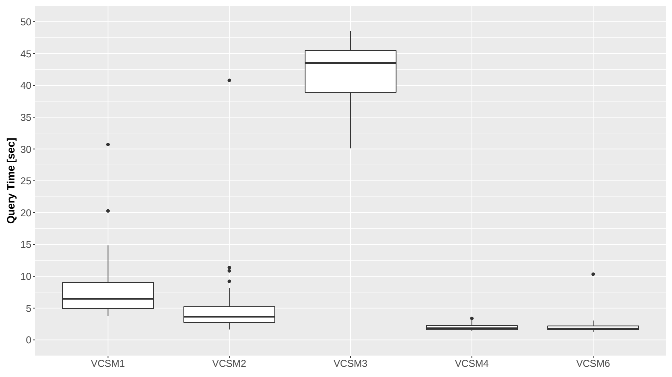


Figure 7.5: Average Query Execution Time Like Figure 7.4 but Without Additional Dictionary Lookup for the Term ID in the Query (tid is provided)

benchamark in figure 7.6 repeats the benchmark from section 7.5.1 but with different batch sizes 1, 500, 10000 and 100000. In order to achieve these different settings the insert scripts from appendix C were applied.
As expected the query response times of VCSM1, VCSM2 and VCSM6 do not vary that much. The table scan and table cracking on the relatively small versions or documents table do not vary at all, once the table is divided into the groups of matching and non-matching column-ID's. Only the performance on VCSM3 is addressed directly with the number of inserts for the document frequency in the historized dictionary table. The dependency of the query performance on the batch size is legible from the benchmark. Anyhow, even with a batch size of hundreds and thousands documents, the data model performs worse than the simpler VCSM2.

7.5.4 The Impact of Term Entropy on Retrieval Model Performance

The entropy in information science is often referred to the content of information. In this context, terms which occur more often than others, have lower entropy. A certain event has an entropy of zero. Applied to the discussed data model a unique term in the collection pins down one document exactly and contributes most to the ranking score. This is what is addressed with the *term frequency* and *document frequency* values in the retrieval models.
Thus, the elements with the highest entropy provide the best selectivity, i.e. the number of terms that match the select statement is small and so further processing should take place more quickly as for terms with little entropy. Figure 7.7 confirms this connection between term entropy and query response time in all data models. Again the query

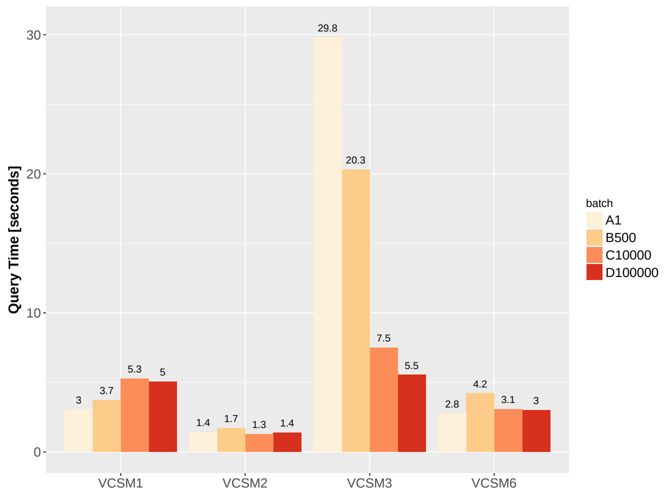
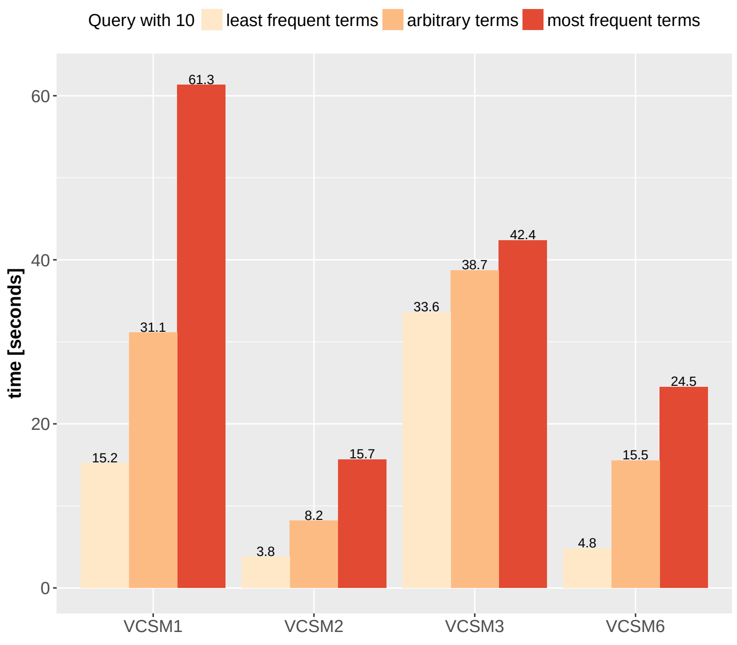


Figure 7.6: Average Query Execution Time on 3 Million Documents, 28.9 Million Dictionary Terms and Varying Batch Sizes

response time is measured for the different data models. This time the retrieval query is equipped with frequent search terms and infrequent ones in the collection. Each of the terms in the first column is only contained once in the entire dataset and therefore has the highest entropy possible. The other extreme is listed in the last column. Each of the terms listed there exists in almost 90% of all documents. The column in the center bridges the two extremes and is formed of arbitrary terms grouped to a meaningful query about Alan Turings' work on the enigma machine during the second world war.
Not surprisingly the first approach is most sensitive to the term entropy. The associative term table has the most candidates for join operations. But also the query execution is four times faster with rare terms. The time gaps in the third approach are similar to the second one and in VCSM6 the time gaps are slightly bigger because the more documents qualify for the retrieval, the more join candidates for the version table exist.
Finally, there is evidence all models are potentially capable of dealing with hundreds of high frequency terms. All suggested data models did respond to the retrieval query with 100 terms, even if it has taken minutes in the VCSM3.

On the bottom line of this first query evaluation the second approach is the fastest solution and consumes up the least storage. VCSM6 is significantly slower and demands slightly more space for the version table but on the other hand has the advantage of separating timestamp data with the actual document data. The evaluation draws a bad picture of VCSM3, the expected performance boost for the calculation of the df value, did not arise. The VCSM1 approach is the most flexible one and the response times are



Document Frequency			
		terms	df
low	t1	tehowarana	1
	t2	samm021	1
	t3	schiffsbot	1
	t4	schwieriger	1
	t5	siphonaldut	1
	t6	adlema001tim	1
	t7	cartujadeportaco	1
	t8	g2udd	1
	t9	goniatiten	1
	t10	rkqselo	1
average	t1	alan	115409
	t2	turing	142
	t3	enigma	5850
	t4	world	945362
	t5	war	548930
	t6	england	372372
	t7	work	1337766
	t8	mathemat	70459
	t9	codebreak	662
	t10	machin	123497
high	t1	refer	2651672
	t2	reflist	2217568
	t3	date	2201698
	t4	http	2165656
	t5	titl	2105422
	t6	www	1885701
	t7	year	1849923
	t8	link	1700721
	t9	cite	1694413
	t10	infobox	1649478

Figure 7.7: Impact of Term Entropy on Query Execution Time, on 3 Million Documents and 28.9 Million Dictionary Terms and a Batch Size of 500

still in reasonable limits. VCSM4 and VCSM5 did not scale satisfactoriry and turned out to be disproportionate burdensome.

7.5.5 Storage Consumption

The storage consumption as shown in figure 7.8 is easy to grasp by multiplying the values from the storage table 7.1 with the number of records from table 7.5. Denormalized terms tables in VCSM4 and VCSM5 have a big impact on storage demand. More than 40 GB are required to store the Wikipedia Enzyklopedia with VCSM5. The lowest storage is claimed by the second setup, that is around 12GB. The storage demand in the third and sixth setup depend on the actual batch size for ingests. The bigger the batch size is, the fewer new entries have to be added in the dictionary history or versions table for the entire collection in order to update the document frequency. This is of significance for VCSM3, where the number of entrances for each batch in the history table encompasses all distinct batch terms. If we assumed a batch size of one, the storage demand climbs up to the size of VCSM5 as for each reference of a dictionary item with a document a df

Table 7.6: Total Storage Consumption per Data Model [MiB]

Model	BL	VCSM1	VCSM2	VCSM3	VCSM4	VCSM5	VCSM6
Dict	847.3	838.3	838.3	5,969.6	838.3	838.3	838.3
Terms	25,499.4	25,499.4	9,841.1	9,841.1	19,682.1	36,083.9	13,121.4
Docs	130.7	177.0	177.0	177.0	177.0	119.1	142.3
Versions	0.0	0.0	0.0	0.0	0.0	0.0	0.1
total	26,477.4	26,514.7	10,856.4	15,987.6	20,697.5	37,041.4	14,102.2

value has to be persisted. Conversely, even taking the worst case, a batch size of one, in VCSM6 causing the version table to grow in lockstep with the documents table, the storage footprint of the versions table is still small. Each record accounts for 21 bytes only. This is different to the dictionary history, where for example 800 records with 24 bytes each, are persisted on each update. Hence, the bigger the batch sizes, the fewer updates have to be made to the history table, fewer storage is consumed and the query latency is reduced for data updates. Keeping in mind that VCSM6 has an additional column for the revid, the storage demand is slightly bigger. The size of the *versions* table alone is negligible and the maximum number of records is limited to the number of records in the documents table.

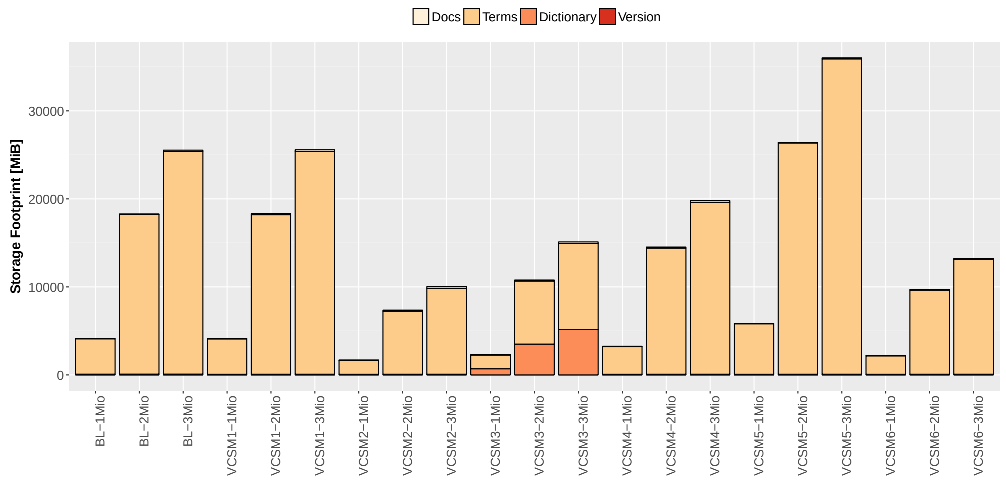


Figure 7.8: Model Storage Requirements with Batch Size 500 on Growing Corpus Employing Dictionary Thresholds (cf. Table 7.5)

7.5.6 Update Performance

Ingest and updating is supported and works with all proposed models. In order to evaluate the update performance the time required to remove documents from the collection is crucial. An update operation can be seen as a removal of the document to be deleted, followed by ingesting of an updated document. The time the removal of 10000 documents in 20 batches out of the Wikipedia collection of more than *three million* documents took, is displayed in figure 7.9. Each of the batch removal jobs is executed as one transaction. The logarithmic scale on the y-axis points out how far the VCSM3 lags behind the other solutions. This is because deletion comprises three different stages. The procedure used in appendix C.4 shows the different steps in more detail. As all the timestamps are also stored along the dictionary terms it is not enough to mark the documents as deleted in the documents table. Further, the deletion of documents need to be reflected also in the historized dictionary table. This step involves an aggregation of all terms in the affected documents to compute the difference of the document frequency. In the next step the old values have to be updated before the new df values are inserted having the corresponding time stamp. Each of these steps are expensive in query execution time whereas for the other approaches it is sufficient to mark elements as deleted in the relatively small documents table. Managing their versions in VCSM6 is slightly more complex because deletion requires to add a new version into the documents table.
Briefly recapitulated, VCSM1, VCSM2 and VCSM6 do achieve satisfactory performance on document updates. VCSM3 may not be a feasible option if the data collection is subject to frequent updates.

