Scalable Distributed Inverted List Indexes in Disaggregated Memory

MANUEL WIDMOSER, University of Salzburg, Austria
DANIEL KOCHER, University of Salzburg, Austria
NIKOLAUS AUGSTEN, University of Salzburg, Austria

Memory disaggregation separates compute (CPU) and main memory resources into disjoint physical units to enable elastic and independent scaling. Connected via high-speed RDMA-enabled networks, compute nodes can directly access remote memory. This setting often requires complex protocols with many network roundtrips as memory nodes have near-zero compute power.

In this paper, we design a scalable distributed inverted list index for disaggregated memory architectures. An inverted list index maps a set of terms to lists of documents that contain this term. Current solutions either partition the index horizontally or vertically with severe limitations in the disaggregated memory setting due to data & access skew, high network latency, or out-of-memory errors. Our method partitions lists into fixed-size blocks and spreads them across the memory nodes to balance skewed accesses. Block-based list processing keeps the memory footprint of compute nodes low and masks latency by interleaving remote accesses with expensive list operations. In addition, we propose efficient updates with optimistic concurrency control and read-write conflict detection. Our experiments confirm the efficiency and scalability of our method.

CCS Concepts: • **Information systems → Data structures**; **Parallel and distributed DBMSs**; • **Networks → Network protocols**.

Additional Key Words and Phrases: Distributed Database Management Systems, Disaggregated Memory, Inverted Index, RDMA.

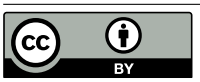
ACM Reference Format:
Manuel Widmoser, Daniel Kocher, and Nikolaus Augsten. 2024. Scalable Distributed Inverted List Indexes in Disaggregated Memory. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 171 (June 2024), 27 pages. https://doi.org/10.1145/3654974

1 INTRODUCTION

Search engines such as Google [5, 19], Yahoo! [4, 13], and Bing [25, 58] heavily rely on inverted list indexes to find relevant documents efficiently. In information retrieval, an inverted index stores a list of ordered *document identifiers* for each vocabulary *term* that appears in the documents. A search query returns all relevant identifiers of documents that contain either all or at least one of the terms. Database systems leverage inverted list indexes to speed up query processing [7, 43, 49, 54]; in graph analytics they are used, for example, to find common followers for a set of given users [17].

Due to the rapid growth of data, index structures that organize very large datasets often do not fit into main memory of a single machine. Spilling data onto slow secondary storage is undesirable for applications that require ultra-low latency access, such as indexes for in-memory database systems. Indexes residing on a single machine are further limited by the number of cores in terms of

Authors' addresses: Manuel Widmoser, University of Salzburg, Austria, manuel.widmoser@plus.ac.at; Daniel Kocher, University of Salzburg, Austria, daniel.kocher@plus.ac.at; Nikolaus Augsten, University of Salzburg, Austria, nikolaus.augsten@plus.ac.at.



ACM 2836-6573/2024/6-ART171
https://doi.org/10.1145/3654974

the access rate they can support. Distributing the index structure improves the scalability to large index sizes and can reduce hot spots, which arise when the index is frequently read or updated [63]. Scaling out in-memory indexes is necessary to build fully elastic distributed database systems with near in-memory performance.

State-of-the-art approaches that distribute inverted list indexes, partition the index at the level of either the documents or the terms. The index components are then distributed to monolithic servers [3, 14, 15, 39, 40, 50]. Each server locally stores and maintains part of the index and answers queries by sending back subsets of the result. The servers communicate using traditional message passing techniques like remote procedure calls (RPCs). Such approaches have been designed under the assumption that the network is the main bottleneck. However, with the availability of high-performance networks, e.g., InfiniBand, this assumption no longer holds.

RDMA-enabled networks provide high bandwidth (100 Gbps) and low latency (2 μs) [20, 22]. Low latency is achieved through one-sided RDMA operations that directly access pre-registered memory regions on a remote machine without involving its CPUs. However, efficient RDMA communication must be carefully designed and often involves complex protocols with multiple network roundtrips. Directly accessing the memory of remote servers allows for memory nodes with near-zero computation power, load balancing, and shared memory usage across server lakes in data centers [23, 45].

Memory disaggregation is a new hardware architecture that physically separates monolithic compute and memory resources into two independent network-attached components [12, 21, 30, 32, 35, 60]. *Compute nodes* are equipped with strong computation power (i.e., many CPU cores) and a small amount of main memory (a few GBs) for caching hot data. *Memory nodes* are equipped with large amounts of main memory but have near-zero computation power (intended for lightweight management tasks). Compute nodes can directly access the memory locations of the memory nodes through high-speed RDMA networks. In contrast to traditional architectures with monolithic servers, memory disaggregation allows for scaling compute power and memory independently and thus significantly improves resource utilization as well as elasticity [1, 53, 65].

In this paper, we design an efficient distributed in-memory inverted list index – an important standard data structure in information retrieval and database systems – for disaggregated memory architectures. We revisit key techniques, such as term-based and document-based partitioning, and evaluate their performance in disaggregated memory scenarios. Conventional message passing with RPCs is not feasible in disaggregated memory due to the low computational resources of the memory nodes. Adapting the existing partitioning techniques to a disaggregated memory architecture raises serious scalability issues: term-based partitioning leads to skewed network access to the memory nodes; document-based distribution suffers from high latency and increased network contention due to many network roundtrips that are required. We face the challenge of developing a new technique that adopts the features of existing techniques without suffering from their shortcomings. Various careful design decisions are required to achieve this goal, taking into account the specifics of the disaggregated memory setting.

To address the limitations of existing distribution schemes, we design a new partitioning technique that divides the lists into fixed-size blocks and spreads them uniformly across the memory nodes. The blocks are connected via remote pointers. Short lists are accessed with a single network roundtrip, and long lists are read from different memory nodes to evenly balance the network traffic.

Our index structure is designed to support update operations, which have received little attention in literature: an insert (delete) operation inserts (removes) a new item into a given list preserving the order. Updating long inverted lists without sacrificing read query performance is challenging. To enable efficient dynamic index operations, we support lock-free reads and use write locks on

blocks to address write-write conflicts. For update operations, fine-grained write-locks are desirable since only small parts of the list (a single block) must be locked rather than the entire list. Similar to insert operations in B-trees, when an insertion exceeds the block capacity, the block is split and its items are distributed. This requires remote memory allocation and is realized using lock-free remote accesses to free lists on the memory nodes. Deleting items may lead to empty blocks, which induces critical paths such as remote pointer swapping and block deallocation. We implement pair-wise pointer-block tagging to mitigate synchronization issues such as the ABA problem. For a high read performance, our index structure supports lock-free reads by validating blocks with cache-line versions to detect conflicting updates [20]. To the best of our knowledge, the proposed method is the first distributed inverted list index that has been specifically designed for disaggregated memory.

Contributions. The main contributions of this paper are the following:

(1) We identify the performance bottlenecks of state-of-the-art inverted list indexes for disaggregated memory settings.
(2) The design of the first inverted list index for disaggregated memory. Compared to prior distributed indexes, our index (a) avoids skewed network traffic even when the data or the queries are skewed, (b) the compute memory is independent of the index size, (c) short lists are read in a single roundtrip.
(3) Our index supports lock-free read operations and fine-grained write-locks for fast concurrent updates.
(4) An extensive performance study demonstrates the scalability of our approach on various real-world benchmarks.

Paper Outline. We discuss applications in Section 2 and provide background information in Section 3. Section 4 defines our objectives and identifies major challenges for developing an efficient distributed inverted list index under memory disaggregation. We revisit traditional partitioning approaches in Section 5. In Sections 6-8, we propose our new index, introduce updates, and discuss memory management. Section 9 presents our experimental study. Related work is discussed in Section 10, and Section 11 concludes the paper.

2 APPLICATIONS

Inverted list indexes are widely adopted in a range of applications to efficiently process search queries. This section briefly summarizes the most common usage scenarios in different domains.

Information Retrieval. Queries are keywords (terms), and documents are fetched using inverted list indexes [4, 5, 13, 19, 25, 58]. A list represents a term that stores all identifiers of those documents that contain the term. A query returns either all document identifiers that share the terms given in the query (i.e., the result is the intersection of the lists) or the document identifiers that contain at least one of the terms (i.e., the result is the union of the lists).

Query Processing. To speed up predicate evaluation of SQL queries, many database systems precompute lists of row ids that satisfy a certain predicate and store them in an index structure [7, 43, 49, 54]. Thus, a list represents a specific predicate. As an example, consider the following SQL query that computes the yearly revenue after 1992 for a specific product brand in the European region:

```
SELECT sum(lo_revenue), d_year, p_brand
FROM lineorder_flat
WHERE s_region = 'EUROPE' AND p_brand = 'MFGR#2239' AND d_year >= 1993
GROUP BY d_year, p_brand;
```

The WHERE-clause of the query can be reduced to an intersection query of three lists of row ids where each list satisfies one of the three predicates, e.g., the first list contains all row ids of the table lineorder_flat where s_region is equal to EUROPE and so on.

Graph Analytics. In graph analytics, a list represents all vertices adjacent to a given vertex [17]. For example, a follower-graph of a social network, where an inverted list represents a user that is followed by the users in the list. A query returns common followers between a group of people.

Other Applications. Inverted list indexes are utilized for efficient query answering in numerous other applications. In similarity search, an inverted list index is built to detect candidates (objects that are potentially similar to each other) [28, 42]. In data warehouse applications, bioinformatics, and survey-based statistical analysis, inverted list indexes are employed for efficient multi-dimensional on-the-fly analysis in data mining systems [34, 36].

3 BACKGROUND

In this section, we concisely describe the fundamentals of memory disaggregation, RDMA, and the inverted list index structure.

3.1 Memory Disaggregation

Memory disaggregation physically decouples traditional monolithic servers into two hardware units: compute and memory nodes. Compute nodes have high computation power (i.e., many CPU cores) but only a few GBs of main memory, for instance, to cache hot data or store translation tables. In contrast, memory nodes have near-zero compute power (i.e., only a few CPU cores to handle elementary maintenance tasks) but large amounts of main memory. Data records and indexes are distributed among the memory nodes and stored in main memory, whereas computation tasks – such as querying the index or analyzing the data – are performed solely on the compute nodes. Therefore, memory and compute power scale independently: If more storage is needed, the memory nodes are scaled up, and if more compute power is required, the number of compute nodes is increased. For efficient memory disaggregation, compute nodes need direct access to the memory of remote nodes, which is enabled through modern networks that support RDMA.

3.2 RDMA

Remote Direct Memory Access (RDMA) is a high-performance network communication mechanism with low latency and primarily leveraged through InfiniBand networks. The main benefit is that remote memory can be accessed directly and efficiently, which is a key requirement to enable memory disaggregation.

