Annotative Indexing

Charles L. A. Clarke   CLACLARK@GMAIL.COM
University of Waterloo
Canada

Editor: Ismail Sengor Altingovde

Abstract

This paper introduces annotative indexing, a novel framework that unifies and generalizes traditional inverted indexes, column stores, object stores, and graph databases. As a result, annotative indexing can provide the underlying indexing framework for databases that support retrieval augmented generation, knowledge graphs, entity retrieval, semi-structured data, and ranked retrieval. While we primarily focus on human language data in the form of text, annotative indexing is sufficiently general to support a range of other data types, and we provide examples of SQL-like queries over a JSON store that includes numbers and dates. Taking advantage of the flexibility of annotative indexing, we also demonstrate a fully dynamic annotative index incorporating support for ACID properties of transactions with hundreds of multiple concurrent readers and writers.

Keywords: Search, Indexing, Inverted Indexes, Minimal-interval Semantics

1 Introduction

Until recently, and with few exceptions, an inverted index provided the foundational file structure for an information retrieval system. Over the years, research progress on file structures for information retrieval was primarily driven by the need to make traditional first-stage sparse retrieval methods (e.g., BM25) as fast as possible, while minimizing storage and memory requirements, motivating the development of specialized processing methods (e.g., WAND) and compression methods (e.g., vByte). To a large extent, this research views an inverted index as single-purpose file structure, with the sole task of delivering the top-k items from a large collection to a second-stage re-ranker with high throughput and low latency. More recently, vector databases supporting dense retrieval have begun to replace inverted indexes, but the focus remains on the efficiency and effectiveness of first-stage retrieval.

Managing large collections of human language data requires more than just a single-minded focus on first-stage retrieval. For example, guidelines for the TREC 2024 RAG Track[1] describe the preparation of a segmented version of the MS MACRO V2 passage corpus for use by track participants. Processing steps include the identification and elimination of duplicate passages to avoid holes and inconsistencies in evaluation. The original corpus was segmented with "a sliding window size of 10 sentences and a stride of 5 sentences" to make it "more manageable for users and baselines." The original corpus and

---

1. `https://trec-rag.github.io/`

its de-duplicated/segmented version are distributed as two independent sets of compressed JSONL files, linked to each other only by a naming convention for document identifiers.
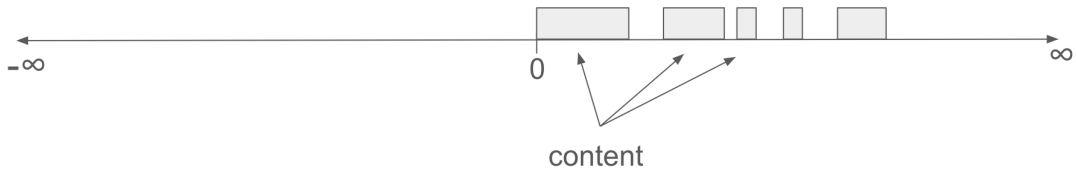
In general, collections of human language data employ a variety of text formats, including JSON, JSONL, TSV, CSV, HTML, CBOR, LaTeX, Word, and PDF. Even source code, such as Python and C++, can be considered as a form of human language data. Processing text collections involves transformations such as tokenization, sentence/word splitting, de-duplication, tagging, and entity linking, as well as generating and storing weights for sparse retrieval and vectors for dense retrieval. Tools for these tasks range from record-at-a-time processing in notebooks to storage in a variety of database systems, including relational databases, search engines, object stores, and knowledge graphs. No single tool allows us to flexibly store, transform, and search multi-format heterogeneous collections of unstructured and semi-structured human language data.

This paper introduces *annotative indexing*, a novel framework that unifies and generalizes traditional inverted indexes, column stores, object stores, and graph databases. As a result, annotative indexing can provide the underlying indexing framework for databases that support retrieval augmented generation, knowledge graphs, entity retrieval, semi-structured data, and ranked retrieval. While we primarily focus on human language data in the form of text, annotative indexing is sufficiently general to support a range of other data types. Annotative indexing facilitates dynamic update, which in turn facilitates text processing pipelines that perform de-duplication, segmentation, and similar operations, expressing these operations by annotating the source text, rather than generating new text.

The next section (Section 2) presents the fundamentals of annotative indexing, providing a foundation for the remainder of the paper. Section 3 places annotative indexing in the context of prior work. Section 4 then presents the overall organization of an annotative index. As a proof of concept, the section also describes the architecture of our reference implementation, called Cottontail[2]. All experimental results in the paper were generated with this reference implementation. The design of the reference implementation reflects the relative simplicity of an annotative index, with a small number of generic components that can be specialized and combined to support different applications.

Section 5 discusses query processing, including an example of a JSON store built on Cottontail, which supports structural containment, Boolean expressions, and similar operations, along with numbers and unified support for dates in differing formats. Section 6 discusses support for dynamic update and transactions, including support for ACID properties. As an example, Section 6 presents a dynamically evolving collection that recapitulates the early years of TREC experiments, with dozens of concurrent writers and hundreds of concurrent readers. Annotative indexing and dynamic update complement and support each other. Dynamic update of traditional inverted indexes is generally limited to adding and deleting entire documents, with no or limited support for concurrent update and transactions. Annotative indexing provides the ability to annotate content after it has been added, enabling richer and more flexible update operations — which, in turn, requires transactional support to ensure concurrency among multiple readers and writers.

---

2. Code for the reference implementation is available at `https://github.com/claclark/Cottontail`. Following past practice in the information retrieval community, the reference implementation is named after an animal, in this case the eastern cottontail, which is the most common species of rabbit in North America. The author often encounters them out and about near the University of Waterloo.

$\mathcal{X}(592856130, 592856138) \Rightarrow$ `To be or not to be, | that is the`

$\mathcal{X}(17905274055, 17905274393) \Rightarrow$

```
{ "docid": "msmarco_v2.1_doc_29_677149#3_1637632" , "end_char": 3061 ,
"headings": "Aeolian Vibration of Transmission Conductors Aeolian
Vibration of Transmission Conductors What is Aeolian Vibration?
Wind causes a variety of motions on transmission line conductors.
Important among them are How Aeolian Vibration Occurs?  Theory/Mechanism...
...that creates an alternating pressure imbalance causing the conductor to
move up and down at a ninety-degree angle to the flow direction." ,
"start_char": 1806 , "title": "Aeolian Vibration of Transmission Conductors",
"url": "https://studyelectrical.com/2019/07/aeolian-vibration-..." }
```

Figure 1: The content of an associative index is situated in an address space, which may contain gaps, where content has been deleted. A translation function $\mathcal{X}(p, q)$ maps an interval in the address space to its associated content. The figure shows examples from an index containing the segmented version of the MS MARCO V2.1 Document Corpus as used by the TREC 2023 RAG Track. In this example, tokenization is word based. At the content level, a JSON object is represented as a sequence of tokens, with special tokens representing JSON structural tokens (`"`, `:`, etc.). Annotations on top of the content define structural elements. For example, the annotation $\langle$`:title:`$, (17905274368, 17905274374)\rangle$ indicates the interval containing the title in this object.

2 Fundamentals of Annotative Indexing

An annotative index stores human language data as its *content* plus a set of *annotations* describing that content. The content is represented by a sequence of *tokens*, where each token is assigned an integer location in an *address space*, as illustrated in Figure 1. If content has been deleted, gaps are possible. By convention, our reference implementation appends content at increasing addresses, starting at zero. However, negative addresses are supported for mathematical simplicity and consistency (see example below). As shown in Figure 1, a translation function $\mathcal{X}(p, q)$ maps an interval in the address space to the associated content. $\mathcal{X}(p, q)$ is undefined if (p, q) contains a gap. For content addressing purposes, tokenization can be flexibly defined at the word or character level. Separate and distinct tokenization and stemming can be also employed for specific applications, e.g. ranking.

Annotations provide information about intervals over the content. An annotation is a triple $\langle f, (p, q), v \rangle$, where f is a *feature*, (p, q) is the interval over which the annotation applies, and v is the value of the feature over that interval, which defaults to 0. For

convenience we define:

$$\langle f, p, v \rangle \quad = \quad \langle f, (p,p), v \rangle \tag{1}$$
$$\langle f, (p,q) \rangle \quad = \quad \langle f, (p,q), 0 \rangle \tag{2}$$
$$\langle f, p \rangle \quad = \quad \langle f, (p,p), 0 \rangle \tag{3}$$

For example, the annotation:

$$\langle \texttt{start\_char:}, (17905274359, 17905274362), 1806 \rangle$$

indicates that over the interval $(17905274359, 17905274362)$ the feature `start_char:` has the value 1806, as shown in Figure 1. The annotation:

$$\langle \texttt{tf:porter:aeolian}, 17905274055, 17 \rangle$$

indicates that the Porter-stemmed term "`aeolian`" appears 17 times in the JSON object starting at address 17905274055. The annotation:

$$\langle \texttt{:}, (17905274055, 17905274393) \rangle$$

indicates that the interval $(17905274055, 17905274393)$ contains a JSON object, as represented by the feature "`:`". The annotation:

$$\langle \texttt{aeolian}, 17905274369 \rangle$$

indicates that the word "`aeolian`" appears at that address. We can use annotations like these to implement BM25 ranking on a JSON store, but the annotative index itself merely stores the content and its associated annotations. The interpretation of the annotations as term and document statistics is left to the ranking algorithm.