7.5.7 Tuning the query performance

So far the focus of the thesis was on improving the data models to be as efficient as possible. But there are are other aspects to database tuning we have not looked at yet.

Database Constraints

To enforce data integrity, database constraints are an integral part of RDBMS. For all data models introduced in the previous section primary- and foreign key constraints could (should) have been applied. Especially the tid and did column entries must be unique by definition. But applying the constraints onto the database columns decreased performance so much that experiments could not be carried out at all. Executing a query with unique and foreign key constraints on terms, document and dictionary table yield a termination of the server daemon immediately. Ingests into the huge terms table took

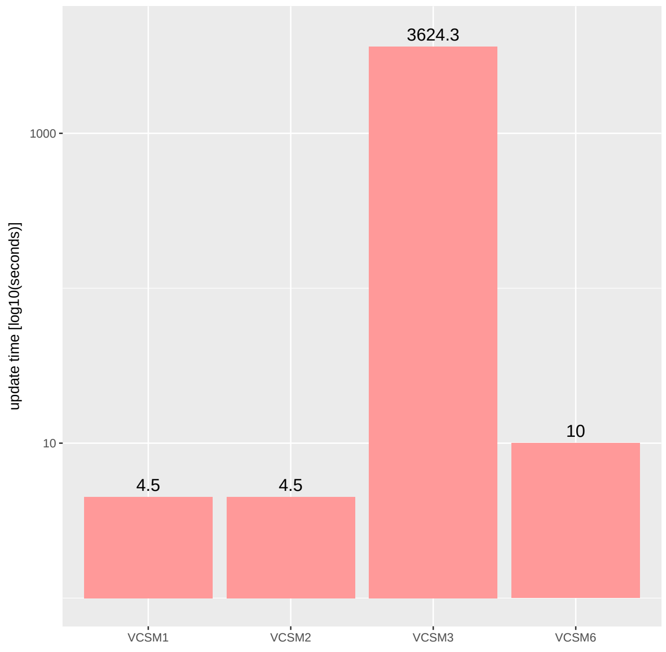


Figure 7.9: Time Required to Update 20 x 500 documents

forever, as the three fold constraint must be checked on each insert transaction. The impact of constraints on performance can not be evaluated in the thesis but as suggested in the next section, not even dedicated indices can improve performance significantly so there is only little, if any performance boost to be expected for queries. The performance of DML statements will drop when constraints are enforced on the schema.

Impact of Column Indices

Conventional row store based database use special data structures on columns to speed up data retrieval by improving direct access to selected columns avoiding full table scans. On the downside more disc space for the index and performance implications for index maintenance are traded in. MonetDB is organized differently and sequentially enhances knowledge about data hierarchy in table columns with every query processed. (cf. section 4.1). A dedicated index on the tid table did not improve select operations at the terms table in VCSM2. Only in the cold run case the response times improved slightly. This observation is in accordance to the fact, indices are treated as advice by MonetDB [9].

[9]MonetDB Indices: `https://www.monetdb.org/Documentation/Manuals/SQLreference/Indices`

It means, there is no enforcement of index structures on the columns and MonetDB employs own strategies on how to achieve the best performance for the provided tables.

Query execution pipeline

Beside the schma definition the query optimization plays an important part. MonetDB uses so called optimizer pipelines, these are chained operations, to transfer the query to MAL intermediate language and BAT algebra most efficiently. MonetDB offers several built in pipelines [10] that use different optimizer chains. Figure 7.10 compares the different queries with one another. None of the other optimizer pipelines performed significantly better on the VCSM2 approach than the default pipe which was used for all evaluation tasks in this chapter.

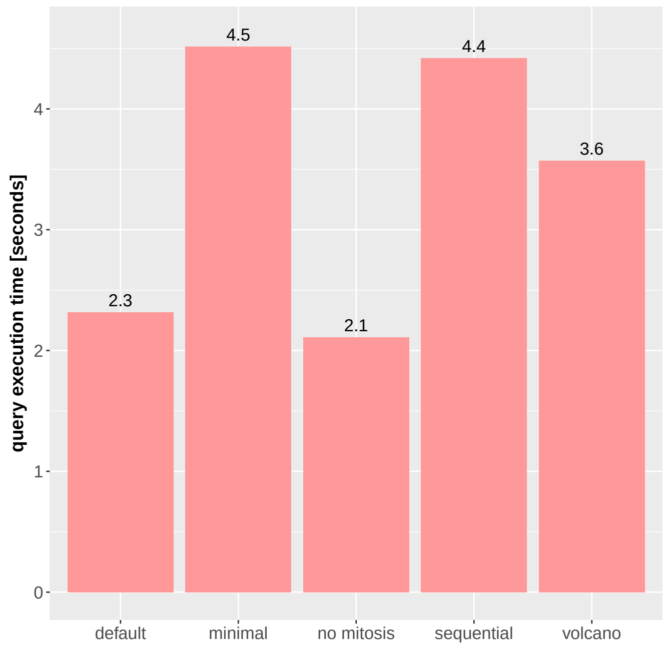


Figure 7.10: Optimizer Pipeline

7.6 Future Work

The performance evaluation is comprehensive in scope and illuminates most aspects of column stores performance. Yet there exist other facets, this thesis does not cover.

[10] MonetDB Optimizers: `https://www.monetdb.org/Documentation/Cookbooks/SQLrecipes/OptimizerPipelines`

For instance, some experiments during evaluation achieved better performance when the sequence of SQL operations was altered (manual query optimization). For instance, reducing the result set with select statements prior to the join operation performed better for some queries than simply joining tables and shrinking the result set with where clauses. More of the topics we could not work on in this thesis are listed below.

- MonetDB and other RDBMS can deal with distributed tables [11]. Beside known advantages of data replication, distributed databases could potentially improve performance when the load is not only concentrated on a single server system.
- MonetDB is a great database for research purposes and represents column-store well though on enterprise level other columns stores are more resilient and offer better performance when the database query demands more resources than available. Databases such as Actian Vector, a commercial spin-off from MonetDB, offer sophisticated query processors[BZN05] and perform better in benchmarks. [BZN05][ZB12].
- Parallel operations were not taken into account at all. All retrieval experiments were executed consecutively.
- An in depth analysis of query optimization pipes for this particular task might leverage existing opportunities for this special use case.
- Other retrieval algorithms than BM25 have not been part of this evaluation.
- On memory intensive operations the column-store MonetDB can become victim to the Out of Memory Killer resulting in inconsistent state of the database. This happened several times when huge number of records were updated. It manifests by showing multiple instances of the same table, not allowing the table to be deleted nor freeing its allocated space. As it is neither a good idea to disable the Out of Memory Killer in production environments one must observe the implications of the mix on scarce resources and DML statements on large tables when using MonetDB.
- Furthermore, deleted record from a table do not free allocated disc space. This might not only influence the storage footprint development but also the query performance, if, for example, legal obligations arise and data records need to be physically removed.

[11]Distributed Tables in MonetDB: `https://www.monetdb.org/Documentation/Cookbooks/SQLrecipes/DistributedQueryProcessing`

- The used integer data types for the document and tid are working at the edge in the Wikipedia setup. If long data types are required for the indices we expect to see direct impact on storage consumption as well as on query timings because main memory will become a more critical resource.

7.7 Conclusion

The column-stores are up to reproducible retrieval on huge dynamic data collections such as Wikipedia. All experiments were launched on a system comparable to commodity hardware and there is evidence the column-store retrieval scales with better hardware provided. The use of even more efficient column stores such as *Actian Vector* or distributed tables offer further opportunities to improve the performance measures derived in this thesis. Nevertheless, the hardware requirements and in particular the memory and storage requirements for retrieval tasks are not to be underestimated.
We compared the performance of the BL data model to VCSM1 and unexpectedly found the VCSM1 to be head-to-head for smaller dictionaries and outperforming the BL for large dictionaries. There exists a negotiable trade-off between the join operation to the dictionary table or an additional aggregation on the terms table in order to get the document frequency value. From a performance perspective for smaller dictionaries the dictionary approach is slightly in favor but from this we conclude, the dictionary table does not contribute to the overall retrieval performance in a way we would have expected.

Operation	**VCSM1**	**VCSM2**	**VCSM3**	**VCSM4**	**VCSM5**	**VCSM6**
Ingest Performance	medium	fast	medium	medium	slow	fast
Update Performance	fast	fast	slow	very slow	very slow	medium
Retrieval Performance	medium	fast	medium	fast	very slow	fast
Storage Requirement	high	low	medium	high	very high	low
Batch size dependency	little	little	very high	high	high	medium
Complexity	low	low	medium	high	high	medium

As depicted in table 7.7, from the 6 approaches introduced in chapter 5, only VCSM1, VCSM2 and VCSM6 remain as viable options concluding from the rather poor query performance (except VCSM4), the added complexity and increased storage requirements. Among the other options the approach VCSM6 is relatively fast but does have a little more complexity to maintain a separate table for data versioning. The performance is slightly worse than in VCSM2, which performed best in all benchmarks. It is not

only the simplest solution but also the most efficient one and is the obvious first choice unless the position of the term in the documents is to be considered which brings in the first approach, or there is good reason to delegate the versioning of the documents to the database system, where VCSM6 comes in handy. Nonetheless, update performance remains the key issue for performance considerations. Each time documents are added, the intermediate cached table results are updated and follow up requests are treated like cold runs. Hence, a data corpus that is frequently changed does perform significantly worse.
Given that all constraints ensuring data integrity are omitted in this setup, the ingestion and update performance on all other approaches except one, two or six is not sufficient.