RDMA Verbs. There are two types of communication primitives, so-called *verbs*, for RDMA: one-sided and two-sided verbs. One-sided verbs consist of RDMA_READ, RDMA_WRITE, and atomic operations (compare-and-swap and fetch-and-add); they require only one active side. For instance, a client node can read from the memory of a remote machine by specifying the remote address of a pre-registered *memory region* without involving the CPU of the remote machine. Two-sided verbs consist of SEND and RECEIVE operations and are similar to socket-based communication. Two-sided verbs are applicable to traditional message passing (e.g., by implementing RPCs), where the remote machine must posts a RECEIVE before another machine can issue a SEND request. For disaggregated memory architectures, only one-sided verbs are eligible, because two-sided verbs also involve the CPU of the remote machine. Therefore, we focus on one-sided RDMA verbs in this work, which effectively access memory servers with near-zero computation power.

RDMA Communication. To communicate among nodes, queue pairs (consisting of a send and a receive queue) and completion queues are created to establish an RDMA connection. For example, if a client node reads from remote memory, the software layer posts a work queue element (WQE) into the send queue of the queue pair that is connected to the target machine. The WQE specifies meta data such as the remote address, the size to read, and the address of the client's local buffer (to which data is read from remote memory). The RDMA network interface controller (NIC) pushes a completion event into the completion queue as soon as the request has been processed. The finalized request can be detected by polling the completion queue. We refer to Ziegler et al. [61] for more details.

3.3 Inverted List Index Structure

We stick to the terminology used in information retrieval and assume a collection of so-called *documents* $\boldsymbol{D}$ of size $|\boldsymbol{D}|$, each of which contains one or multiple *terms*, e.g., a document can be a sequence of words in a particular vocabulary. The set of distinct terms over all documents is the *universe* $\boldsymbol{U}$ of size $|\boldsymbol{U}|$ with t_i, $0 < i \leq |\boldsymbol{U}|$, being the i-th term in $\boldsymbol{U}$. The j-th document of our collection $\boldsymbol{D}$ is a subset of $\boldsymbol{U}$ and has a unique document identifier d_j, $0 < j \leq |\boldsymbol{D}|$. An *inverted (list) index* maintains one list $\boldsymbol{L}_i$ for each term $t_i \in \boldsymbol{U}$, and $\boldsymbol{L}_i$ stores one document identifier d_j (also called *item*) for each document that contains t_i. Entries in a list $\boldsymbol{L}_i$ are unique and strictly ordered by document identifiers.

4 INDEX BLUEPRINT & CHALLENGES

This section introduces a blueprint of an efficient and scalable distributed inverted index for very large documents in a memory-disaggregated database system. To this end, we specify prerequisites, formulate objectives, and identify potential challenges.

4.1 Index Operations

Let t_i be a term and $\boldsymbol{L}_i$ the corresponding list, d_j be a document identifier, and $\{t_{i_1}, t_{i_2}, \ldots, t_{i_k}\}$ be a set of k distinct terms with $i_h \in \mathbb{N}^+$. Then, the following operations are common for inverted list indexes: (1) $lookup(t_i)$: Return $\boldsymbol{L}_i$, or $\emptyset$ if $\boldsymbol{L}_i$ does not exist. (2) $insert(d_j)$: Add d_j to the lists of all terms that appear in the corresponding document while maintaining the ordering (possibly creating new lists). (3) $delete(d_j)$: Remove d_j from the lists of all terms that appear in the corresponding document (possibly removing existing lists). (4) $intersect(\{t_{i_1}, \ldots, t_{i_k}\})$: Return the document identifiers that appear in all lists, i.e., $\{d_x \mid d_x \in \bigcap_{h=1}^{k} \boldsymbol{L}_{i_h}\}$. (5) $union(\{t_{i_1}, \ldots, t_{i_k}\})$: Return the document identifiers that appear in at least one of the lists, i.e., $\{d_x \mid d_x \in \bigcup_{h=1}^{k} \boldsymbol{L}_{i_h}\}$.

4.2 Objective

Memory nodes store the inverted list index and do not execute any computations. All index operations are performed by the compute nodes by remotely accessing the contents of the memory nodes. Our overall objective is to design a highly scalable distributed in-memory inverted list index for disaggregated memory systems with efficient support for all index operations (cf. Section 4.1).

4.3 Challenges

We identify five major challenges C1–C5 that must be addressed for an efficient and scalable distributed in-memory inverted index under disaggregated memory.

C1 ***Data & Access Skew.*** Skewed data and accesses are common problems for inverted indexes, e.g., the word frequency of spoken language follows a power law distribution. Some terms may appear more frequently than others, resulting in lists of different lengths. Similarly, some terms

may be queried more often than others resulting in access skew. In the worst case, the load of all query terms goes to one memory node, thus network-bounding the throughput.

C2 ***Small Memory.*** Compute nodes are equipped with only a few GBs of main memory (we use 10 GB), which affects memory efficiency as well as runtime performance. Every thread running on a compute node has a separate local buffer, hence the memory is split among all threads. For example, the theoretical max. local buffer size for 32 threads is 312 MB, which renders loading multiple lists infeasible (e.g., the longest list in CCNEWS has 170 MB).

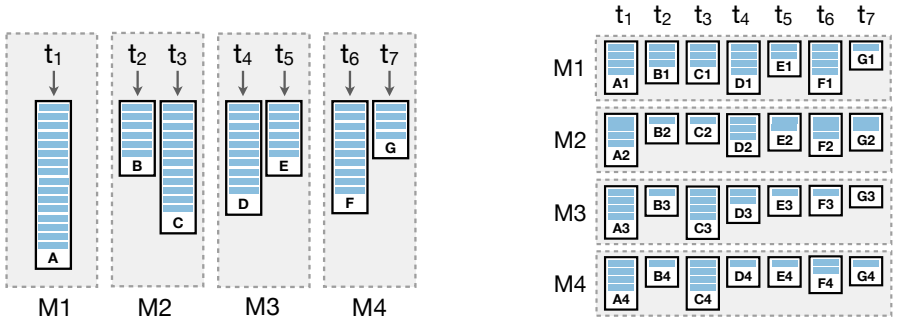
C3 ***Network Latency Masking.*** Despite low-latency remote memory access in modern networks, masking latency is indispensable for a scalable distributed index structure. Compared to local memory latency ($\approx 0.1\ \mu$s), remote memory latency ($\approx 2\ \mu$s) is still an order of magnitude higher. Thus, the challenge is to mask latency as effectively as possible by interleaving operation and network communication (for intersection/union in particular).

C4 ***Minimum Network Roundtrips.*** Data skew may result in short lists and we need more roundtrips to access them in case the lists are partitioned among many memory nodes. The more roundtrips, the more contention on the NIC, and the higher the latency. The challenge is to keep the number of roundtrips low even for many short lists, otherwise the overall throughput will significantly drop.

C5 ***Remote Index Updates.*** Most previous work only briefly considers update operations on distributed inverted indexes. Aside from read accesses, our goal is a distributed inverted list index with support for two update operations: insertion and deletion. The sort order of the lists must be maintained by updates. This requires us to develop a suitable data layout, a concurrency control scheme that provides consistency, a verification mechanism to detect read-write conflicts, and an efficient memory management technique.

5 STATE OF THE ART

In this section, we revisit state-of-the-art approaches for distributed inverted list indexing and discuss their performance in a disaggregated memory setting.

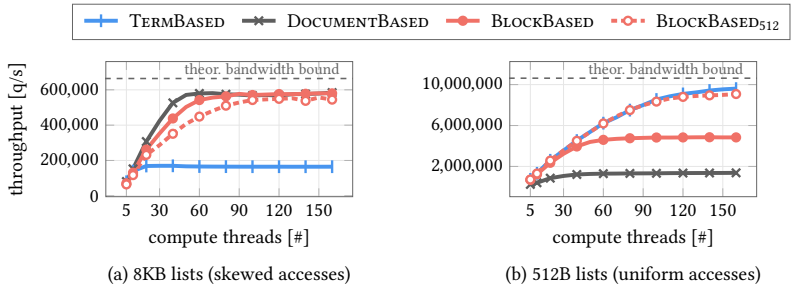


(a) Term-based (b) Document-based

Fig. 1. Partitioning schemes.

5.1 Term-based Partitioning

In term-based partitioning [3, 27, 51], the index is vertically divided as shown in Figure 1a. Hence, an entire list $\boldsymbol{L}_i$ for a term t_i is stored on a predefined memory node. The memory node for a term



(a) 8KB lists (skewed accesses)

(b) 512B lists (uniform accesses)

Fig. 2. Limitations of traditional methods.

is typically determined such that the load distribution remains in balance [15]. For example, choose node $\boldsymbol{M}$ with smallest cost, where the cost is the number of total list items already stored on $\boldsymbol{M}$.

Term-based partitioning can be easily adopted to the disaggregated memory architecture. We store the lists compact on the memory nodes and each compute node stores a catalog (i.e., a hash-table) that returns a *remote address range* for a term t_i. The remote address range encodes the memory node id and an address tuple with references to the beginning and the end of the list in remote memory. The catalog's response is stored in 16 Byte: Two times 48 bits for the addresses (the virtual address space of modern machines is 2^{48}) and the remaining bits to encode the memory node. Note that the catalog is shared among all threads on a compute node and its size depends on the universe size, i.e., $O(|U|)$. In most cases, the catalog requires a few hundred MBs that can be cached in the memory of a compute node. For instance, the catalog of our largest dataset CCNEWS with 29M terms requires approx. 472 MB. If the catalog size exceeds the compute node's memory, the catalog can be stored on the memory nodes and partially cached on the compute node.

To query the index with $\{t_{i_1}, ..., t_{i_k}\}$, a *worker* (i.e., a compute thread) on the compute node first requests the remote addresses from the catalog and reads the lists $\boldsymbol{L}_{i_j}$, $0 < j \leq k$, from remote memory into a contiguous buffer using `RDMA_READ`s. The list operation (e.g., an intersection) is performed locally and the result is reported. For efficient list operations, all required lists must be locally available. A k-way intersection is then computed by cycling through k ordered lists and reporting the ids that occur k times.

The major benefit of term-based partitioning is the minimum number of network roundtrips C4: Each list involved in a query q is read only once from a single memory node to the worker's local buffer, resulting in $|q|$ roundtrips. Figure 2 shows the throughput of TermBased for different scenarios using five compute and four memory nodes (cf. Section 9 for details on the setup). TermBased scales well for uniform list accesses and list sizes (cf. Figure 2b).