Annotations are indexed by feature, with two access methods (τ and ρ) that both take an address k in the address space and return the first annotation greater than or equal to k, according to the start or end address of the interval.

$$f.\tau(k) \ = \ \{\langle (p,q), v \rangle, \text{ where } \langle f, (p,q), v \rangle \text{ is the annotation for } f \text{ with minimal } p \geq k\} \tag{4}$$
$$f.\rho(k) \ = \ \{\langle (p,q), v \rangle, \text{ where } \langle f, (p,q), v \rangle \text{ is the annotation for } f \text{ with minimal } q \geq k\} \tag{5}$$

To simplify index organization and facilitate index processing, the set of annotations for a feature must follow *minimal-interval semantics* as defined in prior work, including Boldi and Vigna (2016, 2018); Clarke and Cormack (2000), Clarke (1996), and Clarke et al. (1995a). Minimal interval semantics requires that no annotation for the same feature can be contained in another, but they can overlap. If $\langle f, (p,q), v \rangle$ and $\langle f, (p',q'), v' \rangle$ are annotations for feature f, then either $p < p'$ and $q < q'$, or $p > p'$ and $q > q'$. The annotations for f are thus totally ordered — in the same order — by their start and end addresses. For mathematical simplicity and consistency, we consider every feature f to have the annotations $\langle f, (-\infty, -\infty), 0 \rangle$ and $\langle f, (\infty, \infty), 0 \rangle$. Boldi and Vigna (2018) describe a set of intervals under minimal-interval semantics as an element of a "Clarke–Cormack–Burkowski lattice". Clarke, Cormack, and Burkowski (1995) themselves call it a "generalized concordance list". In this paper, the term "annotation list" implies an ordered set of annotations under minimal-interval semantics.

Prior work on minimal-interval semantics demonstrated their practical value as a method for expressing queries over heterogeneous collections of semi-structured data, providing efficient support for containment, boolean, merge, proximity, ordering, and other structural operators. This paper extends this prior work in two ways, which together substantially increase expressive power. First, while prior work treated singleton intervals as the only atomic unit for indexing purposes, we index intervals of any length. For example, by indexing intervals we can run a sentence splitter over the content and add annotations to the index indicating sentence boundaries. Second, we associate a value with each interval, which is preserved by containment and merge operations. For example, we can compute terms statistics over the content and add annotations to support ranked retrieval. In this work, the atomic unit for indexing is an annotation, comprising a feature, an interval and a value. Operators combine annotations lists to produce annotation lists.

At its core, an annotative index is just a set of features with the τ and ρ methods of Equations 4 and 5, restricted by minimal interval semantics and as supported by the translation function $\mathcal{X}$. The generality of annotative indexing lies in the flexibility of what the features represent, which can include term statistics, document structure, and links. Since implementation details are hidden behind τ, ρ, and $\mathcal{X}$, an annotative index can implemented with a wide variety of data structures. For example, Cottontail provides both static and dynamic index formats that employ (nearly) distinct storage structures.

Clarke et al. (1995a) and Clarke (1996) contain many examples illustrating queries and query processing. Suppose we want to find all objects (":") with "`aeolian`" in their "`:title:`". the τ and ρ methods work together to provide an efficient solution.

```
1  k ← 0
2  S ← { }
3  while k ≠ ∞:
4      ⟨(p, q), v⟩ ← :title:.ρ(k)
5      ⟨(p', q'), v'⟩ ← aeolian.τ(p)
6      if q' ≠ ∞ and q' ≤ q:
7          S ← S ∪ {:.ρ(q)}
8          k ← q + 1
9      else:
10         k ← q'
```

After execution, the set $\mathcal{S}$ contains a set of annotations corresponding to the required objects, which is itself an annotation list. Lines 4 and 5 generate the next candidate title and occurrence of "aeolian". Line 6 determines if the candidate title contains "aeolian", and if so, we add the associated object to $\mathcal{S}$. Line 10 is the key to efficiency, setting k to skip titles that can't contain the next candidate "aeolian". If "aeolian" is relatively rare, and clustered in a relatively small number of objects, we may be able to avoid considering most titles. For simplicity, this solution assumes that all titles are contained in an object, which may not be true if the context mixes different types of text. As shown later, we would not normally materialize $\mathcal{S}$ as a set, but rather would provide "lazy" access through $\mathcal{S}.\tau$ and $\mathcal{S}.\rho$ implemented in terms of τ and ρ for the ":", "`aeolian`", and "`:title:`" features.

As another example, we can define access methods for a window of $n > 0$ tokens as $\#n.\tau(k) \equiv \langle(k, k + n - 1), 0\rangle$ and $\#n.\rho(k) \equiv \langle(k - n + 1, k), 0\rangle$. In the example above, replacing "aeolian" with the feature $\#12$ generates the set of objects with titles at least 12 tokens long. While titles can no longer be skipped, windows are generated "as needed" to test against each title. Note that $\#12.\rho(0) \equiv \langle(-11, 0), 0\rangle$, providing a simple example of a negative address. Section 3.3 provides additional background on minimal interval semantics, including additional discussion regarding operators and query processing

3 Comparison with Prior Work

3.1 Static and Dynamic Inverted Indexes

Annotative indexes generalize inverted indexes. Büttcher et al. (2010) provides a review of inverted index file structures and associated query processing methods that remains reasonably current. They describe the core techniques that are still widely employed, along with experimental comparisons against competing techniques. A generic inverted index maps each term in a *vocabulary* — maintained in by a *dictionary* — to a *postings list* of *document identifiers* where the term appears. Postings lists often include term frequencies to support ranking formulae and term offsets to support phrase searching. Postings lists are typically gap-encoded and compressed with a method such as vByte (Williams and Zobel, 1999), which usually provides an acceptable trade off between compression ratio and decompression speed. Since query processing methods can often skip documents, synchronization points may be included in the compressed posting lists to improve performance and reduce the need for decompression (Moffat and Zobel, 1996).

Academic research on inverted indexes often views them through the lens of a static file structure, built once from a collection and never changed (e.g., Arroyuelo et al. (2018); Mallia et al. (2019); Mackenzie and Moffat (2020)). If the collection changes, the index is re-built from scratch. For example, if a researcher wants to remove near-duplicates from a collection because they are causing problems with their retrieval experiments, the researcher first filters the collection and then builds a new index for the filtered collection. A complete index re-build can be slow, even for a relatively small collection. At the very least, a complete re-build requires an end-to-end read of the collection to construct postings lists, so that the build time grows linearly with the size of the collection.

Prior research has considered a variety of dynamic update models for inverted indexes (Büttcher et al., 2010). The simplest model provides for *batch updates*, which build index structures for new documents and merge them into the original index without requiring a complete rebuild. During the merge, the index also deletes any unneeded documents. Ideally, the overall process is managed as a transaction, so that a failure during the update process does not corrupt the index structures. The batch update model supports only one transaction at a time. Starting a second update during a transaction either produces an error or blocks until the current transaction completes. If the index is queried during a transaction, atomicity should guarantee that a result over the original index is returned. While a batch update avoids some of the work required by a full rebuild, the index cannot evolve quickly. Depending on the final size of the index, it might take minutes or hours for a change to become visible to queries.

Under the *immediate-access* dynamic update model, changes become visible as soon as they are made (Büttcher et al., 2010). Social media search provides an important use case for immediate-access dynamic update. Asadi et al. (2013) describe index update in the EarlyBird search engine, developed for Twitter. EarlyBird search was designed for a single, high-volume update stream, with many concurrent readers and a strong temporal ranking signal, placing a high priority on recent tweets. Once a tweet is indexed, its indexing does not change. Moffat and Mackenzie (2023) explore trade-offs between insertion speed, query speed, and index size in an immediate-access dynamic index. They describe in-memory indexing structures that supports a stream of interleaved queries and document insertions. Document deletion is not supported, so that index grows with each insertion. To maintain a consistent view of the index, they assume that "all postings associated with each ingested document are processed into the index before the next query operation is permitted", effectively requiring a read-lock on the index during each insertion. Eades et al. (2022) describe index structures for dynamic update that uses fixed volume of memory, with older documents expiring from the index.

Prior research on immediate-access dynamic update does not satisfy the requirements of annotative indexing. In particular, prior work assumes that indexing for a document happens all at once; no additional indexing for a document can be added at a later time (Büttcher et al., 2010). In contrast, annotative indexing enables novel use cases that require additional indexing. For example, imagine an annotative index supporting a document ingestion pipeline for a retrieval augmented generation (RAG) system that includes de-duplication, segmentation, and indexing stages. Each stage reads its input from the index and records its output by adding annotations. For an annotative index to fully support a document processing pipeline, the output from a stage must be immediately visible as soon as the stage finishes. However, each stage must see a complete and consistent view of the output from the previous stages. When stages are independent of each other, it should be possible for them to run concurrently. Since some stages may require considerable processing, it is important for updates to be durable, allowing the pipeline to recover quickly from a failure. To satisfy this scenario and realize the full benefits of annotative indexing, an annotative index must support concurrent access and ensure ACID properties of transactions, requirements that are not met by prior research.