CHAPTER 8

Summary & Outlook

Scientific research is one application where reproducible retrieval result rankings could support the matter of researchers, finding out which search result could be received at a particular point in time, i.e. comprehending what knowledge was available at a point in time. This application is referred to as reproducible text retrieval and cannot easily be achieved with conventional IR systems. Enabling retrieval systems to return reproducible result rankings by combining the IR prototyping approach on column-stores with data citation was the subject of this thesis.
Scalable data citation names the concept of using structured data, i.e. RDBMS, to tag data subsets unambiguously in order to make them versionable and distinguishable from circumjacent data.
But row stores are not performant for aggrdegating huge number of records expected for retrieval applications and thus another technology had to be found. Column-stores have demonstrated to be effective for IR prototyping. Retrieval models (algorithms) are translated to SQL and run on the database. This approach offers a clear separation of data and logic. The retrieval index and the important query statistics such as the *tf-idf* are built during query execution. Moreover, column-stores are designed for data warehousing tasks and have their strength in computing aggregations on columns, the most expensive operational step to obtain the document statistics. The question is if there are database models to achieve reproducibility and if column store performance can cope with respective retrieval tasks. Therefore, six models have been introduced each having different assets and drawbacks. VCSM1 and VCSM2 are straightforward and form the core of the other models. In most cases VCSM2 is the obvious choice the index

described in this model is the easiest and fastest to create and maintain, it achieves the best query response times in most evaluations and consumes the least storage. VCSM1 is the only model which preserves the position of the term within the document and is therefore the only choice for retrieval models where term proximity plays a part. There is no point in extending these simple models by a dictionary history table such as with VCSM3. We have shown in all settings and even with large batch sizes, the model cannot compete - be it query latency, storage consumption or ingest or index update speed. VCSM6 remains as viable option performing slightly worse than the simpler model but offering an interesting data model to separate the version numbering from document data in return. In specific settings where updates are rare and the vocabulary size is small, VCSM4 achieves remarkably low response times. The fully de-normalized VCSM5 approach performed worst in all categories and is not worth considering at all.

From the thesis we conclude that IR retrieval models can be used for reproducible queries efficiently. The system's response times are within reasonable limits even on commmodity hardware and huge data collections. The maximum number of processable data volume scales with the power of the hardware being used.

8.1 Outlook

The scope of the thesis is to evaluate data models for the IR prototyping column-stores to achieve repeatable searches in a dynamic data environment and to measure the response times and storage demand of such. But there are a few aspects not being addressed. The area of distributed computing has not been observed at all although column-stores support it. Further investigations would have to be done, how the system behaves on full load and concurrent use by multiple clients. MonetDB is a powerful column-store for research tasks but lacks resilience, performance and some features proprietary databases like Actian Vektor have. Furthermore, other retrieval models than BM25 could be evaluated to validate the finding of this thesis. Particularly models which use proximity measures can potentially update the viewing angle.
Recently, graph databases have emerged and gained importance. They are usually weak at aggregations on a entity base but utilize their strength at modeling relations between two entities. Modeled as graphs the associate terms table become obsolete and the database would be counting relations of dictionary items within documents which has encouraging performance settings. Furthermore, such relations are much more flexible to update. Native Graph Databases like Neo4j [1] are very robust and can deal with billions of

[1]GraphDB Neo4j:`https://neo4j.com/download/`

records. Further work should also regard graph databases in context of retrieval indices. There is a demand for reproducible IR in information research and the thesis suggests there is a solution to this problem, but it also offers many points of references to additional optimization.

APPENDIX A

Data Generator Scripts

A.1 Databene Data Generator Scripts

Given a plain text file with a single column and a number of records, the following script extracts a random selection of five to 15 terms.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<setup xmlns="http://databene.org/benerator/0.7.0" xmlns:xsi="http://
    www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://databene.org/benerator/0.7.0 http://
      databene.org/benerator-0.7.0.xsd"
  defaultEncoding="UTF-8" defaultDataset="US" defaultLocale="en_US"
  defaultLineSeparator="\r\n">
  <import defaults="true" />
  <execute>
    rand = new java.util.Random()
    </execute>
  <execute>
    count = rand.nextInt(10+1) + 5
    </execute>
  <bean id="austrian_file" class="org.databene.platform.fixedwidth.
      FixedWidthEntitySource">
    <property name="uri" value="austrian.flat.fcw" />
    <property name="entity" value="text" />
    <property name="columns" value="name[15]" />
  </bean>
  <generate type="transaction" count="{count}">
    <variable name="text" type="entity" source="austrian_file"
        distribution="random" />
    <attribute name="text" type="entity" script="text.name" />
    <consumer class="FixedWidthEntityExporter">
```

```
      <property name="uri" value="{'path/' + System.currentTimeMillis()
          + '.fcw'}" />
      <property name="columns" value="text[15]" />
    </consumer>
  </generate>
</setup>
```

APPENDIX B

SQL Setup Scripts

B.1 Timestamps

Listing B.1: Define Timestamps T_1-T_7

```sql
declare timest, T1, T2, T3, T4, T5, T6, T7 TIMESTAMP;
SET timest = current_timestamp;
SET T1 = '2015-10-01 12:00:00';
SET T2 = '2015-10-03 12:00:00';
SET T3 = '2015-10-05 12:00:00';
SET T4 = '2015-10-07 12:00:00';
SET T5 = '2015-10-09 12:00:00';
SET T6 = '2015-10-11 12:00:00';
SET T7 = T5;
```

B.2 Correctness Setup for VCSM1

B.2.1 Table Creations & Constraints

Listing B.2: SQL Create Table for VCSM1

```sql
 CREATE TABLE dict (tid INT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
     term VARCHAR(100) UNIQUE NOT NULL);
 CREATE TABLE docs (did INT PRIMARY KEY, "len" INTEGER NOT NULL,
    added TIMESTAMP DEFAULT CURRENT_TIMESTAMP(), removed TIMESTAMP,
    name VARCHAR(100) NOT NULL);
 CREATE TABLE terms (tid INT NOT NULL, did INT NOT NULL, pos INT NOT
    NULL);
```

B.2.2 SQL Insert Scripts D_1, D_2

Listing B.3: SQL Insert Scripts For VCSM1

```sql
INSERT INTO DICT (term) VALUES ('Alan'), ('Turing'), ('Aileen'), ('
    Kay');
INSERT INTO DOCS (did, name, len, added) VALUES (100, 'Doc 1', 2, T1),
     (200, 'Doc 2', 2, T1);
INSERT INTO TERMS (tid, did, pos) VALUES (1, 100, 1), (2, 100, 2), (3,
     200, 1), (4, 200, 2);
```

B.2.3 SQL Insert Scripts D_3

Listing B.4: SQL Insert Scripts for D_3

```sql
INSERT INTO DICT (term) VALUES ('Mygrott');
INSERT INTO DOCS (did, vid, name, len, added) VALUES (300, 1, 'Doc 3',
     4, T2);
INSERT INTO TERMS (tid, did, vid, pos) VALUES (1, 300, 1, 1), (5, 300,
     1, 2), (1, 300, 1, 3), (2, 300, 1, 4);
```

B.2.4 SQL Delete Scripts T_5

Listing B.5: SQL Delete DML for D_1

```sql
UPDATE docs set removed = T5 WHERE did = 100;
```

B.2.5 SQL Insert Scripts D_{1v2}

Listing B.6: SQL Insert Scripts for D_{1v2}

```sql
INSERT INTO DICT (term) VALUES ('Mathison');
INSERT INTO DOCS (did, name, len, added) VALUES (101, 'Doc 1', 3, T5);

INSERT INTO TERMS (tid, did, pos) VALUES (1, 101, 1), (6, 101, 2), (2,
     101, 3);
```

B.2.6 Query Sub Scores

```sql
WITH
/* filter valid documents and calculate avg(len) and N */
qdocs AS (SELECT * FROM docs WHERE added <= timest AND (removed IS
    NULL OR removed > timest)),
/* valid terms containing one of the search strings */
qterms AS (SELECT terms.tid, terms.did,tdic.term FROM
(SELECT tid, term FROM dict WHERE term IN ('Alan', 'Mathison', '
    Turing')) AS tdic
JOIN terms ON terms.tid = tdic.tid
JOIN qdocs ON qdocs.did = terms.did),
/* average document length (avg(len)) */
len_avg AS (SELECT avg(len) AS anr FROM qdocs),
/* total number of documents (N) */
doc_nr AS (SELECT count(*) AS tnr from qdocs),
/* frequency of terms in documents (tf) = term frequency */
```

```
term_tf AS (SELECT tid, did, COUNT(*) AS tf FROM qterms GROUP BY tid,
     did),
/* number of documents containing search term (df) */
term_df AS (SELECT tid, count(tid) AS df from term_tf GROUP BY tid)
/* document term scores */
SELECT qdocs.did, qdocs."len", term_tf.tid, term_df.df, term_tf.tf,
(SELECT tnr FROM doc_nr) AS n, (SELECT anr FROM len_avg) as av,
(log(((SELECT tnr FROM doc_nr)+0.0)/term_df.df) * term_tf.tf * (1.2 +
     1) / (term_tf.tf + 1.2 * (1 - 0.75 + 0.75 * ((qdocs."len")/((
    SELECT anr FROM len_avg)))))) AS subscore
FROM term_tf
JOIN qdocs ON term_tf.did=qdocs.did
JOIN term_df ON term_df.tid = term_tf.tid;
```

B.3 Correctness Setup for VCSM2

B.3.1 Table Creations & Constraints

Listing B.7: SQL Create Table for VCSM2

```
CREATE TABLE dict (tid INT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    term VARCHAR(100) UNIQUE NOT NULL);
CREATE TABLE docs (did INT PRIMARY KEY, "len" INTEGER NOT NULL, added
     TIMESTAMP DEFAULT CURRENT_TIMESTAMP(), removed TIMESTAMP, name
    VARCHAR(100) NOT NULL);
CREATE TABLE terms (tid INT NOT NULL, did INT NOT NULL, tf INT NOT
    NULL);
```

B.3.2 SQL Insert Scripts D_1, D_2

Listing B.8: SQL Insert Scripts for D_1 & D_2

```
INSERT INTO DICT (term) VALUES ('Alan'), ('Turing'), ('Aileen'), ('
    Kay');
INSERT INTO DOCS (did, name, len, added) VALUES (100, 'Doc 1', 2, T1),
     (200, 'Doc 2', 2, T1);
INSERT INTO TERMS (tid, did, tf) VALUES (1, 100, 1), (2, 100, 1), (3,
     200, 1), (4, 200, 1);
```

B.3.3 SQL Insert Scripts D_3

Listing B.9: SQL Insert Scripts for D_3

```
INSERT INTO DICT (term) VALUES ('Mygrott');
INSERT INTO DOCS (did, name, len, added) VALUES (300,'Doc 3', 4, T2);
INSERT INTO TERMS (tid, did, tf) VALUES (1, 300, 2), (5, 300, 1), (2,
     300, 1);
```

B.3.4 SQL Delete Scripts T_5

Listing B.10: SQL Delete DML for D_1

```
UPDATE docs set removed = T5 WHERE did = 100;
```

B.3.5 SQL Insert Scripts D_{1v2}

Listing B.11: SQL Insert Scripts For D_{1v2}

```
INSERT INTO DICT (term) VALUES ('Mathison');
INSERT INTO DOCS (did, name, len, added) VALUES (101, 'Doc 1', 3, T5);