Limitations. TermBased is susceptible to access skew (a realistic scenario): If specific terms are queried more frequently than others, the throughput is bound by the network bandwidth of the hot memory nodes C1. In Figure 2a, the throughput is bound by a single memory node (worst case). Moreover, a worker in TermBased must wait until all lists are available in the local buffer before performing an operation C3. As a result, workers straggle if they wait for long lists to be copied from remote memory into the local buffers. Finally, all queried lists must fit into a worker's local buffer C2.

5.2 Document-based Partitioning

Another popular distribution scheme is document-based partitioning [11, 18, 26], where the index is partitioned horizontally as shown in Figure 1b: Each memory node logically stores all the terms,

however, the lists are partitioned by document identifiers across the memory nodes. In other words, each document is assigned to a dedicated memory node (e.g., by using a hash function) and each memory node stores its own local index.

In traditional message passing frameworks, a query is sent to *all* memory nodes. Each memory node queries its local index, computes a partial result, and sends it back to the compute node, which combines the partial results to a final query response. This, however, is not possible under memory disaggregation, where the memory nodes have near-zero computation power, and pushing list operations to the memory nodes would drastically limit the scalability.

Nonetheless, we can deploy the partitioning scheme to the disaggregated memory architecture and RDMA_READ the partial lists of each memory node into the local buffer of a worker and compute the result. This can be done incrementally: first, we RDMA_READ for each query term in $\{t_{i_1}, ..., t_{i_k}\}$ the partial lists stored on the first memory node and compute the partial result, then the lists of the second memory node, and so on. For each memory node, the local buffer of the worker can be reused, leading to a smaller memory footprint on the compute nodes. When using more than a single local buffer per worker, the network transfer and the list operation can be interleaved, which boosts query performance because the network latency is masked to some extent C3. Moreover, partitioning the index at the document-level leads to a finer granularity than partitioning the index on the list-level. Therefore, the inverted lists of the index are better balanced among the memory nodes and the index is less prone to skewed accesses C1 compared to TermBased as shown in Figure 2a.

Limitations. Unfortunately, the number of roundtrips does no longer depend on the query size $|q|$ alone but also on the number of memory nodes $|\boldsymbol{M}|$. Hence, for each query term, we must post an RDMA_READ per memory node, leading to $|q| \cdot |\boldsymbol{M}|$ roundtrips C4. As a result, the performance will significantly drop if many partial lists are short, e.g., the partial lists for term t_7 in the example of Figure 1b. The consequences are shown in Figure 2b, where the lists contain 128 items (4 Byte per item) on average. Note that the performance decreases with an increasing number of memory nodes. Another limitation is that each compute node stores a catalog that returns for a term t_i the remote address ranges for each memory node, because *all* memory nodes store parts of *all* lists C2. In contrast to TermBased, the catalog needs $|\boldsymbol{M}|$ times more space: $O(|\boldsymbol{U}| \cdot |\boldsymbol{M}|)$.

6 OPTIMIZING READ OPERATIONS

Most message passing distributed inverted list index implementations make use of document-based partitioning due to better scalability in traditional (slow) network architectures [15, 46]. However, message passing is not efficient under memory disaggregation and adapting both methods to the disaggregated memory architecture (TermBased and DocumentBased, respectively), introduces severe limitations as shown in Figure 2. A term-based index struggles with skewed network accesses C1, requires large local read buffers C2, and cannot efficiently mask network latency C3 (cf. Section 5.1). The document-based scheme requires significantly more roundtrips (dependent on the number of memory nodes) C4, needs more space for the catalog C2, and also, suffers from large local read buffers C2 (cf. Section 5.2). Therefore, we seek for a simple yet efficient solution to circumvent those limitations.

In this section, we introduce the concept of block-based partitioning to optimize the performance of a distributed inverted list index under memory disaggregation and discuss how this scheme overcomes the aforementioned problems.

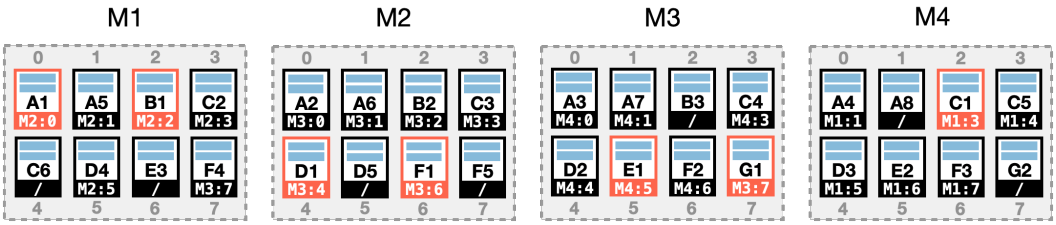


Fig. 3. Block-based partitioning: a pointer to a subsequent block is stored at the end of a block, red blocks indicate the beginning of a list, and pointers to red blocks are stored in the catalog.

6.1 Block-based Partitioning

The intuition of block-based partitioning is to divide the lists into fixed-size chunks, further called *blocks*, and distribute them uniformly across all memory nodes. Blocks that belong to the same list are connected via (remote) pointers as a singly linked list. Figure 3 illustrates block-based partitioning. The fine granularity of the partitioning scheme has several advantages with respect to traditional methods. Next, we describe how lists are processed in the block-based distribution scheme, the structure of a worker's local buffer, and discuss block prefetching.

Fetching and processing the lists for query processing in BlockBased works as follows: First, we RDMA_READ the first block of each list that is involved in the query from remote memory into the first row of the local buffer. The local buffer of a worker can be considered an array of blocks with two rows as shown in Figure 4. The first row stores one block per query term and the second row is used to prefetch linked blocks. The remote address of the first block of a list is stored in a catalog once per compute node, similar as for TermBased. However, the size of the catalog is smaller: the list's end addresses are not required because the blocks are fixed-size. Hence, the catalog size is the size of the universe times 8 Byte (the size of a remote pointer), i.e., $O(|U|)$. The remote pointer encodes the address that points to the block on the memory node with 48 bits and the memory node with the remaining 16 bits.

If the first row is ready (i.e., a completion event has been polled), we visit the blocks (that have been copied to our local buffer) with non-null remote pointers to post RDMA_READs for prefetching succeeding blocks. A local block of the second row is passed as a target buffer, the destination to which the NIC copies the data read from remote memory. A null pointer indicates the end of a list.

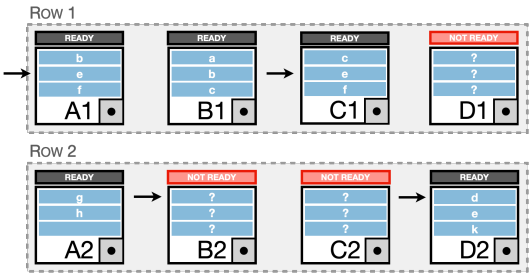


Fig. 4. Local buffer of a worker.

Next, we can start with the operation (e.g., k-way intersection). As soon as we hit the end of a block that is not marked as the end of the list, we move to the next (prefetched) block in the local buffer (e.g., from B1 to B2 in Figure 4). If the data in the next block has not been read yet, we stall for the completion event. Otherwise, i.e., the block is ready (for instance, D2), we again first check the remote pointer to prefetch the next block by reusing the previously processed block (D1) in the local buffer. We keep processing until the operation has been completed.

6.2 Addressing Scalability Challenges

Next, we describe how BlockBased improves the scalability limitations C1–C4 of the traditional methods.

C1 ***Dealing with Skewed Accesses.*** In contrast to term-based partitioning, skewed workloads do not overload individual memory nodes due to the fine granularity of the blocks with size $\boldsymbol{B}$. A list of size $|\boldsymbol{L}|$ is partitioned into $\lceil |\boldsymbol{L}|/\boldsymbol{B} \rceil$ blocks that are distributed round-robin to the memory nodes (cf. Figure 3). Reaching a list that consists of at least $|\boldsymbol{M}|$ blocks will involve all memory nodes, which balances the network traffic uniformly as shown in Figure 2a.

C2 ***Reducing the Local Buffer Size.*** Each worker needs some free resources on the compute node, to which the NIC can copy the data that is read from the memory nodes. For TermBased and a query q, the local buffer of a worker must fit all $|q|$ lists to compute the operation (e.g., intersection) efficiently, i.e., the space complexity is $O(|q| \cdot |\boldsymbol{D}|)$ in the worst case, where $|\boldsymbol{D}|$ is the number of documents in the database. This becomes a severe problem if the lists are sufficiently large because memory is limited on the compute nodes. For DocumentBased, this is slightly better because the lists are partitioned by the number of memory nodes, hence $O(|q| \cdot |\boldsymbol{D}|/|\boldsymbol{M}|)$.

To deal with space limitations, a common approach is to read and process only two lists at a time. Unfortunately, this work-around comes at the cost of increased runtime complexity for list operations: The k-way intersection is replaced by a sequence of 2-way intersections and the union operator must repeatedly scan increasing intermediate result lists to obtain the final result. To efficiently support all index operations (cf. Section 4.2), all k lists must be (partially) available, leading to large local buffer sizes for TermBased and DocumentBased.

BlockBased innately complies with this objective and requires only $2 \cdot |q| \cdot \boldsymbol{B}$ space per worker, where $\boldsymbol{B}$ is the block size. Hence, the local buffer size does not depend on the list size and the space complexity reduces to $O(|q|)$. This is an important feature for the disaggregated memory setting as the index can grow arbitrarily large without requiring more memory on the compute nodes.

Example 6.1. Consider an average list size of 100 MB and assume that each worker issues queries of at most size 6. 32 TermBased workers require a total local buffer size of 19.2 GB, for DocumentBased we need at least 4.8 GB (with $|\boldsymbol{M}| = 4$; no prefetching), and for BlockBased only 384 KB (with $\boldsymbol{B} = 1$ KB) of memory on the compute node is required. We discuss the choice of a useful block size $\boldsymbol{B}$ in Section 6.3.

C3 ***Masking the Network Latency.*** Another important benefit of processing lists in block is that the operation (e.g., computing the intersection) is interleaved with the network communication (reading lists from remote memory). In other words, the network latency is masked by computing list operations, which significantly improves performance for long lists. Unfortunately, this is difficult to implement for TermBased (and DocumentBased) because the completion event is only generated when a (partial) list has been read and copied entirely, i.e., during the read process, we do not have progress information that allows us to operate on partial lists. A work-around would be to split the `RDMA_READ` operations into multiple smaller ones and then carefully check which parts of the lists have been already read, which somehow resembles the BlockBased approach but yet does not solve C1 for TermBased and C4 for DocumentBased.