3.2 First-Stage Retrieval

Over 30 years after its invention, the BM25 formula remains the touchstone for unsupervised first-stage retrieval (Robertson and Walker, 1994; Robertson et al., 1994). When compared to other unsupervised retrieval formulae from the 1990s and early 2000s — which may provide as-good-or-better retrieval effectiveness — BM25 exhibits term saturation properties that can be exploited to substantially improve query performance through WAND query processing (Broder et al., 2003; Petri et al., 2013; Turtle and Flood, 1995) and Block-Max WAND processing (Dimopoulos et al., 2013; Ding and Suel, 2011). Term saturation guarantees an upper bound on the weight given to any single query term, allowing us to skip documents whose score cannot exceed a threshold defined by scores of the current top-k documents.

In their description of standard WAND processing, Petri et al. (2013) assume posting lists will be accessed through three functions: 1) A *first* function, which creates an iterator for the list, 2) a *next* function, which advances the iterator by one posting, and 3) a *seek(d)* function, which advances iterator to the first document identifier greater than or equal to d. The τ and ρ operations in Equation 4 and Equation 5 generalize the *seek* function to intervals. When wrapped in an appropriate iterator, and with appropriate annotations, they directly support WAND processing over annotative indexes.

Using the τ and ρ operations, Cottontail can efficiently implement WAND processing. As a demonstration of performance we use the 6980 "dev small" queries and passages from the original MS MARCO test collection (Bajaj et al., 2018). The corpus comprises 8,841,823 passages with an uncompressed size of 2.85GB. Full cottontail indexing with BM25 annotations using the static index implementation described in Section 4 gives a compressed index of 4.4GB. In addition to the annotations required for BM25 ranking, this index includes token-level annotations, to implement phrase search and structural queries, along with the full text of the collection, to implement the $\mathcal{X}$ translation function.

With BM25 parameters $b = 0.68$ and $k_1 = 0.82$ we achieve an MRR@10 of 0.185. These BM25 parameters are recommended by the Anserini onboarding guide, which uses this collection as an example for teaching indexing and retrieval[3]. MMR@10 is the standard precision metric for this test collection. Running with two threads per physical core, i.e., 46 concurrent queries, it requires less than 20 seconds to rank all queries to a depth of 10, giving a throughput of over 350 queries/second[4]. Running one query at a time gives an average query latency of 65ms. In comparison, using a system based on the Lucene search library, Lin et al. (2020) report a BM25 latency of 55ms and MMR@10 of 0.184 on the same collection. While Lin et al. (2020) do not indicate the hardware used for their measurements, nor do they report query throughput, this comparison suggests that our general indexing framework can be reasonably competitive with a specialized index developed through years of engineering effort.

In recent years, neural retrieval methods have eclipsed traditional unsupervised methods for first-stage retrieval. Neural first-stage retrieval methods fall into two camps: *sparse vector retrieval* and *dense vector retrieval*. Sparse vector retrieval methods represent queries and documents in a high-dimensional space, where each dimension corresponds to a token (Lin and Ma, 2021; Song et al., 2021) and most weights are zero, especially in query vectors. Sparsity allows these vectors to be stored in an inverted index; ranking requires only a dot product between the query and document vectors. Successful approaches to sparse retrieval include DeepCT (Dai and Callan, 2019), HDCT (Dai and Callan, 2020), uniCOIL (Lin and Ma, 2021) and SPLADE (Formal et al., 2021). In particular, SPLADE is widely recognized for its retrieval effectiveness (Lassance et al., 2024; Mallia et al., 2024; Bruch et al., 2024). Despite a few proposals for unsupervised neural sparse methods (e.g., Ma et al. (2023)), neural sparse methods are often called "learned sparse retrieval" methods to distinguish them from traditional unsupervised sparse methods, such as BM25.

Annotative indexing trivially supports learned sparse retrieval by creating an annotation for each element of a sparse vector. It is also trivial to support multiple sparse retrieval

---

3. `https://github.com/castorini/anserini/blob/master/docs/experiments-msmarco-passage.md`
4. All experiments reported in this paper were conducted on a Intel(R) Xeon(R) Gold 5120 CPU with 256GB of memory.

methods (e.g. BM25 and SPLADE) in the same index, or to use different ranking approaches at different structural levels (e.g. BM25 at the document level and SPLADE at the passage level) . Unfortunately, learned weights do not provide the distributional properties that algorithms like WAND exploit to improve query performance. Score-at-a-time ranking approaches can partly address this problem (Mackenzie et al., 2021). It may also be possible to adapt block pruning and other methods to annotative indexes, for example, by adding additional annotations summarizing weights over blocks of documents (Mallia et al., 2024; Bruch et al., 2024; Mallia et al., 2017; Ding and Suel, 2011).

Dense vector retrieval is currently the focus of intense research, with multiple recent surveys available (Pan et al., 2024; Zhao et al., 2024). The simplest form of dense retrieval, *bi-encoder retrieval*, represents queries and documents in a low-dimensional space (e.g., 768 dimensions) where the values in most dimensions are non-zero. Ranking requires only a dot product between the query and document vectors (Reimers and Gurevych, 2019; Karpukhin et al., 2020; Zhan et al., 2020). Various approximate k-nearest neighbor search methods can speed the ranking process. For example, Hierarchical Navigable Small World (HNSW) graphs arrange vectors in a hierarchy of proximity graphs that can be traversed in approximately logarithmic time (Malkov and Yashunin, 2020).

While each dimension could be represented as an annotation list, retrieval would be inefficient because of the relatively large number of non-zero values in a dense vector. To support dense vectors, we would need to extend annotative indexing with a vector store, which might map locations in the address space to vectors. Annotative indexes can also help support hybrid approaches that combine sparse and dense retrieval (Leonhardt et al., 2022). It may also be possible to encode and efficiently search HNSW graphs encoded as annotations.

3.3 Minimal Interval Semantics

Minimal-interval semantics were invented by the author for his Ph.D thesis nearly 30 years ago (Clarke, 1996). If we view the result of a text search over a string as a set of substrings that satisfy the requirements of the search, minimal-interval semantics provide a simple and natural way to linearize the set, as well as enabling fast and flexible algorithms for combining and filtering search results. If we specify the set of substrings S as a set of intervals (p, q), minimal interval semantics allows these intervals to overlap but not to nest.

An interval (p, q) *overlaps* an interval (p', q') if either $p' \leq p \leq q'$ or $p' \leq q \leq q'$, but not both. An interval (p, q) is *nested* in an interval (p', q') if $(p, q) \neq (p', q')$ and $p' \leq p \leq q \leq q'$. If $a = (p, q)$ and $b = (p', q')$ are intervals, the notation $a \sqsubset b$ indicates that a nests in b; the notation $a \sqsubseteq b$ indicates that a is *contained in* b: that either a and b are equal or that a nests in b. Intervals form a partial order under $\sqsubseteq$.

We formalize the reduction of a set of intervals S to a *generalized concordance list* as a function $\mathcal{G}(S)$:

$$\mathcal{G}(S) = \{a \mid a \in S \text{ and } \nexists\, b \in S \text{ such that } b \sqsubset a\}$$

A set S is a generalized concordance list if and only if $S = \mathcal{G}(S)$. Each interval in a generalized concordance list acts as a "witness" to the satisfiability of the requirements of the search (Boldi and Vigna, 2018). As a simple example, consider the query:

```
"peanut butter"  △ "jelly doughnut",
```

117

Containment Operators

    Contained In:

$$A \lhd B = \{a \mid a \in A \text{ and } \exists\, b \in B \text{ such that } a \sqsubseteq b\}$$

    Containing:

$$A \rhd B = \{a \mid a \in A \text{ and } \exists\, b \in B \text{ such that } b \sqsubseteq a\}$$

    Not Contained In:

$$A \ntriangleleft B = \{a \mid a \in A \text{ and } \nexists\, b \in B \text{ such that } a \sqsubseteq b\}$$

    Not Containing:

$$A \ntriangleright B = \{a \mid a \in A \text{ and } \nexists\, b \in B \text{ such that } b \sqsubseteq a\}$$

Combination Operators

    Both Of:

$$A \triangle B = \mathcal{G}\left(\{c \mid \exists\, a \in A \text{ such that } a \sqsubseteq c \text{ and } \exists\, b \in B \text{ such that } b \sqsubseteq c\}\right)$$

    One Of:

$$A \triangledown B = \mathcal{G}\left(\{c \mid \exists\, a \in A \text{ such that } a \sqsubseteq c \text{ or } \exists\, b \in B \text{ such that } b \sqsubseteq c\}\right)$$

    Follows:

$$A \lozenge B = \mathcal{G}\left(\{c \mid \ \exists\, (p,q) \in A \text{ and } \exists\, (p',q') \in B \text{ where } q < p' \text{ and } (p,q') \sqsubseteq c\}\right)$$

Figure 2: Fundamental operators for expressing structural relationships over generalized concordance lists, which underlie annotation lists. A and B can be any generalized concordance lists, including subqueries built from these operators.

where "$\triangle$" indicates Boolean conjunction. If we view the set of intervals satisfying the query `"peanut butter"` as the set of all intervals containing that string, of any length, then $\mathcal{G}\,($`"peanut butter"`$)$ is just the set of intervals corresponding to the string itself. If we view the set of intervals satisfying the conjunction as the set of intervals that contain both strings, then the set of minimal intervals that contain both strings is $\mathcal{G}\,($`"peanut butter"` $\triangle$ `"jelly doughnut"`$)$, which may overlap but not nest. For example, the sentence:

        Peanut butter on a jelly doughnut is better than a peanut butter sandwich.

contains two overlapping intervals which satisfy the conjunction under minimal interval semantics.