INSERT INTO TERMS (tid, did, tf) VALUES (1, 101, 1), (6, 101, 1), (2,
    101, 1);
```

B.3.6 Query Sub Scores

```
WITH
/* filter valid documents and calculate avg(len) and N */
qdocs AS (SELECT * FROM docs WHERE added <= timest AND (removed IS
   NULL OR removed > timest)),
/* valid terms containing one of the search strings */
qterms AS (SELECT terms.tid, terms.did, terms.tf,tdic.term FROM
(SELECT tid, term FROM dict WHERE term IN ('Alan', 'Mathison', '
   Turing')) AS tdic
JOIN terms ON terms.tid = tdic.tid
JOIN qdocs ON qdocs.did = terms.did),
/* average document length (avg(len)) */
len_avg AS (SELECT avg(len) AS anr FROM qdocs),
/* total number of documents (N) */
doc_nr AS (SELECT count(*) AS tnr from qdocs),
/* number of documents containing search term (df) */
term_df AS (SELECT tid, count(tid) AS df from qterms GROUP BY tid)
/* document term scores */
SELECT qdocs.did, qdocs."len", qterms.tid, term_df.df, qterms.tf,
(SELECT tnr FROM doc_nr) AS n, (SELECT anr FROM len_avg) as av,
(log(((SELECT tnr FROM doc_nr)+0.0)/term_df.df) * qterms.tf * (1.2 +
   1) / (qterms.tf + 1.2 * (1 - 0.75 + 0.75 * ((qdocs."len")/((SELECT
    anr FROM len_avg)))))) AS subscore
FROM qterms
JOIN qdocs ON qterms.did=qdocs.did
JOIN term_df ON term_df.tid = qterms.tid;
```

B.4 Correctness Setup for VCSM3

B.4.1 Table Creations & Constraints

Listing B.12: SQL Create Table for VCSM3

```
CREATE TABLE dict (tid INT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
   term VARCHAR(100) UNIQUE NOT NULL);
```

```sql
CREATE TABLE docs (did INT PRIMARY KEY, "len" INTEGER NOT NULL, added
     TIMESTAMP DEFAULT CURRENT_TIMESTAMP(), removed TIMESTAMP, name
    VARCHAR(100) NOT NULL);
CREATE TABLE terms (tid INT NOT NULL, did INT NOT NULL, tf INT NOT
    NULL);
CREATE TABLE dict_hist (tid INT NOT NULL, added TIMESTAMP NOT NULL,
    removed TIMESTAMP, df INT NOT NULL);
```

B.4.2 SQL Insert Scripts D_1, D_2

Listing B.13: SQL Insert Scripts for D_1 & D_2

```sql
INSERT INTO DICT (term) VALUES ('Alan'), ('Turing'), ('Aileen'), ('
    Kay');
INSERT INTO DICT_HIST (tid, added, df) VALUES (1, T1, 1), (2, T1, 1),
     (3, T1, 1), (4, T1, 1);
INSERT INTO DOCS (did, name, len, added) VALUES (100, 'Doc 1', 2, T1),
     (200, 'Doc 2', 2, T1);
INSERT INTO TERMS (tid, did, tf) VALUES (1, 100, 1), (2, 100, 1), (3,
     200, 1), (4, 200, 1);
```

B.4.3 SQL Insert Scripts D_3

Listing B.14: SQL Insert Scripts for D_3

```sql
INSERT INTO DICT (term) VALUES ('Mygrott');
UPDATE DICT_HIST SET removed = T3 WHERE tid IN (1,2,5) AND (removed
    IS NULL AND added<T3);
INSERT INTO DICT_HIST (tid, added, df) VALUES (1, T3, 2), (2, T3, 2),
     (5, T3, 1);
INSERT INTO DOCS (did, name, len, added) VALUES (300,'Doc 3', 4, T3);
INSERT INTO TERMS (tid, did, tf) VALUES (1, 300, 2), (5, 300, 1), (2,
     300, 1);
```

B.4.4 SQL Delete Scripts T_5

Listing B.15: SQL Delete DML for D_1

```sql
UPDATE docs set removed = T5 WHERE did = 100;
UPDATE DICT_HIST SET removed = T5 WHERE tid IN (1,2) AND (removed IS
    NULL AND added<T5);
INSERT INTO DICT_HIST (tid, added, df) VALUES (1, T5, 1), (2, T5, 1);
```

B.4.5 SQL Insert Scripts D_{1v2}

Listing B.16: SQL DML for D_1

```sql
INSERT INTO DICT (term) VALUES ('Mathison');
INSERT INTO DOCS (did, name, len, added) VALUES (101, 'Doc 1', 3, T5);

INSERT INTO TERMS (tid, did, tf) VALUES (1, 101, 1), (6, 101, 1), (2,
     101, 1);
```

```
UPDATE docs set removed = T5 WHERE did = 100;
UPDATE DICT_HIST SET df = 2 WHERE tid IN (1,2) AND removed IS NULL;
INSERT INTO DICT_HIST (tid, added, df) VALUES (6, T5, 1);
```

B.4.6 Query Sub Scores

```
WITH
/* filter valid documents and calculate avg(len) and N */
qdocs AS (SELECT * FROM docs WHERE added <= timest AND (removed IS
    NULL OR removed > timest)),

/* valid terms containing one of the search strings */
qterms AS (SELECT terms.tid, terms.did, dict.term, terms.tf,
    dict_hist.df, qdocs."len" FROM
dict JOIN dict_hist
ON dict_hist.tid = dict.tid
JOIN terms ON terms.tid = dict.tid
JOIN qdocs ON qdocs.did = terms.did
WHERE dict_hist.added <= timest AND (dict_hist.removed is null OR
    dict_hist.removed > timest) AND term IN (term1, term2, term3)
),

/* average document length (avg(len)) */
len_avg AS (SELECT avg(len) AS anr FROM qdocs),

/* total number of documents (N) */
doc_nr AS (SELECT count(*) AS tnr from qdocs)

/* document term scores */
SELECT qterms.did, qterms."len", qterms.tid, qterms.df, qterms.tf, (
    SELECT tnr FROM doc_nr) AS n, (SELECT anr FROM len_avg) as av,
(log(((SELECT tnr FROM doc_nr) + 0.0)/(qterms.df)) * qterms.tf * (1.2
     + 1) / (qterms.tf + 1.2 * (1 - 0.75 + 0.75 * ((qterms."len")/((
    SELECT anr FROM len_avg)))))) AS subscore
FROM qterms;
```

B.5 Correctness Setup for VCSM4

B.5.1 Table Creations & Constraints

Listing B.17: SQL Create Table for VCSM4

```
CREATE TABLE dict (tid INT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    term VARCHAR(100) UNIQUE NOT NULL);
CREATE TABLE docs (did INT PRIMARY KEY, "len" INTEGER NOT NULL, added
     TIMESTAMP DEFAULT CURRENT_TIMESTAMP(), removed TIMESTAMP, name
    VARCHAR(100) NOT NULL, del BOOLEAN DEFAULT FALSE);
CREATE TABLE terms (tid INT NOT NULL, did INT NOT NULL, tf INT NOT
    NULL, removed_df TIMESTAMP, df INT NOT NULL);
```

B.5.2 SQL Insert Scripts D_1, D_2

Listing B.18: SQL Insert Scripts for D_1 & D_2

```
INSERT INTO DICT (term) VALUES ('Alan'), ('Turing'), ('Aileen'), ('
    Kay');
INSERT INTO DOCS (did, name, len, added) VALUES (100, 'Doc 1', 2, T1),
     (200, 'Doc 2', 2, T1);
INSERT INTO TERMS (tid, did, tf, df) VALUES (1, 100, 1, 1), (2, 100,
    1, 1), (3, 200, 1, 1), (4, 200, 1, 1);
```

B.5.3 SQL Insert Scripts D_3

Listing B.19: SQL Insert Scripts for D_3

```
INSERT INTO DICT (term) VALUES ('Mygrott');
INSERT INTO DOCS (did, name, len, added) VALUES (300, 'Doc 3', 4, T3);

UPDATE TERMS SET removed_df = T3 WHERE tid IN (1,2) AND did = 100;
INSERT INTO TERMS (tid, did, tf, df) VALUES (1, 300, 2, 2), (5, 300,
    1, 1), (2, 300, 1, 2);
```

B.5.4 SQL Delete Scripts T_5

Listing B.20: SQL Delete DML for D_1

```
UPDATE docs set removed = T5, del = true WHERE did = 100;
UPDATE TERMS SET removed_df = T5 WHERE tid IN (1,2) AND (removed_df
    IS NULL);
INSERT INTO TERMS (tid, did, tf, df) VALUES (1, 100, 2, 1), (2, 100,
    1, 1);
```

B.5.5 SQL Insert Scripts D_{1v2}

Listing B.21: SQL Update for D_1

```
INSERT INTO DICT (term) VALUES ('Mathison');
INSERT INTO DOCS (did, name, len, added) VALUES (101, 'Doc 1', 3, T5);

/*undo deletion of previous step...undo and delete are strictly
    distinguished */
DELETE FROM TERMS WHERE tid IN (1,2) and removed_df IS null;
INSERT INTO TERMS (tid, did, tf, df) VALUES (1, 101, 1, 2), (6, 101,
    1, 1), (2, 101, 1, 2);
```

B.5.6 Query Sub Scores

```
WITH

qdocs AS (SELECT * FROM docs WHERE docs.del OR (added <= timest AND (
    removed IS NULL OR removed > timest))),

dterms AS (SELECT terms.tid, terms.did, terms.tf, qdocs."len", qdocs.
    added, qdocs.removed, terms.removed_df, terms.df as df, qdocs.del
    FROM terms
JOIN qdocs ON qdocs.did = terms.did
JOIN dict ON terms.tid = dict.tid
WHERE dict.term IN ('Alan', 'Mathison', 'Turing')),

term_df AS (SELECT tid, df FROM dterms WHERE (not del AND added <=
    timest AND (removed_df is NULL OR timest < removed_df))
OR (del AND removed_df is NULL AND removed < timest) OR (del and
    removed_df > timest and removed > timest)),

qterms AS (select tid, did, tf, len FROM dterms WHERE not del OR (del
     AND timest < removed AND removed_df IS NOT NULL)),

qstats AS (SELECT avg(len) AS anr, count(*) AS tnr FROM qdocs where
    removed > timest OR removed is null)

SELECT qterms.did, qterms."len", qterms.tid, term_df.df, qterms.tf, (
    SELECT tnr FROM qstats) AS n, (SELECT anr FROM qstats) as av,
(log(((SELECT tnr FROM qstats) + 0.0)/(term_df.df)) * qterms.tf *
    (1.2 + 1) / (qterms.tf + 1.2 * (1 - 0.75 + 0.75 * ((qterms."len")
    /((SELECT anr FROM qstats)))))) AS subscore
FROM qterms
JOIN term_df ON qterms.tid = term_df.tid;
```

B.6 Correctness Setup for VCSM5

B.6.1 Table Creations & Constraints

Listing B.22: SQL Create Table for Model #5