C4 ***Minimizing Network Roundtrips.*** More roundtrips result in a higher network load, i.e., more contention on the NIC. Therefore, reducing network roundtrips may significantly increase the throughput. In particular, when reading small data fragments from remote memory, the latency negatively impacts the overall throughput. Applied to our scenario, the number of roundtrips must kept low, especially, when the lists are short. TermBased requires only $|q|$ (the number of terms in

a query q) roundtrips, i.e., one roundtrip per list, which is minimal. In contrast, DocumentBased needs $|q| \cdot |\boldsymbol{M}|$ roundtrips, where $|\boldsymbol{M}|$ is the number of memory nodes. The negative impact of $|\boldsymbol{M}|$ times more roundtrips on short lists is visible in Figure 2b ($|\boldsymbol{M}| = 4$). BlockBased with a sufficiently small block size solves this challenge (resulting in good scalability) because the number of roundtrips depends on the list sizes, i.e., for short lists ($\leq \boldsymbol{B}$), only a single roundtrip per query term is required to retrieve the entire list. For long lists, BlockBased masks the network latency by prefetching subsequent blocks C3. We discuss the choice of the block size in the following section.

6.3 Discussion

Traditional distribution schemes introduce severe scalability limitations and cannot deal with skewed accessed C1, require large memory resources on the compute nodes C2, do not mask network latency sufficiently C3, or involve too many network roundtrips for reading lists from remote memory C4. The BlockBased approach for memory disaggregation partitions the lists into fine-granular chunks that are distributed across the memory nodes and successfully addresses the aforementioned challenges. In particular, BlockBased combines the benefits of existing techniques while avoiding their drawbacks under memory disaggregation.

Implications of the Block Size. Large blocks lead to a better network utilization because RDMA verbs saturate the network bandwidth for message sizes greater than 2 KB [6]. The downside of large blocks is the unused space at the end of each list that increases the space requirements on the memory nodes. This effect is particularly pronounced for short lists that cannot fill an entire block. For example, the CCNEWS dataset consists of approximately 29M lists with many short lists containing only a few items. On the memory nodes, we need in total at least 29M blocks, i.e., 29M times $\boldsymbol{B}$ of memory. With $\boldsymbol{B}$ = 4 KB, the lists are over-provisioned and at least 116 GB of main memory is required (accumulated over all memory nodes), although the actual index size of CCNEWS is only 71 GB. Moreover, reading sparsely populated blocks over the network increases the network latency since superfluous data is transferred. The effect is shown in Figure 2b: BlockBased with 1 KB blocks does not saturate the network, while BlockBased_{512} with 512 Byte blocks achieves a throughput close to the theoretical bandwidth bound for 160 compute threads. Small blocks improve memory utilization but lead to more network roundtrips if the lists are large. Hence, the choice of the block size is dataset-dependent. Nonetheless, in our evaluation on various real-world datasets, we found that 1 KB blocks are a good trade-off between memory usage and network performance.

7 INDEX UPDATES

In this section, we address challenge C5 and propose efficient index updates for the BlockBased indexing scheme.

Updates in inverted list indexes have received little attention in previous work [8, 9, 33, 64]. To reduce the number of list accesses, some works propose to collect updates and apply them in batches. In this work, we aim for incremental list updates such that the updates are immediately visible. We distinguish read and update queries: Read queries retrieve one or multiple lists, for example, for an intersection query. Update queries modify the index structure and either insert or delete items to/from inverted lists (cf. Section 4.1).

Next, we discuss an optimistic concurrency control scheme for parallel updates, develop a data layout for the index blocks, propose an algorithm for index updates, and discuss read-write conflicts.

Concurrency Control. We expect a reading-heavy query load for our inverted list index, therefore, a high read performance is inevitable. In a pessimistic scheme, the reader locks (e.g., with a shared lock) the block such that no updater can modify the items that the reader is currently

accessing. In optimistic schemes, readers optimistically assume that no update has occurred and detect modifications with a specified verification mechanism. Previous work has shown that (with RDMA) optimistic concurrency schemes scale better than pessimistic ones [56, 62, 65] because RDMA atomic bottlenecks are avoided. We stick to an optimistic concurrency scheme due to higher scalability and implement lock-free read queries with verification to resolve read-write conflicts. We apply *block-grained* write locks for update queries to resolve write-write conflicts.

7.1 Block Layout

To support update queries, we enhance the block layout introduced in Section 6. In addition to (1) the data items (document identifiers), (2) the remote pointer to the next block (or a null pointer), (3) a verification mechanism to detect corrupted data, (4) a lock flag to acquire write locks, and (5) a *tag* to mitigate the ABA problem is necessary. We add a version to each cache-line for verification and encode in the last 8 Byte a lock bit for write locks and a block tag (`b_tag`) to detect hazardous deallocations. Verification with cache-line versioning and tagging is discussed in Section 7.3. The 8 Byte remote pointer encodes the memory node, the offset in remote memory, and a pointer tag (`p_tag`) that must match the block's `b_tag` to which the pointer points to. Note that 38 bits are sufficient for the offset because we address 1 KB blocks ($2^{48}/2^{10}$). Figure 5 outlines the block layout.

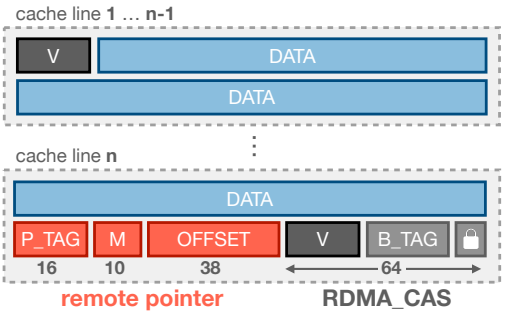


Fig. 5. Layout of a block.

7.2 Update Queries

Update queries use a fine-grained block-based locking mechanism to resolve conflicts. An insert (delete) query inserts (deletes) a given item (e.g., a document identifier) into (from) a list. For instance, in information retrieval, a document (consisting of multiple terms) may be inserted into the index. Then, the document identifier is inserted into all lists that represent the terms contained in the document. In a follower-graph, a user id is inserted only into a single list (i.e., the user follows another user). For consistency, an update query only modifies one list at a time. Hence, a query that is expected to update multiple lists, is split into multiple (sub-) queries.

To update a list, the target block in the sort order of the list must be found, which is done by scanning the list. Once the target block has been identified, the block is locked.

Remote Write Locks. Acquiring remote write locks is implemented as follows: Assume that the block the updater wants to lock has already been copied by the NIC from remote memory into the local buffer of the worker. Next, the block must be verified to guarantee that no concurrent updater has corrupted the block while reading from remote memory. We describe verification in Section 7.3 and assume for now that verification has been successful. We implement a spin lock using the atomic `RDMA_CAS` verb, which compares an 8 Byte word to an address in remote memory for equality and atomically swaps the remote value with a given input value if the comparison succeeds. To acquire a lock on a block, the updater compares the last 8 Byte word of the local and the remote block (cf. Figure 5) and tries to swap the value of the word such that the lock bit is set. If the comparison is successful, the updater obtains the lock, otherwise the writer re-reads the block from remote memory and tries again. Note that it is not sufficient to compare the lock bit in isolation: (1) the version could have changed, i.e., another updater modified and already unlocked

Algorithm 1: `insert(item, term)`

Input: The `item` to insert and the `term` representing the list.

```
B ← find_and_lock_block(item, term)
// split block
if B.size() + 1 exceeds capacity then
    memory_node ← Choose random memory node
    B_new, r_addr ← remote_allocate(memory_node)
    Divide B.items ∪ {item} uniformly and in order
    // adjust pointers, increase cache-line versions, and write block
    B_new.next_ptr ← B.next_ptr
    B.next_ptr ← [B_new.b_tag; memory_node; r_addr]
    B_new.increase_cacheline_versions()
    RDMA_WRITE(B_new)
else
    Insert item into B and maintain ordering
B.increase_cacheline_versions()
B.WRITE_unlock()
```

the block just before we try to get the lock, or (2) the block tag has been incremented meanwhile, i.e., the block has been deallocated (and maybe even reused). In the latter case, the updater must restart the operation and start reading the list again. In Section 7.3, we show how versions and tags are used to prevent reading faulty blocks from remote memory.

Insert Queries. After a block has been locked, we can insert the new item into the block (with respect to the sort order) if there is enough space. Otherwise, the block must be split: we allocate a fresh block B_{new} from a randomly chosen memory node, evenly distribute the items among the two blocks, adjust the pointers accordingly, and write back B_{new} to remote memory. Remote memory allocation is described in Section 8. Note that no other worker has access to B_{new} at this time, thus there is no need to lock B_{new}. Finally, we increase the cache-line versions and unlock the block by resetting the lock bit and posting an `RDMA_WRITE` to write back the block to the memory node. Algorithm 1 summarizes the insertion query.

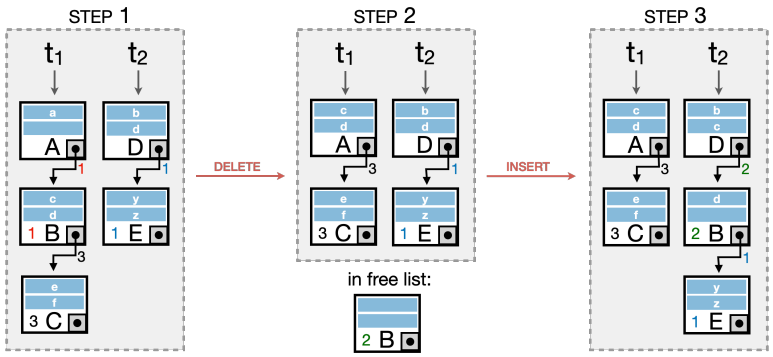


Fig. 6. Edge cases of update operations.

Example 7.1. Figure 6 (step 2 → step 3) shows an update query with block splitting. We want to insert item `c` into list t_2 and find block `D`, which is locked by the updater. Since there is no space

available in D, a new block B is allocated, and the items are split evenly between D and B preserving the order. The pointers are updated, the cache-line versions increased, and finally, the blocks are written back to remote memory.

Delete Queries. Assume that the writer has already obtained a write lock on block B that contains the item to be removed. The item is deleted and the order among the items is preserved. If block B is empty after the deletion, an additional step is required to avoid empty blocks. To remove a block from the linked list, we need to update the pointer of the predecessor to the successor of the deleted block. Since B only stores the pointer to its successor B_{next} but not to its predecessor, we read, verify, and lock B_{next}, move the items of B_{next} to the current block B, and deallocate B_{next}. Remote deallocation is discussed in Section 8. Finally, we increase the cache-line versions, reset the lock bit, and write back the block to remote memory. Algorithm 2 outlines the deletion operation.