    Figure 2 summarizes fundamental operators from Clarke (1996), where A and B are generalized concordance lists, The operators fall into two groups, *containment* and *combination*, which together support a wide range of queries specifying structural relationships. For example, the query for all objects ("`:`") with "`aeolian`" in their "`:title:`" is:

$$: \rhd (\texttt{:title:} \rhd \texttt{aeolian})$$

Generalized concordance lists have the same access methods as annotation lists, but without associated values. They are just ordered sets of intervals under mimimal interval semantics.

If S is a generalized concordance list, then:

$$S.\tau(k) \quad = \quad \{(p, q), \text{ where } (p, q) \text{ is the interval in } S \text{ with minimal } p \geq k\} \qquad (6)$$

$$S.\rho(k) \quad = \quad \{(p, q), \text{ where } (p, q) \text{ is the interval in } S \text{ with minimal } q \geq k\} \qquad (7)$$

A key observation of Clarke (1996) — echoed by Boldi and Vigna (2016) — is that evaluation can be "lazy". Much like WAND processing, lazy evaluation allows us to skip solutions to subqueries that cannot lead to a solution for the overall query. If A and B are generalized concordance lists, we can implement τ and ρ access methods for each operator of Figure 2 in terms of τ and ρ for A and B, which in turn can subqueries built from these operators. For example, the ρ operator for $A \vartriangleright B$ can be written as:

```
1  (A ▷ B).ρ(k) ≡
2    (p, q) ← A.ρ(k)
3    (p′, q′) ← B.τ(p)
4    if q′ ≤ q:
5      return (p, q)
6    else:
7      return (A ▷ B).ρ(q′)
```

If A corresponds to "`:title:`" and B corresponds to "`aeolian`", we obtain a ρ access method for titles containing "aeolian", which can in turn be combined with the generalized concordance list for objects to give a ρ access method for objects containing these titles. Compare this definition to the algorithm on page 113.

Unfortunately, there is no single summary of key theorems and algorithms for operations under minimal interval semantics. Clarke et al. (1995b) contains an overview with many examples. Clarke et al. (1995a) contains additional examples and algorithms for implementing the operators, which are extended with proofs by Clarke (1996). While τ and ρ are sufficient to implement the operators of Figure 2, Clarke (1996) also defines "backwards" version of these access methods that facilitate solutions that start with the last interval, which can be valuable in finding the most-recent solutions to queries over a growing index (Asadi et al., 2013). More recently, Boldi and Vigna (2016, 2018) present a mathematical foundation for minimal-interval semantics based on lattices.

Clarke and Cormack (2000) present an improved implementation framework for the combinational operators, including Boolean operators. Under this framework, finding all solutions to a combinational query with n terms requires no more than $O(n \cdot \mathcal{A})$ calls to access methods for the terms, where $\mathcal{A}$ is the number of **solutions** to the query. Using *galloping search* (Büttcher et al., 2010, pp. 42–44) to implement the access methods for the terms gives an overall time complexity of $O(n \cdot \mathcal{A} \cdot \log(L/\mathcal{A}))$, where L is the length of the longest posting list for a term. The overall time complexity is nearly linear in the number of solutions, rather than the length of any postings list, as might be expected. If there are few solutions, most of the postings lists might be skipped. The reference implementation for annotative indexing, Cottontail, captures most of the cumulative insights from research on minimal interval semantics[5].

---

5. `https://github.com/claclark/Cottontail/blob/main/src/gcl.cc`

3.4 Column Stores

A column store is a physical design strategy for relational database systems that partitions data primarily by column, rather than by row. As opposed to a traditional row-oriented strategy for physical database design, a column-oriented strategy is known to provide better performance on data analytics and other read-intensive workloads. Among other properties, grouping together values from a single column can improve compression because values within the same column are often similar or repetitive. This homogeneity makes it easier to apply compression techniques that exploit redundancy. The popularity of column stores grew from the success of systems such as C-Store (Stonebraker et al., 2005) and MonetDB (Idreos et al., 2012). Currently, open formats such as ORC and Parquet enable support for columnar storage on most platforms for data analytics.

Inverted indexes are close cousins of column stores (Mühleisen et al., 2014). If we consider the terms in the vocabulary as columns of a table whose rows represent documents, then we can imagine the table as containing term weights, perhaps with a special column containing document identifiers. Since most terms appear only in a relatively small number of documents, the table is sparse. Most of the entries are NULL. Computing a retrieval formula requires an aggregation over the columns corresponding to terms in the query. Since an inverted index organizes this table by column, only query columns need to be accessed to compute the retrieval formula.

Annotative indexes generalize inverted indexes to something close to a column store. If we store rows of a table as the content of the annotative index and treat the features as columns, then each annotation represents an entry in the table:

$$\langle column, (start, end), value \rangle,$$

where $(start, end)$ is the location of the value in the row. Data can be accessed by column through annotation lists, and by row through the translation function $\mathcal{X}(p, q)$.

3.5 Graph Structures

Many application areas now require database support to store and process large graphs structures (Sahu et al., 2023). For example, Facebook developed the Unicorn search engine to store and search social graph information at worldwide scale (Curtiss et al., 2013). Unicorn's core file structures extend inverted lists with additional information, similar to annotative indexing. However, it does not support minimal-interval semantics. In an information retrieval context, graph data often takes the form of a *knowledge graph*, with Bast et al. (2025) providing a current survey. Knowledge graphs often encode relationships as subject-predicate-object triples: For example the triple:

$$\langle \texttt{Meryl\_Streep} \rangle - \langle \texttt{won\_award} \rangle - \langle \texttt{Best\_Actress} \rangle$$

indicates that Meryl Streep won an Oscar for Best Actress.

An annotation list can encode a directed graph by storing a location in the address space as the value of annotation, so that the annotation $\langle G, p, v \rangle$ is interpreted as a link from an object containing the location p to an object containing the location v. For example, consider the trivial friend graph:

```
{"name": "Alice", "friends": ["Bob", "Carol", "Dave"]}
{"name": "Bob", "friends": ["Alice", "Dave"]}
{"name": "Carol", "friends": ["Alice"]}
{"name": "Dave", "friends": ["Bob", "Alice"]}
```

If the Alice object is stored at $(0, 26)$ and the Bob object is stored at $(27, 49)$, the annotation $\langle \texttt{@friend}, 7, 27 \rangle$ indicates a link from Alice's friends array to Bob, where 7 is the address of the token "Bob" the occurs in Alice's friend array. Using a similar approach, annotations can encode subject-predicate-object triples.

$$\langle predicate, subject, object \rangle.$$

To encode a triple indicating that Meryl Streep won an award for best actress, the *predicate* would be encoded as the feature `won_award`, the *subject* would be an address in the record associated with Streep, and the *object* would be an address in the record associated with the best actress award.

4 Organization of an Annotative Index

In this section, we consider the organization and construction of an annotative index, using our reference implementation, Cottontail, as an example. Cottontail provides two distinct implementations of the index structures, a *static index* and a fully *dynamic index*. The static index supports larger collections, where it may not be possible to maintain the entire collection in memory. The static index reads annotation lists from storage only for query processing and index update; it supports only a single update transaction at a time under the batch update model. The dynamic index maintains all active index structures in memory, while still durably committing transactions to storage. In this section, we focus on the basic index construction process, which applies to both static and dynamic indexes. Section 6 extends this material with details for the fully dynamic index, including support for immediate update and multiple concurrent readers and writers.

An annotative index extends and generalizes an inverted index, as outlined in Section 3.1, annotations are indexed by feature, with annotations ordered by the start address (and equivalently the end address) of their intervals. If the annotations for feature f are

$$a_0 = \langle f, (p_0, q_0), v_0 \rangle, \ a_1 = \langle f, (p_1, q_1), v_1 \rangle, \ a_2 = \langle f, (p_2, q_2), v_2 \rangle, ...$$

then $\forall i, \ p_i < p_{i+1}$ and $q_i < q_{i+1}$. Since they strictly increase, successive start (and end) addresses can be gap-encoded and compressed with vByte, or other methods developed for compressing postings lists. For a given f, if $\forall i, \ p_i = q_i$, then its end addresses can be compressed away. Similar to column stores, values will tend to share distributional properties that can be exploited to improve compression. For a given f, if $\forall i, \ v_i = 0$, its values can be compressed away.