```
CREATE TABLE dict (tid INT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    term VARCHAR(100) UNIQUE NOT NULL);
CREATE TABLE docs (did INT PRIMARY KEY, name VARCHAR(100) NOT NULL);
CREATE TABLE terms (tid INT NOT NULL, did INT NOT NULL, added
    TIMESTAMP NOT NULL, removed TIMESTAMP, tf INT NOT NULL, removed_df
     TIMESTAMP, df INT NOT NULL, del BOOLEAN DEFAULT FALSE);
```

B.6.2 SQL Insert Scripts D_1, D_2

Listing B.23: SQL Insert Scripts for D_1 & D_2

```
INSERT INTO DICT (term) VALUES ('Alan'), ('Turing'), ('Aileen'), ('
    Kay');
INSERT INTO DOCS (did, name) VALUES (100, 'Doc 1'), (200, 'Doc 2');
INSERT INTO TERMS (tid, did, added, tf, df) VALUES (1, 100, T1, 1, 1),
     (2, 100, T1, 1, 1), (3, 200, T1, 1, 1), (4, 200, T1, 1, 1);
```

B.6.3 SQL Insert Scripts D_3

Listing B.24: SQL Insert Scripts for D_3

```sql
INSERT INTO DICT (term) VALUES ('Mygrott');
INSERT INTO DOCS (did, name) VALUES (300, 'Doc 3');
UPDATE TERMS SET removed_df = T3 WHERE tid IN (1,2) AND did = 100;
INSERT INTO TERMS (tid, did, added, tf, df) VALUES (1, 300, T3, 2, 2),
    (5, 300,T3, 1, 1), (2, 300, T3, 1, 2);
```

B.6.4 SQL Delete Scripts T_5

Listing B.25: SQL Delete DML for D_1

```sql
UPDATE TERMS SET removed_df = T5, removed=T5 WHERE tid IN (1,2) AND (
    removed_df IS NULL);
INSERT INTO terms (tid, did, added, tf, df) VALUES (1, -1, T5, -1, 1),
    (2, -1, T5, -1, 1);
```

B.6.5 SQL Update Scripts D_{1v2}

Listing B.26: SQL Update for D_1

```sql
INSERT INTO DICT (term) VALUES ('Mathison');
INSERT INTO DOCS (did, name, len, added) VALUES (101, 'Doc 1');
/*undo deletion of previous step...undo and delete are strictly
    distinguished */
DELETE FROM TERMS WHERE did = -1;
INSERT INTO TERMS (tid, did,added, tf, df) VALUES (1, 101, T5, 1, 2),
    (6, 101, T5, 1, 1), (2, 101, T5, 1, 2);
```

B.6.6 Query Sub Scores

```sql
WITH

/* filter terms valid according to the respective time frame */
dterms AS (SELECT terms.tid, terms.did, added, removed_df, terms.tf,
    terms.df
FROM terms
WHERE added <= timest AND (removed IS NULL OR removed > timest)),

/* extract list of documents */
qdocs AS (SELECT dterms.did, sum(tf) as "len" from dterms GROUP BY
    dterms.did),

/* filtering terms relevant to the query terms */
qterms AS (SELECT dterms.tid, dterms.did, dterms.tf, dterms.df FROM
    dterms JOIN dict ON dterms.tid = dict.tid WHERE dict.term IN ('
    Alan', 'Mathison', 'Turing')),

/* extract document frequency values */
```

```
term_df AS (SELECT tid, df from dterms WHERE added <= timest AND (
    removed_df IS NULL OR removed_df > timest)),

/* average document length (avg(len)) and total number of documents
    */
stats AS (SELECT avg(len) AS anr, count(*) as tnr FROM qdocs)

/* computing sub query scores */
SELECT qterms.did, qterms.tid, qdocs."len", term_df.df, qterms.tf, (
    SELECT tnr FROM stats) AS n, (SELECT anr FROM stats) as av,
(log(((SELECT tnr FROM stats)+0.0)/(term_df.df)) * qterms.tf * (1.2 +
     1) / (qterms.tf + 1.2 * (1 - 0.75 + 0.75 * ((qdocs."len")/((
    SELECT anr FROM stats)))))) AS subscore
FROM qterms
JOIN qdocs ON qdocs.did = qterms.did
JOIN term_df ON qterms.tid = term_df.tid;
```

B.7 Correctness Setup for VCSM6

B.7.1 Table Creations & Constraints

Listing B.27: SQL Create Table for VCSM5

```
CREATE TABLE versions (vid INT PRIMARY KEY, added TIMESTAMP NOT NULL,
     removed TIMESTAMP, del BOOLEAN);
CREATE TABLE dict (tid INT PRIMARY KEY GENERATED ALWAYS AS IDENTITY,
    term VARCHAR(100) UNIQUE NOT NULL);
CREATE TABLE docs (did INT NOT NULL, vid INT NOT NULL, "len" INT NOT
    NULL, name VARCHAR(100) NOT NULL);
CREATE TABLE terms (tid INT NOT NULL, did INT NOT NULL, vid INT NOT
    NULL, tf INT NOT NULL);
ALTER TABLE DOCS ADD CONSTRAINT docs_did_pkey PRIMARY KEY (did, vid);
```

B.7.2 SQL Insert Scripts D_1, D_2

Listing B.28: SQL Insert Scripts for D_1 & D_2

```
INSERT INTO VERSIONS (vid, added, del) VALUES (1, T1, false);
INSERT INTO DICT (term) VALUES ('Alan'), ('Turing'), ('Aileen'), ('
    Kay');
INSERT INTO DOCS (did, vid, name, len) VALUES (100, 1, 'Doc 1', 2),
    (200, 1, 'Doc 2', 2);
INSERT INTO TERMS (tid,vid, did, tf) VALUES (1, 1, 100, 1), (2, 1,
    100, 1), (3, 1, 200, 1), (4, 1, 200, 1);
```

B.7.3 SQL Insert Scripts D_3

Listing B.29: SQL Insert Scripts for D_3

```
UPDATE versions SET removed = T3 WHERE vid = 1;
```

```
INSERT INTO VERSIONS (vid, added, del) VALUES (2, T3, false);
INSERT INTO DICT (term) VALUES ('Mygrott');
INSERT INTO DOCS (did, vid, name, len) VALUES (300, 2, 'Doc 3', 4);
INSERT INTO TERMS (tid,vid, did, tf) VALUES (1, 2, 300, 2), (5, 2,
    300, 1), (2, 2, 300, 1);
```

B.7.4 SQL Delete Scripts T_5

Listing B.30: SQL Delete DML for D_1

```
UPDATE versions SET removed = T5 WHERE vid = 2;
INSERT INTO VERSIONS (vid, added, del) VALUES (3, T5, true);
/* Document one with vid 3 is reinserted but marked for deletion */
INSERT INTO DOCS (did, vid, name, len) VALUES (100, 3, 'Doc 1', 2);
```

B.7.5 SQL Update Scripts D_{1v2}

Listing B.31: SQL Update for D_1

```
/*undo previous deletion of document one and substitute with update
    */
UPDATE docs SET len = 3 where did = 100 and vid = 3;
UPDATE versions SET del = false WHERE vid = 3;
INSERT INTO DICT (term) VALUES ('Mathison');
INSERT INTO VERSIONS (vid, added, del) VALUES (3, T5, false);
INSERT INTO DOCS (did, vid, name, len) VALUES (100, 3, 'Doc 1', 3);
INSERT INTO TERMS (tid,vid, did, tf) VALUES (1, 3, 100, 1), (6, 3,
    100, 1), (2, 3, 100, 1);
```

B.7.6 Query Sub Scores

```
WITH
/*valid document terms */
qvers AS (SELECT vid, del from versions WHERE added <= timest AND (
    removed IS NULL OR removed > timest)),

/* select the latest document versions of documents, exclude removed
    versions */
qdocs AS (SELECT b.did, b.vid , b.len FROM
(SELECT did, max(vid) as vid FROM docs WHERE vid <= (SELECT vid from
    qvers) GROUP BY did, len) as ldocs
JOIN docs b ON ldocs.did = b.did AND ldocs.vid = b.vid
JOIN versions ON versions.vid = ldocs.vid
WHERE versions.del = false),
/* selection of qualified terms */
qterms AS (SELECT terms.tid, terms.did, terms.vid, tdic.term, terms.
    tf FROM
(SELECT tid, term FROM dict WHERE term IN ('Alan', 'Mathison', '
    Turing')) AS tdic
JOIN terms ON terms.tid = tdic.tid
JOIN qdocs ON qdocs.did = terms.did AND qdocs.vid = terms.vid ),
/* */
stats AS (SELECT avg(len) AS anr, count(*) AS tnr FROM qdocs),

## number of documents containing search term (df)
term_df AS (SELECT tid, count(tid) AS df from qterms GROUP BY tid)
## document term scores

SELECT qdocs.did, qdocs.vid , qdocs."len", qterms.tid, term_df.df,
    qterms.tf, (SELECT tnr FROM stats) AS n, (SELECT anr FROM stats)
    as av,
(log(((SELECT tnr FROM stats) + 0.0)/(term_df.df)) * qterms.tf * (1.2
     + 1) / (qterms.tf + 1.2 * (1 - 0.75 + 0.75 * ((qdocs."len")/((
    SELECT anr FROM stats)))))) AS subscore
FROM qterms
JOIN qdocs ON qterms.did=qdocs.did AND qterms.vid = qdocs.vid
JOIN term_df ON term_df.tid = qterms.tid;
```

APPENDIX C

CRUD Operations Used for Performance Evaluation

C.1 Shift from VCSM2 to VCSM3, Creating the Dictionary History

To create the schema VCSM2 from, the following SQL-statements were used.

C.1.1 Creating Derived Table to Reflect Different Batch Sizes and Import Timings in Documents

Create a derived document table with row numbers.

```

create table docs_batchx
  (row int, rnk int default 0, did int, revid int,
  name varchar(100), added TIMESTAMP, removed TIMESTAMP, len int);

INSERT INTO docs_batchx
  select row_number() over (order by did) as row, 0, did, revid,
  name, added, removed, len from docs1;
```

C.1.2 Updating Document Timestamps in the Batch Interval Desired

```
DECLARE beg timestamp;
SET beg = '2011-11-11';
DECLARE batSize, incrementSec int;
SET batsize = 500;
SET incrementSec = 360;

```

```
UPDATE docs_batchx set added = (select beg) + (row - ((row-1)%batsize
    )) / batsize * incrementSec, rnk = (row - ((row-1)%batsize)) /
    batsize +1 ;

UPDATE docs_batchx set rnk = (row - ((row-1)%batsize)) / batsize +1 ;
```

C.1.3 Derive Rank of Batch Ingest Timings

```
create table rnk_timestamp as select row_number() over () as rnk,
 added as stamp from docs_batchx group by added order by added asc;
```

C.1.4 Get Rank for Timestamped Documents in Collection

```
# table reflects the ingest rank of documents with respect to the
    batch sizes
CREATE table tid_rnk_dfX (rnk int, tid int, df int);

#computationally expensive operation, potentially split up by
    defining tid ranges (=factor)!
  (use factor <10 as an advice
  ##exponential growth because of distribution of terms
CREATE PROCEDURE fill_de_scheiss (factor int)
BEGIN
DECLARE start, maxtid INT;
SET maxtid = (select count(*) from dict);
SET start = 1;
WHILE (start <= maxtid) DO
INSERT INTO tid_rnk_dfX
  SELECT rnk, tid, max(rownum) AS df FROM
    (SELECT rnk, tid, row_number() over (partition by tid ORDER BY rnk
       ) AS rownum
    FROM terms2
    JOIN docs_batchx ON terms2.did = docs_batchx.did
    AND terms2.revid = docs_batchx.revid
    where tid >= start
    and tid < start * factor) as sub
GROUP BY sub.rnk, sub.tid;
SET start = start * factor;
END WHILE;
END;

#calling procedure
call fill_de_scheiss(10);
```

C.1.5 Force Ordering Numbers on Ranked Documents

This step is again separated because the huge numbers of insert operations are computationally expensive. To speed this process up another temporary table is created and insertion tuples are divided into smaller portions to be handled by the database.