Example 7.2. Consider Figure 6 (step 1 → step 2), where item a is removed from list t_1. First, we search the block containing the item (block A), lock the block, and delete the item. Block A is now empty: we read, verify, and lock the next block (B) and move the items from B to A. The remote pointer of A (cf. Figure 5) is replaced with that of B, and B can be safely deallocated (since it is locked, no other worker has write access to B). The lock bit of block A is cleared and A is written back to remote memory.

Algorithm 2: `delete(item, term)`

Input: The `item` to delete and the `term` representing the list.

```
1  B ← find_and_lock_block(item, term)
2  Delete item in B and maintain ordering
   // merge blocks
3  if B.is_empty() then
4      B_next ← read_and_verify(B.next_ptr)
5      B_next.lock()
6      B.items ← B_next.items
       // adjust pointer, increment tag, and deallocate empty block
7      B.next_ptr ← B_next.next_ptr
8      B_next.b_tag ← B_next.b_tag + 1
9      remote_deallocate(B_next)
10 B.increase_cacheline_versions()
11 B.WRITE_unlock()
```

7.3 Detecting Read-Write Conflicts

Write-write conflicts are resolved with block-grained write locks, i.e., an updater cannot interfere other updaters. Reads do not use any locks. Instead, read-write conflicts are resolved with verification and retries to detect data corruption and faulty blocks. A read query optimistically assumes that the blocks are not updated. However, if a concurrent modification occurs, data is potentially corrupted. To guarantee correctness, we implement two verification mechanisms, block verification and reusage detection, discussed in detailed below. The read and verification routine is shown in Algorithm 3.

Block Verification. When a reader issues an RDMA_READ, a block is copied from remote memory to the worker's local buffer. Subsequently, the block is verified to detect whether a concurrent updater has modified the block while the NIC was reading the block from remote memory. A 4 Byte

Algorithm 3: `read_and_verify(r_ptr)`

Input: The remote pointer `r_ptr` to verify.
Output: The read and verified block in the local buffer.

```
1 B ← Get address of local buffer
2 do
3 |   B ← RDMA_READ(r_ptr)
4 |   if B.b_tag != r_ptr.p_tag then Restart operation
5 while !B.verify_cachelines() or B.is_locked()
6 return B
```

version number is added to each cache-line in the block, similarly to the proposal by Dragojevic et al. [20] (cf. Figure 5). An updater first locks a block, modifies the data (e.g,. inserts a new value), increases *all* cache-line versions, and finally, unlocks the block. A reader query detects corrupted data by comparing the cache-line versions. If all versions are equal, no modification has been occurred and the data is correct, otherwise, the block must be re-read from remote memory.

Note that we need to version each cache-line. It is not sufficient to use version numbers at the beginning and at the end of a block. The PCIe specification allows all cache-lines of a memory read request to be handled concurrently and there is no guarantee on the processing order [62]. Hence, data may be reordered at cache-line granularity (within an `RDMA_READ` operation) such that the begin- and end-versions of a block may match, although an updater might have modified data in between.

Reusage Detection. In addition to data corruption within a block, we must check whether the block we read is indeed the expected block. To detect reused blocks and mitigate the ABA problem, we use block and pointer tagging. Pointer tagging is a well-known technique in many concurrent data structures [24]. Applied to our index structure, we associate a tag with a block (`b_tag`) and a pointer tag (`p_tag`) to a pointer to that block (cf. Figure 5). When a new block is read, we post an `RDMA_READ` to fetch the block and also pass the `p_tag` as the expected tag for verification. Once the block has been copied from remote memory into the local buffer, we check whether the expected `p_tag` and the `b_tag` match (cf. line 4 in Algorithm 3). In the case of a mismatch, the block has been deallocated – on deallocation the block tag is increased – and possibly even reused, i.e., the block may be part of another list. Then, the read operation is aborted and must be restarted.

Example 7.3. Consider Figure 6 and the following scenario: A worker W_1 starts reading list t_1 and fetches the first block `A` into its local buffer; assume that verification succeeds. To prefetch the next block, W_1 posts an `RDMA_READ` using the remote pointer stored in `A`. Next, an updater W_2 deletes item `a` in t_1, deallocates block `B`, and increases the `b_tag` (cf. line 8 in Algorithm 2). The intermediate result is shown in Figure 6 (step 2). Another updater W_3 inserts item `c` into t_2. Since there is no space available in block `D`, a new block is allocated, and block `B` is reused. Note that the block tag (`b_tag`) of `B` is now 2. Also, this tag is installed as `p_tag` in the remote pointer of `D` (line 7 in Algorithm 1). Assume that the prefetching operation of W_1 has completed and block `B` is read into its local buffer. However, block `B` is meanwhile part of another list, which W_1 is able to detect by comparing the expected pointer tag (`p_tag` = 1) of `A` against the block tag (`b_tag` = 2) of `B`. Since they do not match, W_1 restarts the operation (i.e., starts scanning list t_1 from the beginning).

7.4 Fault Tolerance

A compute node that crashes while updating the index might leave a block in a locked state, which must be detected for proper failure handling. For example, assume an updater W_1 wants

to modify a block X, acquires a write lock on X, and subsequently crashes. Another worker W_2 (i.e., reader or updater) reads block X, detects that X is locked, and thus must retry. For failure detection, W_2 additionally stores the cache-line version of block X (which is to be increased by W_1 after successfully modifying X). If the cache-line version remains unchanged over several retries (mimicking a timeout), W_1 did not make any progress and a failure is assumed. After detection, failure handling must be initiated at the system level (e.g., the database layer).

Failure handling, replication protocols, and recovery are beyond the scope of this work, and we refer to RDMA-based strategies proposed in the literature, e.g., [29, 37, 52].

8 MEMORY MANAGEMENT

In this section, we describe how the space of a memory node is initialized and managed by the compute nodes. The space of a memory node consists of three parts: (1) the pre-allocated data blocks that store the partial lists, (2) a free list for memory allocation, and (3) a catalog that stores the entry points of the lists that are located on this memory node as shown in Figure 7.

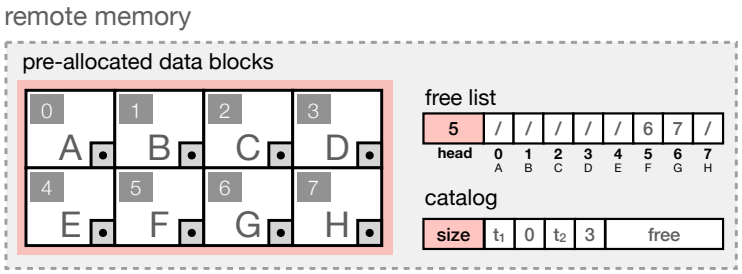


Fig. 7. Layout of a memory node.

8.1 Free List

A free list keeps track of available blocks on the memory node. The free list is implemented as a lock-free fixed-size stack consisting of β many 4 Byte entries (where β is the number of data blocks). The i-th entry of the free list represents the i-th data block in the pre-allocated fixed-size buffer. A head points to the first free block in the free list. Each entry in the free list points to the next free block on the stack; a null value indicates the end of the list. Figure 7 shows the free list initialized with F as the first available block. Note that the physical size of the free list remains fixed and is not affected by (de-)allocations because pointers are only swapped logically.

Allocation. If a worker on a compute node allocates a new block, it first RDMA_READs the entries of head (e.g., 5 in Figure 7) and head_next (6) – the value of `list[head]`. Then, the worker tries to atomically swap head with head_next (by using an RDMA_CAS). The value of head is the block position in the data buffer, i.e., the offset from the start address of the memory region is head $\cdot \boldsymbol{B}$, where $\boldsymbol{B}$ is the block size. Subsequently, `list[head]` is set to null. If the value of head is null, no more space is left on the memory node.

Deallocation. For deallocating a block with offset α on memory node $\boldsymbol{M}$, a worker RDMA_READs head and RDMA_WRITEs the value of head to `list[`α`]` – the free list of $\boldsymbol{M}$. Then, the worker tries to compare-and-swap head with α. We repeat the whole process until the RDMA_CAS succeeds. For example, if a worker wants to deallocate B (cf. Figure 7), it writes 5 (the current value of head) to position 1 in the free list, and swaps head with 1. Note that only a single worker at a time can deallocate a specific block since the block is locked (cf. Algorithm 2 line 5).

Optimizations. We found that on high contention on head, i.e., many workers try to (de-)allocate, the performance significantly degrades. We recommend to relax the sequential specification of the free list (the concurrent stack) [24] by introducing k heads, or in other words, using an array of k free lists. When accessing the free list, one of the k heads is chosen uniformly at random, which significantly decreases contention. In our experiments, we set $k = 16$, resulting in $(k-1)\cdot$ 8 Byte space overhead per memory node (8 Byte because RDMA_CAS operates on 8 Byte values only).

8.2 Catalog Updates

Each compute node stores a catalog (i.e., a hash map) that maps each term t_i to the *beginning block* of its list on the memory node (using a remote pointer). A serialized version of the partial catalogs is stored contiguously on the memory nodes as shown in Figure 7. The catalog consists of a size field (the number of beginning blocks), and size many (term, offset)-tuples. A new compute node fetches the size of the catalog from each memory node, reads the serialized catalogs, and inserts the values into its local hash map: the remote pointers are constructed by concatenating the block offset with the memory node (p_tag remains zero for beginning blocks). In addition, the size field for each memory node is stored locally on the compute node to enable lazy catalog updates.

Extending the Catalog. An update query may attempt to insert into a list that is not known to the local catalog, for instance, when a new term occurs in a document. First, we check whether another worker has already created the list by refreshing the catalog (as described below). If the list does not exist, we try to atomically increase the size field of the serialized catalog using an RDMA_CAS. If the compare-and-swap fails, we restart the catalog update procedure. Since we can perform a remote atomic operation only on a single memory node, we globally fix the memory node at which we insert a specific new term by using a hash function shared among all compute nodes. This will avoid that another concurrent worker creates a list for the same term on a different memory node resulting in different remote pointers for the same term.

Lazy Catalog Refresh. Before we insert a new term or read a list where the beginning address is not available in the catalog, we must refresh the local catalog. The relevant catalog entries are placed on a specific memory node, which is determined by a global hash function. To avoid retrieving the entire catalog, we compare the locally stored size field against the remote value on the memory node. If the values are not equal, we only RDMA_READ the difference, i.e., the last $\texttt{size}_{\text{new}}$ - $\texttt{size}_{\text{old}}$ (term, offset)-tuples stored in the serialized catalog, which we insert into our local catalog.