Figure 3 provides an overview of the major components of Cottontail. The various components of an annotative index are grouped into a `Warren`, which manages transactions and simplifies common operations that interact with multiple components[6]. Apart from a

---

6. The author is aware that eastern cottontail rabbits are solitary and don't live in warrens.

`Warren:` Groups the following components and manages transactions.

   Operations: `clone`, `start`, `end`, `transaction`, `ready`, `commit`, `abort`

   `Tokenizer:`Facilitates content addressability (Section 4).
      Operations: `tokenize`, `split`, `skip`

   `Featurizer:` Maps a feature (expressed as a string) to a 64-bit value (Section 4).
      Operations: `featurize`

   `Annotator:` Inserts and deletes annotations (Section 4).
      Operations: `annotate`, `erase`

   `Appender:` Appends text to the content (Section 4).
      Operation: `append`

   `Idx:` Provides read access to annotations (Section 5).
      Operation: `hopper(f)` — create a cursor (called a `Hopper`) for the feature `f`

   `Txt:` Provides read access to content (Section 5).
      Operation: `translate(p, q)` — return content associated with the interval ($\mathcal{T}(p, q)$)

Figure 3: Major components of Cottontail, the reference implementation for annotative indexing. A `Warren` object contains and manages one instance of each of the other components (`Tokenizer`, `Featurizer`, etc.). Cottontail provides multiple versions of each component, each specialized for a different purpose. Section numbers indicate where the component is discussed. The transaction model for a `Warren` is discussed in Section 6.

`Warren`, each component implements no more than three operations. Cottontail provides multiple versions of each component, each specialized for a different purpose, which can be mixed and matched in a `Warren`.

   A `Tokenizer` facilitates content addressability by splitting strings into tokens, computing token boundaries, and skipping tokens. Support for ASCII content with HTML-style tags is provided by `AsciiTokenizer`, which is intended for use with older TREC collections. Generic support for Unicode is provided by `Utf8Tokenizer`, which is intended for use with JSON and other modern content. The role of a `Tokenizer` in a `Warren` is limited to facilitating content addressability. Other tokenization (e.g., language specific or WordPiece) can be used by features in annotations to support ranking and other applications.

   Internally, cottontail represents an annotation as four 64-bit values, using a `Featurizer` to map a feature expressed as a string to a 64-bit value. `HashingFeaturizer` maps strings to 64-bit values with a MurmurHash function. `HashingFeaturizer` can be wrapped by other `Featurizer` classes to record vocabulary items and to exclude selected features

from indexing. By convention, features mapped to 0 are not indexed. For example, the `JsonFeaturizer` wraps any `Featurizer`, and maps to 0 those tokens that represent JSON structural elements, such as the curly braces surrounding objects.

An `Appender` and an `Annotator` work together for index construction and update. Both support two-phase commit protocols, with the overall transaction managed by the `Warren`. An `Appender` appends data to the content through its `append` operation:

$$\texttt{append}(content) \rightarrow (p, q)$$

The `append` operation returns the interval where the appended content is located. An `Annotator` adds an annotation to the index through the `annotate` operation:

$$\texttt{annotate}(f, v, p, q)$$

which adds the annotation $\langle f, (p, q), v \rangle$, where the value v is optional.

Figure 4 shows a partial trace of `append` and `annotate` operations, while adding a nested JSON object to an annotative index. With the help of a fast JSON parser[7], support for a general JSON store requires less than 500 lines of C++ beyond the core generic annotative indexing code. The example object is taken from a set of open source examples available on Adobe's website[8]. The order that JSON key-value pairs are added differs from the textual order in the object because the object is first parsed into a C++ map and then traversed to add the object to the annotative store[9].

In the figure, the operation "`append("batters":)`" appends four tokens: """, "`batters`", """, and "`:`", returning the interval $(1, 4)$. Tokens marking structural elements of the JSON object ("`{`", "`}`", """, "`:`", etc.) are encoded as special tokens using Unicode noncharacters, which are permanently reserved for the internal use by systems that store and transmit text. With this encoding, the `translate` operation of a `Txt` component, which implements $\mathcal{X}(p, q)$, can return any interval of the content and recognize the difference between a "`:`" separating a JSON key-value pair and a "`:`" that happens to appear in a string.

For conciseness, the trace omits `annotate` operations that add annotations for single tokens, which are automatically performed as part of an `append` operation. For example, as part of the "`append("Regular")`" operation, the annotation $\langle \texttt{regular}, 35 \rangle$ is automatically added. As previously mentioned, `JsonFeaturizer` returns 0 for tokens marking structural elements, suppressing automatic annotation to avoid unnecessary indexing.

All structure and nesting is retained in the features. For example, the annotation $\langle \texttt{:batters:batter:[0]:type:}, (24, 26) \rangle$ indicates the "`type`" property of the first element of the "`batter`" array of the "`batters`" property. A JSON object is not "flattened" in any sense. The content (i.e, $\mathcal{X}(0, 254)$) contains the full JSON object, which can be accessed by the `translate` operation of the `txt` component.

By convention, the feature "`:`" is used as the root of the object, as seen in the annotation $\langle \texttt{:}, (0, 254) \rangle$. Individual objects in a collection of JSON objects, e.g. a JSONL file, can be accessed through this '`:`" feature. In the annotation $\langle \texttt{:batters:batter:}, (10, 84), 4 \rangle$ the value 4 gives the length of the array. In a later example, we apply the convention of storing

7. `https://github.com/nlohmann/json`
8. `https://opensource.adobe.com/Spry/samples/data_region/JSONDataSetSample.html`
9. `https://github.com/claclark/Cottontail/blob/main/src/json.cc`

```
{
 "id": "0001",
 "type": "donut",
 "name": "Cake",
 "ppu": 0.55,
 "batters":
  {
   "batter":
    [
      { "id": "1001",
        "type": "Regular"},
      { "id": "1002",
        "type": "Chocolate"},
      { "id": "1003",
        "type": "Blueberry"},
      { "id": "1004",
        "type": "Devil's Food"}
    ]
  },
 "topping":
  [
   { "id": "5001",
     "type": "None"},
   { "id": "5002",
     "type": "Glazed"},
   { "id": "5005",
     "type": "Sugar"},
   { "id": "5007",
     "type": "Powdered Sugar"}
   { "id": "5006",
     "type": "Chocolate with Sprinkles" },
   { "id": "5003",
     "type": "Chocolate" },
   { "id": "5004",
     "type": "Maple" }
  ]
}
```

```
transaction()
append({) → (0, 0)
append("batters":)  → (1, 4)
append({) → (5, 5)
append("batter":)  → (6, 9)
append([) → (10, 10)
append({) → (11, 11)
append("id":)  → (12, 15)
append("1001") → (16, 18)
annotate(:batters:batter:[0]:id:, 16, 18)
append(,) → (19, 19)
append("type":)  → (20, 23)
append("Regular") → (24, 26)
append(}) → (27, 27)
annotate(:batters:batter:[0]:type:, 24, 26)
annotate(:batters:batter:[0]:, 11, 27)
...
annotate (:batters:batter:, 10, 84, 4)
...
append("name":)  → (95, 98)
append("Cake") → (99, 101)
annotate(:name:, 99, 101)
append(,) → (102, 102)
append("ppu":)  → (103, 106)
append("0.5500") → (107, 110)
annotate(:ppu:, 107, 110, 0.55)
...
annotate(:, 0, 254)
ready()
commit()
```

Figure 4: Constructing an annotative index. The inset on the right shows a partial trace of `append` and `annotate` operations during the addition of the JSON object on the left.

the array length as the value for the array feature to step through ("explode") arrays of different lengths in different objects. These conventions, as well as other conventions used to support a JSON store, are independent of the underlying associative index.

5 Query Processing

The τ and ρ access methods, as defined by Equations 4 and 5, provide the foundation for query processing. In Cottontail, the `hopper(f)` operation of the `Idx` component creates a `Hopper` object for the 64-byte feature value `f`. A `Hopper` object acts as a cursor, supporting the τ and ρ access methods over the feature and caching the most recent result from each access method. All accesses to the underlying index structures are abstracted by τ and ρ, which we are then free to implement in any suitable way. For example, the index structures might include synchronization points to allow the `Hopper` to skip annotations (Moffat and Zobel, 1996). The current version of Cottontail represents annotation lists as arrays, compressed until active, and skips annotations with galloping search. However, since the index structures are known only to the `Idx` component, it could employ any file structures and storage strategies able to efficiently support the τ and ρ access methods[10].

The translation function $\mathcal{X}(p, q)$ is implemented by the `translate(p, q)` operation of the `Txt` component. A typical query processing loop for a structural query expressing containment relationships might start with a query Q expressed by the operators of Figure 2. Calls to τ or ρ generate successive solutions, with the content translated and the results aggregated as needed.

$$
\begin{aligned}
&1 \quad Solve(Q) \equiv \\
&2 \quad\quad \langle(p, q), v\rangle \leftarrow Q.\tau(0) \\
&3 \quad\quad \textbf{while } p \neq \infty: \\
&4 \quad\quad\quad Translate/Aggregate \ \langle(p, q), v\rangle \\
&5 \quad\quad\quad \langle(p, q), v\rangle \leftarrow Q.\tau(p + 1)
\end{aligned}
$$

The access methods return $\langle(\infty, \infty), 0\rangle$ to indicate the end of the list. As the solutions are generated, the τ and ρ operators allow solutions to subqueries to be skipped when they cannot lead to a solution for the overall query. The specific translation ($\mathcal{X}$) and aggregation operations required on line 4 depend on the problem at hand. Aggregations include the standard SQL aggregations (MAX, MIN, COUNT, etc.), which need to be preformed in memory.