```sql
CREATE table row_rnk_tid (row int, tid int, stamp timestamp, df int);

CREATE PROCEDURE fill_de_rowrank(factor int)
BEGIN
DECLARE start, maxtid INT;
SET maxtid = (select count(*) from dict);
SET start = 1;
WHILE (start <= maxtid) DO
  INSERT INTO row_rnk_tid (row, tid, stamp, df)
    SELECT row_number() over(partition by tid order by t1.rnk) as row,
         tid, t1.stamp as stamp, df
    FROM
      tid_rnk_dfX rtid
      JOIN
      rnk_timestamp t1
      ON rtid.rnk = t1.rnk
      where tid >= start and tid < start * factor
      #optional ordering
      #order by tid, t1.rnk;
SET start = start * factor;
END WHILE;
END;
```

C.1.6 Creating Actual History Table, and Verifying df Result Values

```sql
CREATE TABLE wiki.dict_hist_x1000 (tid int, added timestamp, removed
    timestamp, df int);

INSERT INTO wiki.dict_hist_x1000 (tid, added, removed, df)
SELECT r1.tid, r1.stamp as added, r2.stamp as removed, r1.df
  FROM row_rnk_tid r1
  LEFT JOIN
  row_rnk_tid r2
  ON r1.row = r2.row-1
  AND r1.tid = r2.tid;
```

C.1.7 Verify Results

```sql
WITH
  ## fill in desired parameters
  param as (select 134 as tid, '2012-04-03 23:11:11.000000' as stamp),
  sub1 as (select df from dict_hist_x where added = (select stamp from
       param) and tid = (select tid from param)),
  sub2 as (select count(*) as df from terms2 join docs_batchx on
      terms2.did = docs_batchx.did
    where terms2.tid= (select tid from param) and docs_batchx.added <=
        (select stamp from param))
select sub1.df, sub2.df, sub1.df = sub2.df as approved from sub1,
    sub2;
```

C.1.8 Celanup

```
drop table dict_hist_x;
drop table row_rnk_tid;
drop procedure fill_de_scheiss;
drop procedure fill_de_rowrank;
drop table tid_rnk_dfX;
drop table rnk_timestamp;
drop procedure update_docsBatchX;
drop table docs_batchx;
```

C.2 Baseline Comparison Queries

Hereafter, the queries for the baseline evaluations are listed.

C.2.1 Baseline BM25-Query

Listing C.1: Baseline Query

```
WITH
qterms AS (SELECT tid, did FROM terms1 WHERE tid in (%QUERY%),
subscores AS (
select term_tf.did,
(log(((3034603) - qdict.df + 0.5)/(qdict.df + 0.5)) * term_tf.tf *
    (1.2 + 1) / (term_tf.tf + 1.2 * (1 - 0.75 + 0.75 * ((docs1."len")
    /(( 898.9168003194732)))))) AS subscore
FROM
(select did, tid, count(*) as tf from qterms group by did, tid) as
    term_tf
join docs1
on docs1.did = term_tf.did
join dict_base qdict
on qdict.tid = term_tf.tid
join (select did from qterms group by did having count(distinct tid)
    =3) cdocs
on cdocs.did = docs1.did
)
SELECT did, score FROM
(SELECT did, sum(subscore) AS score
FROM subscores GROUP BY did) AS scores
ORDER BY score DESC LIMIT 50;
```

C.2.2 VCSM1 BM25-Query

Listing C.2: VCSM1 Baseline Query

```
WITH
qdocs AS (SELECT * FROM docs1 WHERE added <= timest),# AND (removed
    IS NULL OR removed > timest)),
qterms AS (SELECT terms1.tid, terms1.did FROM
terms1 JOIN qdocs ON qdocs.did = terms1.did
where terms1.tid in (%QUERY%)),
term_tf AS (SELECT tid, did, COUNT(*) AS tf FROM qterms GROUP BY tid,
    did),
term_df AS (SELECT tid, count(tid) AS df from term_tf GROUP BY tid),
subscores AS (
SELECT qdocs.did,
(log(((3034603) - term_df.df + 0.5)/(term_df.df + 0.5)) * term_tf.tf
   * (1.2 + 1) / (term_tf.tf + 1.2 * (1 - 0.75 + 0.75 * ((qdocs."len"
   )/((898.9168003194732)))))) AS subscore
FROM term_tf
JOIN qdocs ON term_tf.did=qdocs.did
JOIN term_df ON term_df.tid = term_tf.tid)
SELECT subscores.did, sum(subscores.subscore) AS score
FROM subscores GROUP BY subscores.did
ORDER BY score desc LIMIT 50;
```

C.3 Dictionary Thresholds

The following SQL-statements show how we filled the dictionary table purged by the tids complying to the defined threshold criteria. Those tids contained in 25% of all documents and those only contained fewer than 6 times are excluded!

Listing C.3: Pruning the Dictionary Table Applying tid Thresholds

```
CREATE TABLE wiki.tids_ommited (tid int, df int);

#finding upper bound values
INSERT into wiki.tids_ommited
SELECT tid, count(*)
FROM terms2
group by tid
having count(*) > (select count(*) * 0.25 from docs1);

#excluding lower bound values
INSERT into wiki.tids_ommited
SELECT tid, count(*)
FROM terms2
group by tid
having count(*) <= 5;

CREATE TABLE wiki.dict (tid int, term varchar(100));
INSERT INTO wiki.dict (tid, term)
SELECT tid, term from dict
where not exists
(select tid, df from wiki.tids_ommited o where o.tid = dict.tid);
```

C.4 Measuring Update Performance

Listing C.4: Measuring Time to Update Indices on VCSM1

```
CREATE TABLE wiki.tids_ommited (tid int, df int);

CREATE PROCEDURE updates1 (numOfUpdates INT, batchsize INT)
BEGIN
DECLARE i INT;
declare timest TIMESTAMP;
set timest = now();
SET i = 0;
WHILE (i<numOfUpdates) DO
SET timest = timest + interval '1' hour;
update docs set removed = timest
where exists
(select did from docs_sample_10000
      where docs.did=docs_sample_10000.did
      and docs.revid=docs_sample_10000.revid
      and docs_sample_10000."L1" > (i * batchsize)
      and docs_sample_10000."L1" <= (i+1) * batchsize);
set i = i+1;
END WHILE;
END;

(select tid, df from wiki.tids_ommited o where o.tid = dict.tid);
```

C.5 BM25- Retrieval Queries

The following queries have been used for benchmarking the different data models.

C.5.1 VCSM1 BM25 Query

```
WITH
qdocs AS (SELECT * FROM wiki.docs1 WHERE added <= T1 AND (removed IS
    NULL OR removed > T1)),
qdict AS (SELECT * FROM wiki.dict WHERE dict.term in (%QUERY%)),
qterms AS (SELECT terms1.tid, terms1.did FROM qdict
JOIN wiki.terms1 ON terms1.tid = qdict.tid
JOIN qdocs ON qdocs.did = terms1.did),
stats AS (SELECT avg(len) AS anr, count(*) AS tnr FROM qdocs),
term_tf AS (SELECT tid, did, COUNT(*) AS tf FROM qterms GROUP BY tid,
     did),
term_df AS (SELECT tid, count(tid) AS df from term_tf GROUP BY tid),
subscores AS (
SELECT qdocs.did, qdocs."len", term_tf.tid, term_df.df, term_tf.tf, (
    SELECT tnr FROM stats) AS n, (SELECT anr FROM stats) as av,
(log(((SELECT tnr FROM stats) - term_df.df + 0.5)/(term_df.df + 0.5))
    * term_tf.tf * (1.2 + 1) / (term_tf.tf + 1.2 * (1 - 0.75 + 0.75 *
    ((qdocs."len")/((SELECT anr FROM stats)))))) AS subscore
FROM term_tf
JOIN qdocs ON term_tf.did=qdocs.did
JOIN term_df ON term_df.tid = term_tf.tid)
SELECT subscores.did, sum(subscores.subscore) AS rnk
FROM subscores GROUP BY subscores.did ORDER BY rnk desc
LIMIT 10;
```

C.5.2 VCSM2 BM25 Query

```
WITH
qdocs AS (SELECT * FROM wiki.docs1 WHERE added <= T1 AND (removed IS
    NULL OR removed > T1)),
qdict AS (select tid, term from wiki.dict where term in (%QUERY%)),
qterms AS (SELECT qdict.tid, qdocs.did, terms2.tf, qdocs."len" from
    qdict
  JOIN wiki.terms2 on qdict.tid = terms2.tid
  JOIN qdocs ON qdocs.did = terms2.did),
stats AS (SELECT avg(len) as anr, count(*) as tnr FROM qdocs),
term_df AS (SELECT tid, count(*) as df FROM qterms group by tid),
subscores AS (
  SELECT qdocs.did, qdocs."len", qterms.tid, term_df.df, qterms.tf, (
      SELECT tnr FROM stats) AS n, (SELECT anr FROM stats) as av, (log
      (((SELECT tnr FROM stats) - term_df.df + 0.5)/(term_df.df + 0.5))
       * qterms.tf * (1.2 + 1) / (qterms.tf + 1.2 * (1 - 0.75 + 0.75 *
       ((qdocs."len")/((SELECT anr FROM stats)))))) AS subscore
FROM qterms
  JOIN qdocs ON qterms.did=qdocs.did
  JOIN term_df ON term_df.tid = qterms.tid)
SELECT subscores.did, sum(subscores.subscore) AS rnk
FROM subscores GROUP BY subscores.did ORDER BY rnk desc LIMIT 10;
```

C.5.3 VCSM3 BM25 Query

```
WITH
qdocs AS (SELECT * FROM wiki.docs1 WHERE added <= T1 AND (removed IS
    NULL OR removed > T1)),
qdict AS (SELECT tid FROM wiki.dict WHERE dict.term IN (%QUERY%)),
tqdict AS (SELECT qdict.tid, df FROM wiki.dict_hist JOIN qdict on
    dict_hist.tid = qdict.tid where dict_hist.added <= T1 AND (removed
     IS NULL OR removed > T1)),
qterms AS (SELECT terms2.tid, terms2.did, terms2.tf, tqdict.df, qdocs.
    "len"
FROM tqdict
  JOIN wiki.terms2 ON terms2.tid = tqdict.tid
  JOIN qdocs ON qdocs.did = terms2.did),
stats AS (SELECT avg(len) as anr, count(*) as tnr FROM qdocs),
subscores AS (
SELECT qdocs.did, qdocs."len", qterms.tid, qterms.df, qterms.tf, (
    SELECT tnr FROM stats) AS n, (SELECT anr FROM stats) as av, (log
    (((SELECT tnr FROM stats) - qterms.df + 0.5)/(qterms.df + 0.5)) *
    qterms.tf * (1.2 + 1) / (qterms.tf + 1.2 * (1 - 0.75 + 0.75 * ((
    qdocs."len")/((SELECT anr FROM stats)))))) AS subscore
FROM qterms
 JOIN qdocs ON qterms.did=qdocs.did)
SELECT subscores.did, sum(subscores.subscore) AS rnk
FROM subscores GROUP BY subscores.did ORDER BY rnk desc
LIMIT 10;
```

C.5.4 VCSM4 BM25 Query