Space Requirements. The overall size of the catalog is 2×4 Byte per pre-allocated block plus the space of the size field (8 Byte due to RDMA_CAS). The total size of the free list is 4 Byte per pre-allocated data block plus the space for the head(s). Note that we use 4 Byte for the block offsets, which is sufficient to address 4 TB of main memory (per memory node) with 1 KB blocks. On a memory node with S Bytes available, we can pre-allocate $\beta = \lfloor (S - (8 + 8k))/(\boldsymbol{B} + 4 + 2 \cdot 4) \rfloor$ blocks (k is the number of free list heads). For a block size of 1 KB ($\boldsymbol{B} = 2^{10}$), the meta data (catalog and free list) occupies less than 1.5% of the memory resources on a memory node.

9 EXPERIMENTAL EVALUATION

We analyze the different index designs using real-world and synthetic datasets, and evaluate the performance and the scalability for read-only and mixed (read+write) workloads, respectively. Our code is open source and publicly available[1].

[1]https://github.com/DatabaseGroup/rdma-inverted-index

Experimental Setup. All experiments were conducted on a 9-node cluster with five compute nodes and four memory nodes. Each compute node has two Intel Xeon E5-2630 v3 2.40 GHz processors with 16 cores (8 cores each; hyperthreading enabled) and the memory nodes have two physical Intel Xeon E5-2603 v4 1.70 GHz processors with 12 cores (6 cores each). All machines run Debian 10 Buster with a Linux 4.19 kernel, and are equipped with 96 GB of main memory and a Mellanox ConnectX-3 NIC connected to a 18-port SX6018/U1 InfiniBand switch (FDR 56 Gbps). The NIC is installed on a single NUMA socket, which inevitably leads to NUMA effects [47]. Therefore, we allocate the memory on the socket of the NIC (using `numactl`[2] and `mmap`[3] with hugepages to reduce address translation cache misses on the NIC [20]) and alternately assign worker threads to both NUMA sockets on the compute nodes following Ziegler et al. [62]. To model a disaggregated memory scenario, we limit a compute node's memory size to 10 GB (shared among all compute threads) and assign only a single CPU core to each memory node (similar to previous works [38, 56]). To avoid queue pair thrashing on the NIC [20], we share 8 queue pairs among the workers per compute node.

Baselines. We compare BlockBased with 1 KB blocks against two baselines, TermBased and DocumentBased, which are implemented as described in Section 5. To leverage efficient list operations, all methods operate on $|q|$ lists at a time, where $|q|$ is the query size.

Datasets & Workloads. We empirically evaluate TermBased, DocumentBased, and BlockBased on three real-world datasets from different domains, i.e., information retrieval (CCNEWS), query processing (SSB), and graph analytics (TWITTER), and one synthetic dataset with a uniform distribution of terms and documents (TOY).

Table 1. Dataset and query characteristics.

		CCNEWS	TWITTER	SSB	TOY
Datasets	Raw index size	71 GB	7.7 GB	201 MB	764 MB
	Number of documents	43,495,426	49,395,940	6,001,216	2,000,000
	Number of terms	29,497,691	43,983,853	29	100,000
Queries	List size in query (avg)	7,076,710	60,188	1,812,920	2000
	List size in query (max)	42,257,759	779,958	6,001,215	2218
	Query size (avg)	6	5	3	5

CCNEWS is a collection of English news articles crawled from January 2016 to March 2018 [41] and a standard benchmark for information retrieval. An article consists of terms (words) on which we build the inverted list index. The queries are predefined and contain real queries selected from different crowd workers [41].

TWITTER is a widely used dataset in graph analytics and consists of 53M vertices (users) and 2 billion edges (*follows-a* relationship) [16]. A list in the dataset represents an adjacency list of a vertex. Since there are no queries available, we sample them uniformly at random. We generate queries using the k most popular users in terms of followers (k=1000) drawn uniformly at random.

SSB is a star schema benchmark developed for performance measurements of data warehousing applications [48]. The benchmark incorporates a single fact table (`LINEORDER`) along with four dimension tables (`CUSTOMER`, `SUPPLIER`, `PART`, and `DATE`). We join the tables and build an inverted list index using the evaluated predicates of the queries as discussed in Section 2. The predefined

[2]https://linux.die.net/man/8/numactl
[3]https://linux.die.net/man/2/mmap

queries are based on the well-known TPC-H benchmark with different selectivities. We use the queries: Q1.1, Q1.2, Q1.3, Q2.1, Q2.2, Q2.3, Q3.1, Q3.2, and the query of Wang et al. [54], which are intersection queries using one or more AND operators in the WHERE clause.

TOY is a synthetic dataset that follows a uniform distribution. We sampled 2M documents with an avg. size of 100, and 100k different terms distributed uniformly at random.

Table 1 summarizes the dataset and query characteristics. *Raw index size* shows the memory footprint of the inverted index; *Number of documents* and *terms* is the total number of indexed documents and the number of lists, respectively. For SSB, there are only 29 lists as only lists for query terms are built (cf. small raw index size). *List size in query* is the avg. (max.) size of all queried lists, and *Query size* is the avg. number of terms per query.

9.1 Read-Only Performance

In this section, we evaluate the throughput over the number of compute threads (5–160) in queries per seconds [q/s] for read-only workloads with a focus on the computation-heavy intersection operation. Intersection is a key operation of inverted list indexes and computationally more challenging than plain reads. Figure 8 shows the throughput results for all indexes and datasets. Notably, we observe a high variance in absolute numbers due to varying list sizes for different datasets (i.e., intersection of long lists is more expensive than for short lists).

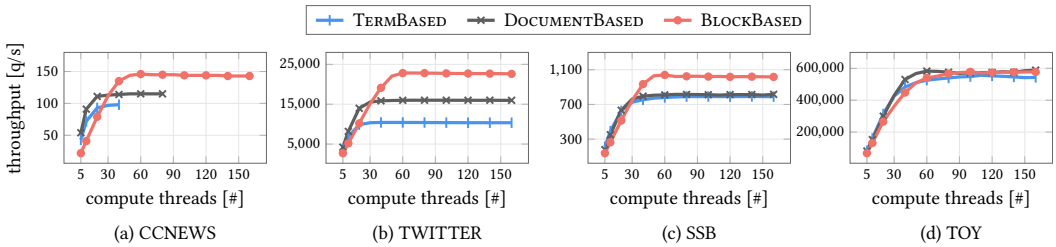


Fig. 8. Throughput over the number of compute threads for read-only workloads (intersection).

For CCNEWS, DocumentBased slightly outperforms TermBased, which we attribute to the long lists (up to 170 MB per list). Network communication interleaves with the (partial) list operation because DocumentBased divides every list into four chunks (one per memory node), resulting in higher overall throughput (cf. Section 5.2). However, large local buffers cause both approaches to run out of memory for > 80 compute threads. Each compute thread has its own local buffer whose size depends on the list sizes. Because we only read a fraction of $2/|M|$ per list (involved in the query) from remote memory into local buffers, DocumentBased can utilize up to 80 compute threads before running out of memory. The factor of 2 in the list fraction is due to the prefetching of subsequent chunks of the lists to mask the network latency (cf. Section 5.2). In contrast, the avg. memory footprint of BlockBased's local buffers for 160 compute threads is 384 KB per compute node. BlockBased's fine-grained block partitioning also improves network latency masking, resulting in a 1.3× higher throughput compared to DocumentBased for 80 compute threads. For up to 20 compute threads, the throughput of DocumentBased and TermBased increases fast and is about 1.4× higher than the throughput of BlockBased. This is a consequence of using small (1 KB) blocks, which results in more network roundtrips and contention on the NIC. However, the throughput of both DocumentBased and TermBased reaches a plateau for >30 compute threads, whereas the throughput of BlockBased scales up to 60 compute threads, reaching its max. throughput (≈145 queries per second) for ≥60 compute threads. Experiments with larger (2 KB)

blocks reveal that the throughput of BlockBased increases at a similar rate as DocumentBased while retaining the higher overall throughput. Nonetheless, we recommend small blocks to achieve high throughput for reading short lists (cf. Section 6).

Despite considerably shorter lists (≈240 KB on avg.), the performance gap between DocumentBased and TermBased is larger for TWITTER. TermBased suffers from heavily skewed list accesses (M_1 stores >50% of the queried lists), effectively underutilizing the network. BlockBased is not susceptible to access skew and significantly outperforms the other approaches. We observe a similar behavior for SSB, but the performance gap between DocumentBased and TermBased is smaller as the majority of list accesses is uniformly distributed among all memory nodes. For TOY and its short lists (8 KB on avg.), the throughput of the three approaches is comparable and network latency masking becomes negligible.

Summarizing, BlockBased outperforms both DocumentBased and TermBased for >60 compute threads on all instances. The fine-grained block partitioning interleaves remote memory accesses and list operations, is not susceptible to access skew, and requires fewer network roundtrips. It offers efficient list operations with a small memory footprint at the compute nodes, whereas DocumentBased and TermBased quickly run out of memory, e.g., for CCNEWS.

9.2 Update Performance

In this section, we analyze the update performance of BlockBased with a focus on the insert operation (cf. Section 7). We analyze the throughput and the scalability for mixed workloads with a varying ratio of read/insert queries (up to 50% insert queries), and we evaluate how contention on a few lists impacts the performance of read and update queries, respectively.

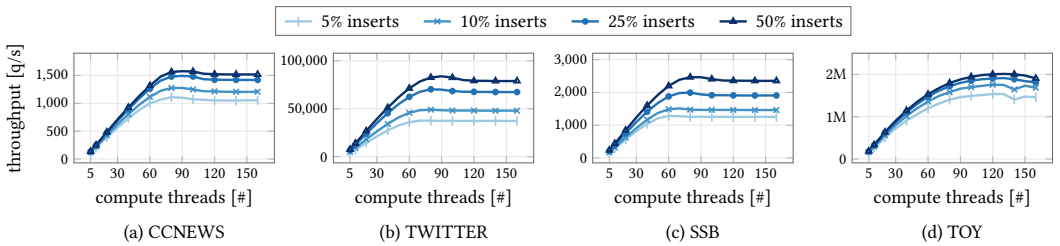
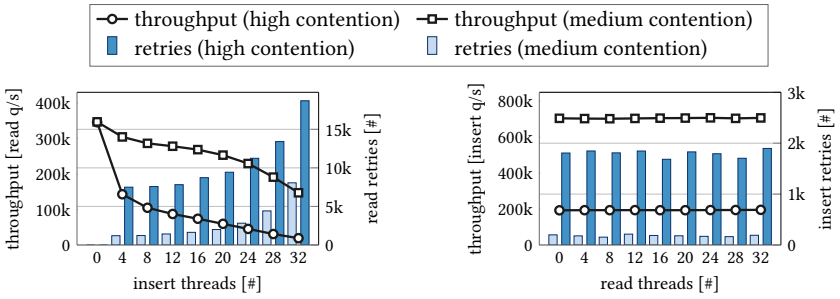


Fig. 9. Throughput of BlockBased over the number of compute threads for mixed workloads (intersection).