To provide more concrete examples, we use the heterogeneous collection of JSON objects presented in Figure 5. We base our examples on this collection due to its level of heterogeneity and its independence from this work. The collection was originally created as a resource for exploring and learning MongoDB. Compared to standard benchmarking tools (Belloni et al., 2022) it provides a reasonable source of clear and simple examples, with an emphasis on heterogeneity. Single-thread build time for this collection is just over 4 minutes for a static index and just over 3 minutes for a dynamic index.

---

10. The name "Cottontail" was inspired by the ability of the τ and ρ access methods to efficiently "hop" around the index.

Data Set	Description	Records	Size
books	Descriptions of technical books	431	524K
city_inspections	Results of NYC business inspections	81,047	23M
companies	Overviews of tech companies	18,801	74M
countries-big	Country names by language	21,640	2291K
covers	Book ratings	5,071	470K
grades	Grades for homework assignments	280	91K
products	Phone and cable products	11	2K
profiles	Update log records	1,515	454K
restaurant	Restaurant addresses and ratings	2,548	666K
students	Student grades	200	34K
trades	Stock trades	1,000,001	231M
zips	NYC zip codes	29,353	3107K
Total		**1,160,898**	**337M**

Figure 5: Curated collection of heterogeneous JSON objects compiled by Özler as a resource for exploring MongoDB (`https://github.com/ozlerhakan/mongodb-json-files`).

Figure 6 presents these examples. The Cottontail repo contains associated source code[11]. For each query, we give a description in English, a description in an SQL-like notation, and a query in the structural query notation of Figure 2. The source code should be consulted for full details. The figure includes query execution times for both static and dynamic indexes.

Examples 1-3 follow the general pattern above, i.e. a single query with different types of aggregation. Example 4 involves exploding an array containing author names. In the figure, the structural query for this example returns each array of author names as a whole, while the example code in the repo illustrates the use of array indexes to access individual elements one at a time. Example 5 requires roughly a second on both indexes. Processing this query requires over 80,000 accesses to the content, corresponding to an average access time of $20\,\mu s$ on the static index. Even with various caching methods in place, there are limits on random access to compressed text. As much as possible, query processing should take place over the annotations.

The "FROM *" notation in Examples 7 is not valid SQL. If it were, it would imply a Cartesian product of all tables. Here, it suggests the ability to run queries that span objects with different schema. Examples 8 and 9 provide a more substantial example of annotative indexing that enables unified queries over objects with different schema. The objects in many of the subcollections include properties indicating their creation date. For example, in the city_inspections subcollection, dates are specified in a human readable format (e.g, {"date":"Feb 20 2015"). In the companies subcollection, some dates are specified as UNIX timestamps in milliseconds (e.g. "created_at" : { "$date" : 1180075887000 }). With annotative indexing, we can annotate the objects to provide consistent date annotations, allowing Example 9 to count the objects created on a specific date across all subcollections.

---

11. `https://github.com/claclark/Cottontail/blob/main/apps/json-examples.cc`

	Static	Dynamic

Example 1: Statistics for restaurant ratings
    `SELECT MIN(rating), AVG(rating), MAX(rating)`
       `FROM restaurant`
    `:rating: ◁ Files/restaurant.json`
	14 ms	<1 ms

Example 2: How many zip codes does New York have?
    `SELECT COUNT(*) FROM zips WHERE CITY = "NEW YORK"`
    `(:city: ▷ "New York") ◁ Files/zips.json`
	23 ms	2 ms

Example 3: Names of nanotech companies
    `SELECT name FROM companies`
       `WHERE category_code CONTAINS "nanotech"`
    `:name: ◁ (: ▷ (nanotech`
       `◁ (:category_code: ◁ Files/companies.json)))`
	133 ms	3 ms

Example 4: Titles and authors of books
    `SELECT title, EXPLODE(authors) AS author FROM books`
    `(:title: ▽ :authors:) ◁ Files/books.json`
	95 ms	21 ms

Example 5: How many stock trades?
    `SELECT COUNT(*) FROM trades`
    `: ◁ Files/trades.json`
	70 ms	71 ms

Example 6: Outcomes from city inspections
    `SELECT result, COUNT(result) FROM city_inspections`
       `GROUP BY result`
    `:result: ◁ Files/city_inspections.json`
	1,686 ms	939 ms

Example 7: How many objects in the database?
    `SELECT COUNT(*) FROM *`
    `:`
	< 1 ms	< 1 ms

Example 8: Titles of books publised in 2008
    `SELECT title FROM books`
       `WHERE created >= '2008-01-01'`
       `AND created <= '2008-12-31'`
    `:title: ◁ (Files/books.json ▷ year=2008)`
	13 ms	9 ms

Example 9: Count objects created on December 1, 2008.
    `SELECT COUNT(*) FROM * WHERE created = '2008-12-01'`
    `: ▷ (year=2008 △ month=12 △ Day=01)`
	13 ms	4 ms

Figure 6: Illustrative examples of containment and other operations over the JSON collection from Figure 5, with query processing times over static and dynamic index structures. Examples 8 and 9 depend on additional date annotations not present in the original JSON. The SQL queries are provided for explanatory purpose; they cannot be directly executed by the reference implementation. The structural queries describe index access only; additional processing is required to complete query processing, including aggregations.

6 Dynamic Update

Annotative indexing fosters a dynamic view of the content it stores. After we append text to the content, we can annotate it in different ways and for different purposes. For example, the transformations applied to the MS MARCO corpus described in the introduction, including tagging and segmentation into passages, could be achieved through annotations. For ranking purposes, term frequency values at the document level can be combined with sparse learned weights at the passage level to support hybrid search. Fields in heterogeneous collections of objects can be unified and related objects can be linked.

This section outlines an approach to dynamic update of an annotative index that maximizes flexibility, including support for multiple simultaneous readers and writers. Updates are grouped into transactions. At the start of a transaction, a snapshot is taken of the index state, which remains active until the transaction is committed or aborted. Both content and annotations in this snapshot can be accessed on read-only basis until the transaction ends. For example, during the transaction we might read the content to identify sentence boundaries in passages, or to compute term statistics. During the transaction, we can append to the content and add annotations, but these changes will not be immediately visible in the snapshot. We can also erase content and annotations. Once the update is complete, we follow a two-phrase protocol to commit or abort the update, allowing us to support transactions that span independent annotative indices. After the transaction is complete, the updated content and annotations become visible.

While the Cottontail's static index supports only one transaction at a time, its dynamic index supports multiple concurrent transactions. Each transaction is managed by a `Warren` (see Figure 3). The `clone` operation allows a `Warren` to be copied for the purpose of supporting concurrent transactions, with each clone managing one transaction at a time. For example, in a multi-threaded application each thread could `clone` a copy for it own use. The `start` operation captures the read-only snapshot of the index, while the `end` operation releases this snapshot. Any accesses to the `Warren`, even read-only access, must be bracketed by a `start`/`end` pair. The `transaction` operation starts a write transaction, at which point the `Appender` and `Annotator` may be used. In addition, to the `annotate` operator, the `Annotator` supports an `erase` operation that removes the content and its annotations over a specified interval by annotating the interval with the reserved feature 0. `Txt` and `Hopper` objects skip these intervals until the associated content and annotations can be garbage collected. The remaining operations — `ready`, `commit`, and `abort` — complete the two-phase commit protocol. The update is not visible to the `Warren` until after the `end` operation, followed by another `start`.

Internally, each committed transaction creates a special update `Warren` object that contains only the newly added content and annotations[12]. After a commit, an update `Warren` object is immutable. At the start of the ready phase of the two-phase commit, the index assigns an update `Warren` a sequence number. A vector of `Warren` objects in sequence order provides the snapshot used for read access. In the background `Warren` objects are merged and garbage collected, with a merged `Warren` representing a subindex of the full index, corresponding to a range of updates in sequence order. Once a `Warren` is merged into a larger range and is released from all active snapshots, it is deleted.

---

12. In a dynamic index, warrens multiply like rabbits.

During an update, content and annotations are assembled in a separate address space. At the start of the ready phase, when the index knows the final length of the appended content, it assigns a permanent address interval to the content and maps newly added annotations to this interval. During the ready phase the update is also logged durably to storage. If the commit is aborted after the ready phase, the assigned address interval becomes a gap, and the update is garbage collected from the log. During the update process, a global lock is held only for brief periods, such as when a snapshot is taken or when sequence numbers and address intervals are assigned.