```
WITH
qdocs AS (SELECT * FROM wiki.docs1 WHERE added <= T1 AND (removed IS
    NULL OR removed > T1)),
qdict AS (SELECT tid FROM wiki.dict WHERE dict.term IN (%QUERY%)),
qterms AS (
SELECT terms.tid, terms.did, docs.added, docs.removed, terms.rem_df,
    terms.tf, docs.len, terms.df
FROM wiki.terms4 terms
join qdict dict on terms.tid = dict.tid
join qdocs docs on docs.did = terms.did),
term_df AS (SELECT distinct df, tid from qterms where added <= T1 and
     (rem_df is null or rem_df > T1)),
param AS (SELECT avg(len) AS anr, count(*) as tnr FROM qdocs),
subscores AS (
SELECT qterms.did, qterms.tid, qterms."len", term_df.df, qterms.tf, (
    SELECT tnr FROM param) AS n, (SELECT anr FROM param) as av, (log
    (((SELECT tnr FROM param) - term_df.df + 0.5)/(term_df.df + 0.5))
    * qterms.tf * (1.2 + 1) / (qterms.tf + 1.2 * (1 - 0.75 + 0.75 * ((
    qterms."len")/((SELECT anr FROM param)))))) AS subscore
FROM qterms
JOIN term_df on term_df.tid = qterms.tid)
SELECT subscores.did, sum(subscores.subscore) AS rnk
FROM subscores GROUP BY subscores.did ORDER BY rnk desc
LIMIT 10;
```

C.5.5 VCSM5 BM25 Query

```
WITH
qdict AS (SELECT tid FROM wiki.dict WHERE dict.term IN (%QUERY%)),
tterms AS (SELECT terms.tid, terms.did, terms.added, terms.removed,
    terms.rem_df, terms.len, terms.tf, terms.df
from wiki.terms5 terms
where added <= T1 AND (removed IS NULL OR removed > T1)),
qdocs AS (SELECT did, max(len) as len from tterms group by did),
qterms as (select terms.tid, terms.did, terms.added, terms.removed,
    terms.rem_df, terms.len, terms.tf, terms.df
from tterms terms
```

```sql
where terms.tid in (select tid from qdict)),
term_df AS (SELECT distinct df, tid from terms5 where rem_df is null
    or rem_df > T1),
param AS (SELECT avg(len) AS anr, count(*) as tnr FROM qdocs),
subscores AS (
SELECT qterms.did, qterms.tid, qterms."len", term_df.df, qterms.tf, (
    SELECT tnr FROM param) AS n, (SELECT anr FROM param) as av, (log
    (((SELECT tnr FROM param) - term_df.df + 0.5)/(term_df.df + 0.5))
    * qterms.tf * (1.2 + 1) / (qterms.tf + 1.2 * (1 - 0.75 + 0.75 * ((
    qterms."len")/((SELECT anr FROM param)))))) AS subscore
FROM qterms
JOIN term_df on term_df.tid = qterms.tid)
SELECT subscores.did, sum(subscores.subscore) AS rnk
FROM subscores GROUP BY subscores.did ORDER BY rnk desc
LIMIT 10;
```

C.5.6 VCSM6 BM25 Query