For each dataset, we use 5% of the documents as insert queries (drawn uniformly at random). The index is then built from the remaining 95% documents: All blocks are full and splits are likely to happen. For example, we observe a block is split for more than 98% and 93% for CCNEWS and SSB, respectively. To account for large differences in the number of terms per document among our datasets, we split each query into single-term subqueries, each of which modifies exactly one list at a time. For TWITTER, we insert the k=1000 most popular terms to model a common scenario: users follow other users with many followers. We evaluate different read/insert ratios: $q \in \{5\%, 10\%, 25\%, 50\%\}$.

Figure 9 shows the overall throughput (i.e., the number of mixed queries per second) over the number of compute threads. At first, it may be counterintuitive that we observe a higher throughput for an increasing ratio of insert queries. However, recall that a read query (1) accesses multiple lists from remote memory and (2) computes the intersection of these lists, which is often more expensive than updating a single list. This effect is particularly evident for datasets with long lists (e.g., CCNEWS): The throughput increases up to 9× compared to the read-only workloads (cf. Figure 8a). Overall, the update performance of BlockBased scales well in all instances.

Reads versus Updates. We use the TOY dataset to study the interference of reads and updates. To this end, we measure the throughput individually and collect additional statistics like the number of read retries (i.e., failed verifications). We use a fixed number of two compute nodes (16 threads per node, i.e., 32 threads overall) to execute read and update queries while increasing the number of updates or reads, respectively.



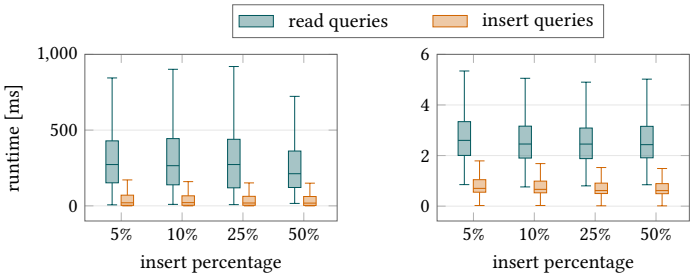
(a) Read throughput over increasing insert threads

(b) Insert throughput over increasing read threads

Fig. 10. Impact of read/insert interference.

Figure 10a depicts the read throughput (left y-axis) and the number of read retries (right y-axis) over an increasing number of insert threads, respectively. To emulate medium and high contention, the 32 read threads execute queries that involve 10 and 100 lists, respectively. We use z insert threads to interfere with read threads, i.e., they update the same lists. The remaining $(32 - z)$ insert threads update only lists that are not touched by read threads to reduce network noise (fewer threads lead to less contention on the NIC, which potentially reduces latency). Under high contention (10 lists), the read throughput degrades sharply for 32 insert threads (up to 18× less throughput compared to no contention) because verification keeps failing and read retries occur frequently. For medium contention (100 lists), the read throughput decreases only by 2.3×.

Figure 10b shows the update performance over the number of reader threads. As expected, insert throughput is stable since read threads do not interfere with insert threads due to lock-free reads.



(a) CCNEWS

(b) TWITTER

Fig. 11. Runtime comparison of single read and insert queries under mixed workloads.

Figure 11 depicts the runtimes of read and insert queries for our real-world datasets under mixed workloads (different read/insert ratios). We observe a stable performance of BlockBased without being sensitive to contention, e.g., TWITTER has the highest contention among our datasets (read/insert from/to the k=1000 most popular terms) but BlockBased's runtimes remain stable.

9.3 Memory Efficiency

As disaggregated memory systems assume a limited amount of main memory, we benchmark the memory footprint on the compute nodes. The catalog is stored once per compute node and its size mainly depends on the universe size. Furthermore, each thread has a separate local buffer and we track the memory that is actually consumed. Table 2 depicts the results for all three approaches over all datasets, which confirm the memory efficiency of BlockBased.

Table 2. Memory footprint on compute nodes, i.e., catalog size (per node) and local buffer size (per thread).

		CCNEWS	TWITTER	SSB	TOY
Catalog	DocumentBased	1.4 GB	2.1 GB	1.3 KB	4.8 MB
	TermBased	472 MB	703 MB	464 B	1.6 MB
	BlockBased	236 MB	352 MB	232 B	800 KB
Buffers	DocumentBased	640 MB	8 MB	28 MB	41 KB
	TermBased	1.3 GB	8 MB	56 MB	82 KB
	BlockBased	63 KB	20 KB	8 KB	20 KB

As expected (cf. Sections 5/6), DocumentBased has the largest catalog and BlockBased consumes only half of the memory compared to TermBased. The size of the local buffer of BlockBased is fully decoupled from the dataset size and requires only a few KBs. Contrarily, TermBased and DocumentBased consume more memory, which also causes out-of-memory errors for CCNEWS.

10 RELATED WORK

Distributed Inverted Indexes. Term-based [3, 27, 51] and document-based partitioning [11, 18, 26] have been proposed as the two major partitioning schemes for distributed inverted list indexes. Subsequent work [15, 31, 44, 46, 50] addressed the network communication cost as the main bottle of both schemes (in Ethernet networks). We deploy term-based and document-based partitioning in the disaggregated memory setting and identify the root causes for low scalability. Furthermore, we propose a distributed inverted list index for memory disaggregation on fast, RDMA-enabled networks with a fine-granular partitioning scheme that divides lists into small blocks and spreads them uniformly across all memory nodes.

Index Structures for Memory Disaggregation. Recently, various specific index structures for memory disaggregation have gained much attention in literature. Wang et al. [56] introduce Sherman, a distributed write-optimized B^+-tree index under disaggregated memory with a focus on write-intensive workloads and on-chip memory of RDMA NICs to enable fast exclusive locks. Zuo et al. [65] propose RACE, an extendible hash index under memory disaggregation that supports lock-free accesses and remote resizing. Wang et al. [57] propose dLSM, an optimized LSM-tree for faster writes in disaggregated memory. Luo et al. [38] introduce SMART, an adaptive radix tree that reduces read and write amplifications over B^+-trees in memory disaggregation to further improve performance. Optimistic concurrency control has become the standard for designing all of these index structures (incl. ours). In addition to lookup/insert/delete, inverted list indexes are required to compute the intersection or union of multiple lists. To the best of our knowledge, we are the first to develop an efficient distributed inverted list index structure under memory disaggregation.

Optimizations & Related Problems. List compression has been used to trade additional decompression time to reduce memory usage [54, 59]. This orthogonal optimization can be applied on top of our block-based partitioning: Divide the lists into blocks after compression and compute

threads decompress a block once it is in their local buffer. For example, Simple8b [2] is a fixed-size compression scheme (multiple integers are pooled into 64-bit words) that can be integrated into BlockBased without substantial changes.

Ranking is a related problem that is common in information retrieval [10, 55]: Document identifiers in list $\boldsymbol{L}_i$ are augmented with the frequency of term t_i in this document. A query returns only the k most promising documents based on the frequencies. BlockBased can be extended to support ranking: Read the augmented lists remotely and compute the result on the compute nodes.

11 CONCLUSION

We have shown that state-of-the-art partitioning schemes for distributed inverted list indexes are not suitable for the disaggregated memory setting. We proposed a novel distributed inverted list index that partitions lists into fixed-size blocks and distributes them across the memory nodes. Processing lists in blocks interleaves remote accesses with list operations, prevents compute nodes from running out of memory, and mitigates the effects of skew. We use optimistic concurrency control for efficient and consistent updates, and a verification mechanism to detect read-write conflicts. We empirically demonstrated the efficiency and scalability of our techniques on real-world datasets and different workloads.

ACKNOWLEDGMENTS

This research was funded in whole or in part by the Austrian Science Fund (FWF) P 34962. For open access purposes, the authors have applied a CC BY public copyright license to any author accepted manuscript version arising from this submission. This research was also partially funded by the Early Career Program at the University of Salzburg ID 35010271.

REFERENCES

[1] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. 2019. Designing Far Memory Data Structures: Think Outside the Box. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*. ACM, 120–126. https://doi.org/10.1145/3317550.3321433

[2] Vo Ngoc Anh and Alistair Moffat. 2010. Index compression using 64-bit words. *Softw. Pract. Exp.* 40, 2 (2010), 131–147. https://doi.org/10.1002/spe.948

[3] Claudine Santos Badue, Ricardo A. Baeza-Yates, Berthier A. Ribeiro-Neto, and Nivio Ziviani. 2001. Distributed Query Processing Using Partitioned Inverted Files. In *Eighth International Symposium on String Processing and Information Retrieval, SPIRE 2001, Laguna de San Rafael, Chile, November 13-15, 2001*, Gonzalo Navarro (Ed.). IEEE Computer Society, 10–20. https://doi.org/10.1109/SPIRE.2001.989733

[4] Ricardo A. Baeza-Yates, Carlos Castillo, Flavio Junqueira, Vassilis Plachouras, and Fabrizio Silvestri. 2007. Challenges on Distributed Web Retrieval. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis (Eds.). IEEE Computer Society, 6–20. https://doi.org/10.1109/ICDE.2007.367846

[5] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. 2003. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro* 23, 2 (2003), 22–28. https://doi.org/10.1109/MM.2003.1196112

[6] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *Proc. VLDB Endow.* 9, 7 (2016), 528–539. https://doi.org/10.14778/2904483.2904485

[7] Truls Amundsen Bjørklund, Nils Grimsmo, Johannes Gehrke, and Øystein Torbjørnsen. 2009. Inverted indexes vs. bitmap indexes in decision support systems. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*, David Wai-Lok Cheung, Il-Yeol Song, Wesley W. Chu, Xiaohua Hu, and Jimmy Lin (Eds.). ACM, 1509–1512. https://doi.org/10.1145/1645953.1646158

[8] Carolina Bonacic, Danilo Bustos, Veronica Gil-Costa, Mauricio Marín, and Victor Sepulveda. 2015. Multithreaded Processing in Dynamic Inverted Indexes for Web Search Engines. In *Proceedings of the 2015 Workshop on Large-Scale and Distributed System for Information Retrieval, LSDS-IR 2015, Melbourne, Australia, October 23, 2015*, Ismail Sengor Altingovde, Berkant Barla Cambazoglu, and Nicola Tonellotto (Eds.). ACM, 15–20. https://doi.org/10.1145/2809948.2809952