Cottontail supports ACID properties of transactions. Transactions are fully atomic, with newly added content and annotations remaining invisible to `Txt` and `Idx` operations until the transaction is committed. Cottontail guarantees consistency in that updates to annotations preserve minimal interval semantics. However, to maximize concurrency, Cottontail provides limited support for isolation. If concurrent transactions add annotations for the same feature that nest, the index retains only the innermost. If concurrent transactions add annotations with the same start and end addresses, the index retains only the value from the one with the largest sequence number. A failure before the start of a final commit phase of a two-phase commit, guarantees that the transaction is aborted, with no changes. A failure after a commit guarantees that the update is durably recorded. A failure during commit processing will leave the index in a consistent state, with the transaction either committed or aborted.

Figure 7 proves an illustration of dynamic update with multiple concurrent readers and writers[13]. The figure recapitulates four years of older TREC experiments when the test collection changed substantially from year to year (Voorhees and Harman, 1998). Documents for the test collection were distributed on five disks, encoded in an HTML-like format and organized into 4,905 files. TREC-4 used disks 2 and 3; TREC-5 used 2 and 4; TREC-6 used 4 and 5; TREC-7 dropped the low quality CR subcollection from disk 4. 50 new queries were introduced each year, but one query was excluded from TREC-4, leaving 199 queries in total. Each query was judged for relevance over the collection from the year it was introduced, with an average of 1,866 judgments/query. The figure was generated by hundreds of threads concurrently reading and writing a Cottontail dynamic index, including:

1. 28 appending threads, one for each core. Together they append the entire collection, a file at a time. Each file is appended as a separate transaction. After each append is committed, the thread re-reads the documents from the index, computes term statistics for them, and writes the statistics to the index as a second transaction. Finally, if there are documents in the file that are relevant to any of the queries, annotations reflecting these relevance judgments are written to the index as annotations in a third transaction.

2. 199 querying threads, one for each query. Each repeatedly starts a read access, runs its query with BM25, expands the query using pseudo-relevance feedback over the top 20 documents, runs the expanded query to return the top 1000 documents, reads relevance judgments from the index, computes average precision, and reports it on output where it is captured for later summarization on a per-year basis.

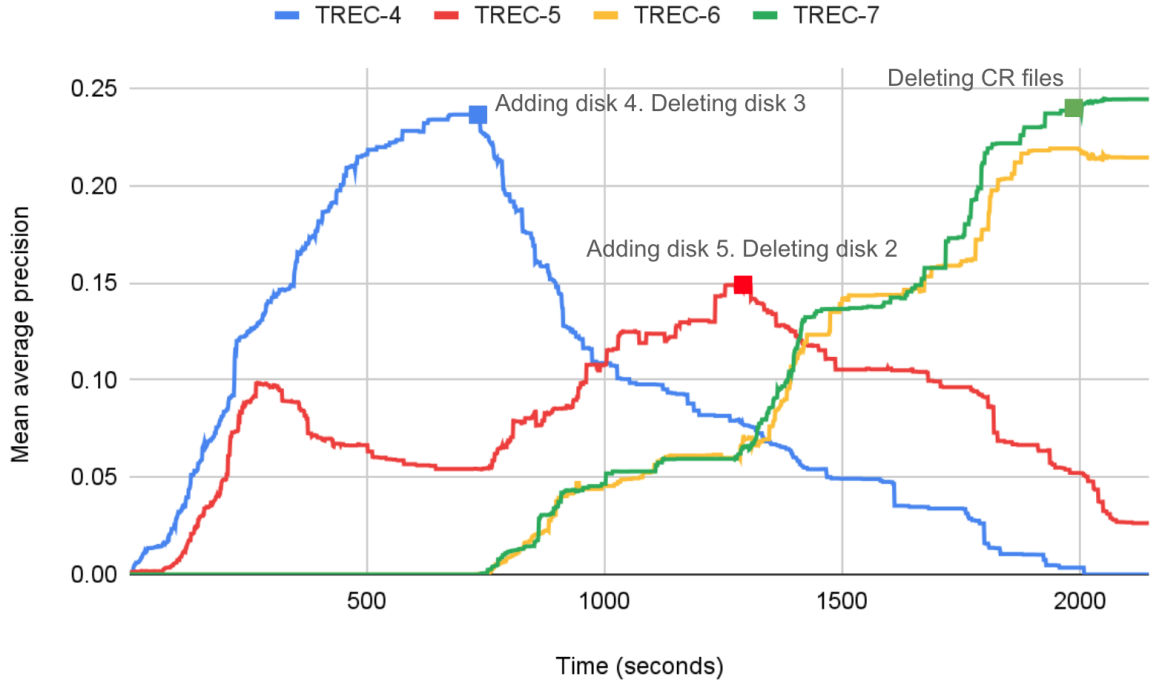13. `https://github.com/claclark/Cottontail/blob/main/apps/trec-example.cc`

Figure 7: Example of transaction processing in cottontail. The example recapitulates four years of early TREC experiments, when the collection was changing significantly from year to year. The example was generated by 28 appending threads — one for each processor core — one deletion thread, and 199 querying threads — one for each query in the TREC-4 to TREC-7 test collections. The appending threads append each of the 4,905 files in the TREC collection as a separate transaction. They then add ranking statistics and relevance information as separate transactions. The deletion thread removes documents, so that collection evolves from year to year. The querying threads run continuously, each executing a BM25 query with pseudo-relevance feedback and then computing mean average precision using relevance information from the index. The lines in the figure plot MAP values as they change over the course of the experiment.

3. One deletion thread. It erases documents, a file at a time, so that the collection evolves over time. Each file is erased as as separate transaction. The squares in the figure indicate points where the deletion thread synchronizes with the other threads so that all queries are executed at least once on the entire collection for a given year.

In addition to these application-level threads, maintenance threads work throughout the experiment to merge and garbage collect the index. The experiment requires 16,442 update transactions in total. By the end of the experiment, these have been merged into 12 subindexes, each corresponding to a thousand or so sequence numbers. Throughout the experiment, processor utilization essentially remains at 100% on all cores.

As documents are added to the index for a given year, the MAP value for that year increases until it hits a synchronization point. It then drops as documents are deleted. The

BM25 parameters are tuned for more recent collections. The peak MAP values represent good performance on TREC-6 and TREC-7, and reasonable performance on TREC-4 and TREC-5.

7 Conclusion

This paper introduces and explores annotative indexing, a novel and flexible indexing framework, which unifies and generalizes inverted indexes, column stores, object stores, and graph databases. A particular feature of annotative indexing is its ability to manage heterogeneous collections of semi-structured data, unifying common elements across diverse formats. Text in any format can simply be appended to the content, with annotations added at a later time for a variety of purposes, such as sentence segmentation, tagging, or indexing for ranked retrieval.

Integrating annotative indexing into a retrieval augmented generation (RAG) system (Gao et al., 2024) forms a primary focus for current and future work. Given a few examples, a large language model (LLM) can generate structural queries using the operators of Figure 2, allowing natural language queries to be translated into structured queries over heterogeneous content. For example, imagine a life-logging application supported by a RAG system that integrates an annotative index. Messages, mail, conversations, and other experience could be poured into the index as content for ongoing tagging, linking, indexing, and other annotation. From the perspective of a person using the application, querying their past experience ("I really liked the movie I saw on the plane last weekend. What are similar movies I haven't seen yet?") happens in natural language, but internally this query could be handled by a combination of ranked retrieval and structured queries to a knowledge graph linked with the experiences.

Extending annotative indexing to support dense retrieval provides another immediate goal. While a 64-bit value in an annotation cannot store a dense vector, it can store a vector identifier. However, to better support a fully dynamic index, the author plans to mimic the approach taken for the content translation function $\mathcal{X}(p,q)$ by associating vectors with positions in the address space. A vector mapping function $\mathcal{V}(p)$ would return the vector associated with a location in the address space, presumably the location where the corresponding content appears. In this way, dense vectors can be garbage collected as intervals in the address space are erased.

To provide further support for dense retrieval, current work also includes an exploration of methods for encoding HNSW graphs as annotations. Graph structures can be represented in two ways by an associative index. First, as suggested in Section 3.5, we can store an address as the value in an annotation, so that $\langle G, p, v \rangle$ indicates a directed edge from p to v in the graph G. However, unless we are careful with updates, this representation can create "dangling references" to deleted content. An alternative representation stores a feature representing a list of out edges as the value in an annotation. Under this representation, the value in the annotation $\langle G, p, E_p \rangle$ is a feature indicating outlinks from the content at p in the graph G. An annotation for E of the form $\langle E_p, p' \rangle$ indicates a directed edge from p to p'. While the details are left for a future paper, this second representation should allow for the representation and traversal of HNSW graphs through annotations.

At the time of writing, the largest collection indexed by Cottontail is the 350GB C4 corpus,[14] which the author routinely uses for cross-collection pseudo-relevance feedback. Ongoing work includes scaling Cottontail to handle larger collections, as well as fully distributed collections. Cottontail has indexed Wikidata as eight shards directly from its JSON dump,[15] and the author is exploring support for knowledge graph queries over this collection.

At the time of writing, Cottontail remains an experimental system. Compilation requires the Bazel build system and the Boost C++ library, both of which require some effort to install. The system has not been ported to Mac or Windows, and it currently runs only on Ubuntu. In the near future, I plan a single-file release in the spirit of SQLite, along with examples illustrating common use cases. Wider use of Cottontail also requires a Python wrapper.