```sql
WITH
qversion AS (SELECT vid from versions WHERE added <= T1 AND (removed
    IS NULL OR removed > T1)),
qdocs AS (SELECT vvalid.did, vvalid.vid, docs.name, docs.len
FROM (SELECT docs.did, max(docs.vid) as vid FROM wiki.docs6 docs
    WHERE vid <= (select vid from qversion) group by docs.did) as
    vvalid
join docs6 docs on docs.did = vvalid.did and docs.vid = vvalid.vid
join versions versions on versions.vid = docs.vid
where not versions.deleted),
qdict AS (select tid, term from wiki.dict where term in (%QUERY%)),
qterms AS (SELECT qdict.tid, qdocs.did, qdocs.vid, t.tf, qdocs."len"
from qdict
JOIN wiki.terms6 t on qdict.tid = t.tid
JOIN qdocs ON qdocs.did = t.did and qdocs.vid = t.vid),
stats AS (SELECT avg(len) as anr, count(*) as tnr FROM qdocs),
term_df AS (SELECT tid, count(*) as df FROM qterms group by tid),
subscores AS (SELECT qdocs.did, qdocs.vid, qdocs."len", qterms.tid,
    term_df.df, qterms.tf, (SELECT tnr FROM stats) AS n, (SELECT anr
    FROM stats) as av, (log(((SELECT tnr FROM stats) - term_df.df +
    0.5)/(term_df.df + 0.5)) * qterms.tf * (1.2 + 1) / (qterms.tf +
    1.2 * (1 - 0.75 + 0.75 * ((qdocs."len")/((SELECT anr FROM stats)))
    ))) AS subscore
FROM qterms
JOIN qdocs ON qterms.did=qdocs.did and qterms.vid = qdocs.vid
JOIN term_df ON term_df.tid = qterms.tid)
SELECT subscores.did, subscores.vid, sum(subscores.subscore) AS rnk
FROM subscores GROUP BY subscores.did, subscores.vid ORDER BY rnk
    desc
LIMIT 10;
```

Bibliography

[ABH$^+$13] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[ACD$^+$15] Jaime Arguello, Matt Crane, Fernando Diaz, Jimmy Lin, and Andrew Trotman. Report on the sigir 2015 workshop on reproducibility, inexplicability, and generalizability of results (rigor). *ACM SIGIR Forum*, 12(2):107–116, 2015.

[AMH08] Daniel J. Abadi., Samuel R. Madden, and Nabil Hachem. 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, pages 967–980, New York, NY, USA, June 2008. ACM.

[AMWZ09] Timothy Armstrong, Alistair Moffat, William Webber, and Justin Zobel. Improvements that don't add up: ad-hoc retrieval results since 1998. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM09)*, pages 601–610. ACM, 2009.

[BCC10] Stefan Büttcher, Charles LA Clarke, and Gordon V Cormack. *Information retrieval: Implementing and evaluating search engines.* Mit Press, 2010.

[BGT09] Truls A. Bjørklund, Johannes Gehrke, and Øystein Torbjørnsen. A confluence of column stores and search engines: Opportunities and challenges. In *Proceedings of the 2009 ACM VLDB workshop on sing Search Engine Technology for Information Management (USETIM09)*, 2009.

[BLY14] Vimala Balakrishnan and Ethel Lloyd-Yemoh. Stemming and lemmatization: A comparison of retrieval performances. *Lecture Notes on Software Engineering*, 02(3):262–267, Aug 2014. The authors are with the Faculty of Computer Science and Information Systems, University of Malaya, Kuala Lumpur, Malaysia (e-mail: vimala.balakrishnan@um.edu.my, ethel_lloyd@siswa.um.edu.my).

[Bus10] Michael Busch. Twitter's new search architecture. https://blog.twitter.com/2010/twitters-new-search-architecture, Oct 2010.

[BZN05] Peter Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *In CIDR*, 2005.

[CDM$^+$09] Hinrich Schütze Hinrich Christopher D. Manning, Prabhakar Raghavan et al. *Introduction to information retrieval*, volume 1. Cambridge University Press Cambridge, 2009.

[CK85] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. *SIGMOD Record*, 14(4):268–279, May 1985.

[CPW15] Christian Collberg, Todd Proebsting, and Alex M Warren. Repeatability and benefaction in computer systems research - a study and a modest proposal. Tech report, University of Arizona TR 14-04, 2015.

[CRW05] Surajit Chaudhuri, Raghu Ramakrishnan, and Gerhard Weikum. Integrating db and ir technologies: What is the sound of one hand clapping. In *In CIDR*, pages 1–12, 2005.

[ELR15] Sarah Edwards, Xuan Liu, and Niky Riga. Creating repeatable computer science and networking experiments on shared, public testbeds. *SIGOPS - Operating Systems Review.*, 49(1):90–99, Jan 2015.

[FBS12] Juliana Freire, Philippe Bonnet, and Dennis Shasha. Computational reproducibility: State-of-the-art, challenges, and database research opportunities. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 593–596, New York, NY, USA, 2012. ACM.

[Fou12] The Apache Software Foundation. Apache lucene. https://lucene.apache.org/core/, 2012.

[Gra94] G. Graefe. Volcano: An extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, Feb 1994.

[HK06] Hans-Werner Hilse and Jochen Kothe. *Implementing Persistent Identifiers.* Consortium of European Research Libraries, London, www.cerl.org European Commission on Preservation and Access, Amsterdam, www.knaw.nl/ecpa, Nov 2006.

[HRF15] Matthias Hagen, Jinfeng Rao, and Nicola Ferro. Advances in information retrieval. In Allan Hanbury, Gabriella Kazai, Andreas Rauber, and Norbert Fuhr, editors, *37th European Conference on IR Research, ECIR 2015, Vienna, Austria, March 29 - April 2, 2015. Proceedings*, volume 37 of *9022*, pages 741–780, Herrengasse 13, 1010 Wien, Austria, Mar 2015. European Conference on IR Research, Springer International Publishing.

[IGN$^+$12] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Engineering Bulletin Issues*, 35(1):40–45, 2012.

[IKM07] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Updating a cracked database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 413–424, June 2007.

[IKM09] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-organizing tuple reconstruction in column-stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 297–308, New York, NY, USA, 2009. ACM.

[IKNG09] Milena G. Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo A.P. Gonçalves. An architecture for recycling intermediates in a column-store. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 309–320, New York, NY, USA, June 2009. ACM.

[Jiv11] Anjali Ganesh Jivani. A comparative study of stemming algorithms. *International Journal of Computer Technology and Applications*, 02(06):1930, Nov 2011.

[Jon72] Karen Spärck Jones. A statistical interpretation of term specifity and its application in retrieval. *Journal of Documentation*, 28(1):11–21, 1972.

[JWR00] Karen Spärck Jones, Steve Walker, and Stepfen E. Robertson. A probabilistic model of information retrieval: Development and comparative experiments part 1. *Information Processing & Management*, 36(6):779–808, Nov 2000.

[Kir13] Raffi Kirkorian. New tweets per second record, and how! https://blog.twitter.com/2013/new-tweets-per-second-record-and-how, Aug 2013. Statistics.

[KR02] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling.* John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2002.

[LZW06] Nicholas Lester, Justin Zobel, and Hugh Williams. Efficient online index maintenance for contiguous inverted lists. *Information Processing & Management*, 42(4):916–933, July 2006.

[MKB09] Stefan Manegold, Martin L. Kersten, and Peter A Boncz. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *Proceedings of the VLDB Endowment*, 2(2):1648–1653, Aug 2009.

[MLS12] Ian Mitchell, Randall LeVeque, and Victoria Stodden. Reproducible research for scientific computing: Tools and strategies for changing the culture. *Computing in Science and Engineering*, 14(4):13–17, July 2012.

[MSLdAV14] Hannes Mühleisen, Thaer Samar, Jimmy Lin, and de Arjen Vries. Column stores as an ir prototyping tool. In Maarten de Rijke, Tom Kenter, Arjen de Vries, ChengXiang Zhai, Franciska de Jong, Kira Radinsky, and Katja Hofmann, editors, *Advances in Information Retrieval*, volume 8416 of *Lecture Notes in Computer Science*, pages 789–792. Springer International Publishing, Apr 2014.

[MSLdV14] Hannes Mühleisen, Thaer Samar, Jimmy Lin, and Arjen de Vries. Old dogs are great at new tricks: Column stores for ir prototyping. In *Proceedings of the 37th International ACM SIGIR Conference on Research &#38; Development in Information Retrieval*, SIGIR '14, pages 863–866, New York, NY, USA, July 2014. ACM.

[ODCSPoC13] CODATA-ICSTI Task Group On Data Citation Standards and Out of Mind: The Current Sices PractOut of Cite. Out of cite, out of mind: The current state of practice, policy, and technology for the citation of data. *Data Science Journal*, 12, Oct 2013.

[PDF07] Heather A. Piwowar, Roger S. Day, and Douglas B. Fridsma. Sharing detailed research data is associated with increased citation rate. *PLOS ONE*, 2(3):1–5, Mar 2007.

[Por80] Martin F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[PR13a] Stefan Pröll and Andrea Rauber. Scalable data citation in dynamic, large databases: Model and reference implementation. In *2013 IEEE International Conference on Big Data*, pages 307–312, Silicon Valley, CA, Oct 2013. IEEE.

[PR13b] Stefan Pröll and Andreas Rauber. Citable by design a model for making data in dynamic environments citable. In *2nd International Conference on Data Management Technologies and Applications (DATA2013)*, July 2013.

[PV13] Heather A. Piwowar and Todd J. Vision. Data reuse and the open data citation advantage. *PeerJ*, 1:e175, Oct 2013.

[RAvUP15] Andreas Rauber, Ari Asmi, Dieter van Uytvanck, and Stefan Pröll. Data citation of evolving data, recommendations of the working group on data citation (wgdc). `https://www.rd-alliance.org/system/files/documents/RDA-DC-Recommendations_151020.pdf`, Oct 2015.

[RAvUP16] Andreas Rauber, Ari Asmi, Dieter van Uytvanck, and Stefan Pröll. Identification of reproducible subsets for data citation, sharing and re-use. *Bulletin of IEEE Technical Committee on Digital Libraries (TCDL)*, 12, May 2016.

[RJ94] Stephen Robertson and Karen Späerck Jones. Simple, proven approaches to text retrieval. Technical Report 356, University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue Cambridge CB3 0FD United Kingdom, Dec 1994.

[RZ09] Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends in Information Retrieval*, 3(4):333–389, 2009.

[SB88] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513 – 523, 1988.

[SBH14] Thomas Stadelmann, Daniel Blank, and Andreas Henrich. Implementierung von IR-modellen auf basis spaltenorientierter datenbanken oder invertierter listen. Universitat Bamberg, December 2014.

[Sha48] Claude E Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, July 1948.

[Sno00] Richard T. Snodgrass. *Developing Time-oriented Database Applications in SQL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, July 2000.

[YF16] Peilin Yang and Hui Fang. A reproducibility study of information retrieval models. In *Proceedings of the 2016 ACM International Conference on the Theory of Information Retrieval (ICTIR16)*, pages 77–86. ACM, 2016.

[ZB12] Marcin Zukowski and Peter A. Boncz. Vectorwise: Beyond column stores. *Data Engineering Bulletin Issues*, 35(1):21–27, 2012.

[ZMD16] Carlo Maria Zwölf, Nicolas Moreau, and Marie-Lise Dubernet. New model for datasets citation and extraction reproducibility in vamdc. *Journal of Molecular Spectroscopy*, 327:122–137, 2016.

Glossary

Binary Independence Model The Binary Independence Assumption is that documents are binary vectors. That is, only the presence or absence of terms in documents are recorded. Terms are independently distributed in the set of relevant documents and they are also independently distributed in the set of irrelevant documents. The representation is an ordered set of Boolean variables. That is, the representation of a document or query is a vector with one Boolean element for each term under consideration.[1] . 14

cold run Cold-runs on c-stores are queries where the system has no a-prior knowledge on the structure of the data and thus, has to consider the entire corpus for processing. In case of the column-store MonetDB cold-runs describe the first access to a database(-table) since thereafter the system has already collected data-distribution according to the input data used in the cold-run. 93, 96

column-store Is a database management system that stores data tables decomposed into columns to improve analytical workload processing. 2, 3, 27, 29, 30, 31, 32, 36, 79, 95, 96, 99, 100

data warehouse Is a central collection of data for reporting and analysis and is considered a core component of business intelligence. 29

dictionary The dictionary table in the data models mapping tid to actual term strings. 33, 34, 36, 39, 40, 44, 52, 66, 67, 85

dictionary history The history table allots a df-value to a given time frame in VCSM3. 39, 44, 46, 52, 58, 64, 67, 68, 82, 83, 86, 90, 99

documents The documents table is keeping meta-information about the document. 33, 36, 39, 40, 42, 48, 52, 60, 61, 67, 71

DOI A digital object identifier (DOI) is a type of persistent identifier used to uniquely identify objects. The DOI system is particularly used for electronic documents such as journal articles. Metadata about the object is stored in association with

[1]`https://en.wikipedia.org/wiki/Binary_Independence_Model`

the DOI name. It may include a location, such as a URL, indicating where the object can be found. The DOI for a document remains fixed over the lifetime of the document, whereas its location and other metadata may change. Referring to an online document by its DOI provides more stable linking than simply using its URL, because if its URL changes, the publisher only needs to update the metadata for the DOI to link to the new URL[2] . 24

Information Entropy A measure for the information content of a message. 9

MAL The primary textual interface to the Monetdb kernel is a simple, assembly-like language, called MAL. The language reflects the virtual machine architecture around the kernel libraries and has been designed for speed of parsing, ease of analysis, and ease of target compilation by query compilers. The language is not meant as a primary programming language, or scripting language.[3] . 30, 94

Out of Memory Killer For Unix-OS a configured Out Of Memory management monitors the system health. When the system runs out of memory, tasks overstressing defined boundaries on heap or CPU-time are likely to be selected by the OOM-killer to be killed.. 95

row store Is a conventional database management system that organizes records an a row basis. 29, 30, 36, 93

scalable data citation Proposed methodology to make data citable on large dynamic databases.. vii, 26, 27, 39, 99

terms The terms table establish a link between the dictionary item and the document.. 33, 36, 39, 40, 42, 43, 44, 46, 48, 58, 60, 61, 67, 70, 83, 85

tf-idf In information retrieval, tf-idf, short for term frequency-inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. It is often used as a weighting factor in information retrieval and text mining. The tf-idf value increases proportionally to the number of times a word appears in the document, but is offset by the frequency of the word in the corpus, which helps to adjust for the fact that some words appear more frequently in general.[4] . vii, 10, 13, 39

versions The versions table keeps track of system updates and manages the vid in VCSM6. 39, 49, 50, 52

[2]https://sw.wikipedia.org/wiki/Digital_object_identifier
[3]https://www.monetdb.org/Documentation/Manuals/MonetDB/MALreference
[4]https://en.wikipedia.org/wiki/Tf%E2%80%93idf

warm run In the domain of c-stores the database and the OS-kernel collect information about the structure of stored data in caches. Consecutive queries run against the system, so-called warm-runs or hot-runs, benefit from caches yielding faster response times.. 86, 87

Acronyms

BAT Binary Association Table. 30, 32, 36, 87, 94

BL Baseline Model. xii, 77, 78, 79, 83, 84, 96

BM25 Best Match Model Nr. 25. 3, 12, 13, 16, 35, 36, 41, 42, 52, 66, 84, 100

CDE Code, Data, and Environment. 21

df document frequency. 10, 11, 18, 33, 36, 39, 40, 41, 42, 44, 46, 52, 56, 58, 60, 61, 64, 66, 67, 68, 70, 76, 79, 82, 83, 86, 89, 90, 91

did document ID. 9, 10, 33, 36, 40, 49, 50, 51, 60, 61, 66, 68, 75, 76, 82, 86, 92

DML Data Manipulation Language. 25, 36, 71, 77, 92, 95

DSM Decomposition Storage Model. 29, 30

FORCE11 Future of Research Communications and e-Scholarship. 23

IDF inverse document frequency. 10, 11, 56

IR Information Retrieval. 1, 2, 3, 5, 6, 7, 9, 11, 12, 13, 16, 17, 18, 21, 22, 27, 29, 32, 33, 36, 39, 41, 52, 70, 73, 77, 99, 100

MMU Memory Management Unit. 30

NSM n-ary storage model. 30

oid object identifier. 30

OLAP Online Analytical Processing. 29, 32, 36

OLTP Online Transaction Processing. 29, 36

PAIR Patent Application Information Retrieval. 2

PID Persistent Identifier. 24, 25, 26

PK primary key. 25, 26, 50

PRF Probability Relevance Framework. 12

RA Registration Authority. 24

RDA Research Data Alliance. 25

RDBMS Relational Database Management System. 5, 25, 27, 30, 31, 32, 33, 36, 66, 92, 95, 99

revid revision ID. 49, 75, 76, 86, 90

SIGMOD Special Interest Group Management of Data. 21

tf term frequency. 10, 11, 36, 39, 42, 43, 44, 52, 56, 58

tid term ID. 33, 36, 40, 42, 44, 46, 47, 58, 60, 61, 66, 77, 79, 82, 83, 85, 86, 92, 93, 95

TREC Text REtrieval Conference. 22

VCSM1 Versioned Column-Store Model 1. x, xii, xiii, 40, 52, 58, 64, 66, 77, 78, 79, 81, 82, 83, 84, 85, 86, 87, 89, 91, 96, 99

VCSM2 Versioned Column-Store Model 2. x, 42, 50, 52, 58, 66, 68, 70, 82, 83, 86, 87, 91, 93, 94, 96, 99

VCSM3 Versioned Column-Store Model 3. x, xiii, 44, 46, 52, 58, 60, 64, 67, 79, 82, 83, 86, 87, 88, 89, 90, 91, 99

VCSM4 Versioned Column-Store Model 4. x, xiii, 46, 60, 70, 71, 79, 82, 83, 86, 89, 90, 99

VCSM5 Versioned Column-Store Model 5. x, xiii, 47, 63, 71, 79, 82, 83, 86, 89, 90, 99

VCSM6 Versioned Column-Store Model 6. x, xiii, 39, 49, 64, 71, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 96, 99

vid version ID. 26, 49, 50, 51, 52, 63, 64, 71, 85, 86

VLDB Very Large Databases. 21