Current performance on basic BM25 ranking does not quite match that of Lucene. In the immediate future, I plan to focus attention on improving performance of the lowest level code for the `Hopper` operations, which should improve general performance, including ranking. Finally, the only delete operation supported by Cottontail is to erase all content and annotations from an interval of the address space. Additional delete operations might delete specific annotations or all annotations for a given feature.

Acknowledgments and Disclosure of Funding

References

Diego Arroyuelo, Mauricio Oyarzún, Senén González, and Víctor Sepúlveda. Hybrid compression of inverted lists for reordered document collections. *Information Processing & Management*, 54(6):1308–1324, 2018.

Nima Asadi, Jimmy Lin, and Michael Busch. Dynamic memory allocation policies for postings in real-time Twitter search. In *19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 1186–1194, 2013.

Payal Bajaj, Daniel Campos, Nick Craswell, Li Deng, Jianfeng Gao, Xiaodong Liu, Rangan Majumder, Andrew McNamara, Bhaskar Mitra, Tri Nguyen, Mir Rosenberg, Xia Song, Alina Stoica, Saurabh Tiwary, and Tong Wang. MS MARCO: A human generated machine reading comprehension dataset, 2018. URL `https://arxiv.org/abs/1611.09268`.

---

14. `https://huggingface.co/datasets/allenai/c4`
15. `https://www.wikidata.org/wiki/Wikidata:Database_download`

Hannah Bast, Johannes Kalmbach, Theresa Klumpp, and Claudius Korzen. Knowledge graphs. In Omar Alonso and Ricardo Baeza-Yates, editors, *Information Retrieval: Advanced Topics and Techniques*, chapter 2. ACM Press, 2025.

Stefano Belloni, Daniel Ritter, Marco Schröder, and Nils Rörup. Deepbench: Benchmarking JSON document stores. In *9th International Workshop on Testing Database Systems*, page 1–9, 2022.

Paolo Boldi and Sebastiano Vigna. Efficient optimally lazy algorithms for minimal-interval semantics. *Theoretical Computer Science*, 648:8–25, 2016.

Paolo Boldi and Sebastiano Vigna. On the lattice of antichains of finite intervals. *Order*, 35(1):57–81, 2018.

Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *12th International Conference on Information and Knowledge Management*, page 426–434, 2003.

Sebastian Bruch, Franco Maria Nardini, Cosimo Rulli, and Rossano Venturini. Efficient inverted indexes for approximate retrieval over learned sparse representations. In *47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, page 152–162, 2024.

Stefan Büttcher, Charles L. A. Clarke, and Gordon V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010.

Charles L. A. Clarke. *An Algebra for Structured Text Search*. PhD thesis, University of Waterloo, 1996. URL `https://plg.uwaterloo.ca/~claclark/phd.pdf`.

Charles L. A. Clarke and Gordon V. Cormack. Shortest-substring retrieval and ranking. *ACM Transactions on Information Systems*, 18(1):44–78, 2000.

Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38(1):43–56, 1995a.

Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. Schema-independent retrieval from hetrogeneous structured text. In *4th Annual Symposium on Document Analysis and Information Retrieval*, pages 279–289, Las Vegas, Nevada, 1995b.

Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, Sriram Sankar, Guanghao Shen, Gintaras Woss, Chao Yang, and Ning Zhang. Unicorn: A system for searching the social graph. *VLDB Journal*, 6(11):1150–1161, 2013.

Zhuyun Dai and Jamie Callan. Context-aware sentence/passage term importance estimation for first stage retrieval, *ArXiv preprint arXiv:1910.10687*, 2019. URL `https://arxiv.org/abs/1910.10687`.

Zhuyun Dai and Jamie Callan. Context-aware document term weighting for ad-hoc search. In *The Web Conference*, page 1897–1907, 2020.

Constantinos Dimopoulos, Sergey Nepomnyachiy, and Torsten Suel. Optimizing top-k document retrieval strategies for block-max indexes. In *6th ACM International Conference on Web Search and Data Mining*, page 113–122, 2013.

Shuai Ding and Torsten Suel. Faster top-k document retrieval using block-max indexes. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 993–1002, 2011.

Patrick Eades, Anthony Wirth, and Justin Zobel. Immediate text search on streams using apoptosic indexes. In *44th European Conference on IR Research*, page 157–169, 2022.

Thibault Formal, Benjamin Piwowarski, and Stéphane Clinchant. SPLADE: Sparse lexical and expansion model for first stage ranking. In *44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, page 2288–2292, 2021.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey, *ArXiv preprint arXiv:2312.10997*, 2024. URL https://arxiv.org/abs/2312.10997.

Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In *Conference on Empirical Methods in Natural Language Processing*, 2020.

Carlos Lassance, Hervé Dejean, Stéphane Clinchant, and Nicola Tonellotto. Two-step SPLADE: Simple, efficient and effective approximation of splade. In *46th European Conference on Information Retrieval*, page 349–363, 2024.

Jurek Leonhardt, Koustav Rudra, Megha Khosla, Abhijit Anand, and Avishek Anand. Efficient neural ranking using forward indexes. In *ACM Web Conference*, page 266–276, 2022.

Jimmy Lin and Xueguang Ma. A few brief notes on DeepImpact, COIL, and a conceptual framework for information retrieval techniques, *ArXiv preprint arXiv:2010.11386*, 2021. URL https://arxiv.org/abs/2106.14807.

Sheng-Chieh Lin, Jheng-Hong Yang, and Jimmy Lin. Distilling dense representations for ranking using tightly-coupled teachers. *ArXiv preprint arXiv:2010.11386*, 2020. URL https://arxiv.org/abs/2010.11386.

Xueguang Ma, Hengxin Fun, Xusen Yin, Antonio Mallia, and Jimmy Lin. Enhancing sparse retrieval via unsupervised learning. In *1st Annual International ACM SIGIR Conference*

on Research and Development in Information Retrieval in the Asia Pacific Region, page 150–157, 2023.

Joel Mackenzie and Alistair Moffat. Examining the additivity of top-k query processing innovations. In *29th ACM International Conference on Information & Knowledge Management*, page 1085–1094, 2020.

Joel Mackenzie, Andrew Trotman, and Jimmy Lin. Wacky weights in learned sparse representations and the revenge of score-at-a-time query evaluation, *ArXiv preprint arXiv:2110.11540*, 2021. URL https://arxiv.org/abs/2110.11540.

Yu A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, page 824–836, 2020.

Antonio Mallia, Giuseppe Ottaviano, Elia Porciani, Nicola Tonellotto, and Rossano Venturini. Faster BlockMax WAND with variable-sized blocks. In *40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, page 625–634, 2017.

Antonio Mallia, Michał Siedlaczek, and Torsten Suel. An experimental study of index compression and DAAT query processing methods. In *41st European Conference on IR Research*, page 353–368, 2019.

Antonio Mallia, Torsten Suel, and Nicola Tonellotto. Faster learned sparse retrieval with block-max pruning. In *47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, page 2411–2415, 2024.

Alistair Moffat and Joel Mackenzie. Efficient immediate-access dynamic indexing. *Information Processing & Management*, 60(3), 2023.

Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, 1996.

Hannes Mühleisen, Thaer Samar, Jimmy Lin, and Arjen de Vries. Old dogs are great at new tricks: Column stores for IR prototyping. In *37th International ACM SIGIR Conference on Research & Development in Information Retrieval*, page 863–866, 2014.

James Jie Pan, Jianguo Wang, and Guoliang Li. Survey of vector database management systems. *The VLDB Journal*, 33(5):1591–1615, 2024.

Matthias Petri, J. Shane Culpepper, and Alistair Moffat. Exploring the magic of WAND. In *18th Australasian Document Computing Symposium*, page 58–65, 2013.

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, 2019.

S. E. Robertson and S. Walker. Some simple effective approximations to the 2-Poisson model for probabilistic weighted retrieval. In *17th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 232–241, 1994.

S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, and M. Gatford. Okapi at TREC-3. In *3rd Text REtrieval Conference*, 1994.

Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *VLDB Journal*, 11(4), 2023.

Xinying Song, Alex Salcianu, Yang Song, Dave Dopson, and Denny Zhou. Fast Word-Piece tokenization, In: *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2021. `https://aclanthology.org/2021.emnlp-main.160/`.

Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented DBMS. In *31st International Conference on Very Large Data*, pages 553–564, 2005.

Howard Turtle and James Flood. Query evaluation: Strategies and optimizations. *Information Processing & Management*, 31(6):831–850, 1995.

Ellen M. Voorhees and Donna Harman. Overview of the seventh Text REtrieval Conference (TREC-7). In *7th Text REtrieval Conference*, 1998.

Hugh E. Williams and Justin Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999.

Jingtao Zhan, Jiaxin Mao, Yiqun Liu, Min Zhang, and Shaoping Ma. RepBERT: Contextualized text embeddings for first-stage retrieval, *ArXiv preprint arXiv:2006.15498*, 2020. URL `https://arxiv.org/abs/2006.15498`.

Wayne Xin Zhao, Jing Liu, Ruiyang Ren, and Ji-Rong Wen. Dense text retrieval based on pretrained language models: A survey. *ACM Transactions on Information Systems*, 42 (4), 2024.