[9] Carolina Bonacic, Carlos García, Mauricio Marín, Manuel Prieto-Matías, and Francisco Tirado. 2010. Building efficient multi-threaded search nodes. In *Proceedings of the 19th ACM Conference on Information and Knowledge Management, CIKM 2010, Toronto, Ontario, Canada, October 26-30, 2010*, Jimmy X. Huang, Nick Koudas, Gareth J. F. Jones, Xindong Wu, Kevyn Collins-Thompson, and Aijun An (Eds.). ACM, 1249–1258. https://doi.org/10.1145/1871437.1871595

[10] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Y. Zien. 2003. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the 2003 ACM CIKM International Conference on Information and Knowledge Management, New Orleans, Louisiana, USA, November 2-8, 2003*. ACM, 426–434. https://doi.org/10.1145/956863.956944

[11] Brendon Cahoon, Kathryn S. McKinley, and Zhihong Lu. 2000. Evaluating the performance of distributed architectures for information retrieval using a variety of workloads. *ACM Trans. Inf. Syst.* 18, 1 (2000), 1–43. https://doi.org/10.1145/333135.333136

[12] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. 2021. Rethinking software runtimes for disaggregated memory. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, Tim Sherwood, Emery D. Berger, and Christos Kozyrakis (Eds.). ACM, 79–92. https://doi.org/10.1145/3445814.3446713

[13] Berkant Barla Cambazoglu and Ricardo Baeza-Yates. 2016. Scalability and Efficiency Challenges in Large-Scale Web Search Engines. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval, SIGIR 2016, Pisa, Italy, July 17-21, 2016*, Raffaele Perego, Fabrizio Sebastiani, Javed A. Aslam, Ian Ruthven, and Justin Zobel (Eds.). ACM, 1223–1226. https://doi.org/10.1145/2911451.2914808

[14] Berkant Barla Cambazoglu, Aytul Catal, and Cevdet Aykanat. 2006. Effect of Inverted Index Partitioning Schemes on Performance of Query Processing in Parallel Text Retrieval Systems. In *Computer and Information Sciences - ISCIS 2006, 21th International Symposium, Istanbul, Turkey, November 1-3, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4263)*, Albert Levi, Erkay Savas, Hüsnü Yenigün, Selim Balcisoy, and Yücel Saygin (Eds.). Springer, 717–725. https://doi.org/10.1007/11902140_75

[15] Berkant Barla Cambazoglu, Enver Kayaaslan, Simon Jonassen, and Cevdet Aykanat. 2013. A term-based inverted index partitioning model for efficient distributed query processing. *ACM Trans. Web* 7, 3 (2013), 15:1–15:23. https://doi.org/10.1145/2516633.2516637

[16] Meeyoung Cha, Hamed Haddadi, Fabrício Benevenuto, and P. Krishna Gummadi. 2010. Measuring User Influence in Twitter: The Million Follower Fallacy. In *Proceedings of the Fourth International Conference on Weblogs and Social Media, ICWSM 2010, Washington, DC, USA, May 23-26, 2010*, William W. Cohen and Samuel Gosling (Eds.). The AAAI Press. http://www.aaai.org/ocs/index.php/ICWSM/ICWSM10/paper/view/1538

[17] Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Søren B. Lassen, Philip Pronin, Sriram Sankar, Guanghao Shen, Gintaras Woss, Chao Yang, and Ning Zhang. 2013. Unicorn: A System for Searching the Social Graph. *Proc. VLDB Endow.* 6, 11 (2013), 1150–1161. https://doi.org/10.14778/2536222.2536239

[18] Owen de Kretser, Alistair Moffat, Tim Shimmin, and Justin Zobel. 1998. Methodologies for Distributed Information Retrieval. In *Proceedings of the 18th International Conference on Distributed Computing Systems, Amsterdam, The Netherlands, May 26-29, 1998*. IEEE Computer Society, 66–73. https://doi.org/10.1109/ICDCS.1998.679488

[19] Jeffrey Dean. 2009. Challenges in building large-scale information retrieval systems: invited talk. In *Proceedings of the Second International Conference on Web Search and Web Data Mining, WSDM 2009, Barcelona, Spain, February 9-11, 2009*, Ricardo Baeza-Yates, Paolo Boldi, Berthier A. Ribeiro-Neto, and Berkant Barla Cambazoglu (Eds.). ACM, 1. https://doi.org/10.1145/1498759.1498761

[20] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, Ratul Mahajan and Ion Stoica (Eds.). USENIX Association, 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%C4%87

[21] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, João Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network Requirements for Resource Disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 249–264. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gao

[22] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. 2016. RDMA over Commodity Ethernet at Scale. In *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016*, Marinho P. Barcellos, Jon Crowcroft, Amin Vahdat, and Sachin Katti (Eds.). ACM, 202–215. https://doi.org/10.1145/2934872.2934908

[23] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. 2019. Who limits the resource efficiency of my datacenter: an analysis of Alibaba datacenter traces. In *Proceedings of the International*

Symposium on Quality of Service, IWQoS 2019, Phoenix, AZ, USA, June 24-25, 2019. ACM, 39:1–39:10. https://doi.org/10.1145/3326285.3329074

[24] Andreas Haas, Thomas Hütter, Christoph M. Kirsch, Michael Lippautz, Mario Preishuber, and Ana Sokolova. 2015. Scal: A Benchmarking Suite for Concurrent Data Structures. In *Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 9466)*, Ahmed Bouajjani and Hugues Fauconnier (Eds.). Springer, 1–14. https://doi.org/10.1007/978-3-319-26850-7_1

[25] Md. E. Haque, Yong Hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. 2015. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). ACM, 161–175. https://doi.org/10.1145/2694344.2694384

[26] Donna Harman, Wayne McCoy, Robert Toense, and Gerald Candela. 1991. Prototyping a Distributed Information Retrieval System That Uses Statistical Ranking. *Inf. Process. Manage.* 27, 5 (sep 1991), 449–459. https://doi.org/10.1016/0306-4573(91)90062-Q

[27] Byeong-Soo Jeong and Edward Omiecinski. 1995. Inverted File Partitioning Schemes in Multiple Disk Systems. *IEEE Trans. Parallel Distributed Syst.* 6, 2 (1995), 142–153. https://doi.org/10.1109/71.342125

[28] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. 2014. String Similarity Joins: An Experimental Evaluation. *Proc. VLDB Endow.* 7, 8 (2014), 625–636. https://doi.org/10.14778/2732296.2732299

[29] Antonios Katsarakis, Vasilis Gavrielatos, M. R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 201–217. https://doi.org/10.1145/3373376.3378496

[30] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S. Milojicic, and Gustavo Alonso. 2022. Farview: Disaggregated Memory with Operator Off-loading for Database Engines. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org. https://www.cidrdb.org/cidr2022/papers/p11-korolija.pdf

[31] Tayfun Kucukyilmaz, Ata Turk, and Cevdet Aykanat. 2012. A Parallel Framework for In-Memory Construction of Term-Partitioned Inverted Indexes. *Comput. J.* 55, 11 (2012), 1317–1330. https://doi.org/10.1093/comjnl/bxr133

[32] Seung-Seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. MIND: In-Network Memory Management for Disaggregated Data Centers. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 488–504. https://doi.org/10.1145/3477132.3483561

[33] Nicholas Lester, Justin Zobel, and Hugh E. Williams. 2006. Efficient online index maintenance for contiguous inverted lists. *Inf. Process. Manag.* 42, 4 (2006), 916–933. https://doi.org/10.1016/j.ipm.2005.09.005

[34] Xiaolei Li, Jiawei Han, and Hector Gonzalez. 2004. High-Dimensional OLAP: A Minimal Cubing Approach. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer (Eds.). Morgan Kaufmann, 528–539. https://doi.org/10.1016/B978-012088469-8.50048-6

[35] Kevin T. Lim, Jichuan Chang, Trevor N. Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, Stephen W. Keckler and Luiz André Barroso (Eds.). ACM, 267–278. https://doi.org/10.1145/1555754.1555789

[36] Eric Lo, Ben Kao, Wai-Shing Ho, Sau Dan Lee, Chun Kit Chui, and David W. Cheung. 2008. OLAP on sequence data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, Jason Tsong-Li Wang (Ed.). ACM, 649–660. https://doi.org/10.1145/1376616.1376682

[37] Xuhao Luo, Ramnatthan Alagappan, and Aishwarya Ganesan. 2024. SplitFT: Fault Tolerance for Disaggregated Datacenters via Remote Memory Logging. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, to appear.

[38] Xuchuan Luo, Pengfei Zuo, Jiacheng Shen, Jiazhen Gu, Xin Wang, Michael R. Lyu, and Yangfan Zhou. 2023. SMART: A High-Performance Adaptive Radix Tree for Disaggregated Memory. In *17th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2023, Boston, MA, USA, July 10-12, 2023*, Roxana Geambasu and Ed Nightingale (Eds.). USENIX Association, 553–571. https://www.usenix.org/conference/osdi23/presentation/luo

[39] Yung-Cheng Ma, Chung-Ping Chung, and Tien-Fu Chen. 2011. Load and storage balanced posting file partitioning for parallel information retrieval. *J. Syst. Softw.* 84, 5 (2011), 864–884. https://doi.org/10.1016/j.jss.2011.01.028

[40] Andrew MacFarlane, Julie A. McCann, and Stephen E. Robertson. 2000. Parallel Search Using Partitioned Inverted Files. In *Seventh International Symposium on String Processing and Information Retrieval, SPIRE 2000, A Coruña, Spain, September 27-29, 2000*, Pablo de la Fuente (Ed.). IEEE Computer Society, 209–220. https://doi.org/10.1109/SPIRE.2000.878197

[41] Joel M. Mackenzie, Rodger Benham, Matthias Petri, Johanne R. Trippas, J. Shane Culpepper, and Alistair Moffat. 2020. CC-News-En: A Large English News Corpus. In *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*, Mathieu d'Aquin, Stefan Dietze, Claudia Hauff, Edward Curry, and Philippe Cudré-Mauroux (Eds.). ACM, 3077–3084. https://doi.org/10.1145/3340531.3412762

[42] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. 2016. An Empirical Evaluation of Set Similarity Join Techniques. *Proc. VLDB Endow.* 9, 9 (2016), 636–647. https://doi.org/10.14778/2947618.2947620

[43] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to information retrieval*. Cambridge University Press. https://doi.org/10.1017/CBO9780511809071

[44] Mauricio Marín, Veronica Gil-Costa, Carolina Bonacic, Ricardo Baeza-Yates, and Isaac D. Scherson. 2010. Sync/Async parallel search for the efficient design and construction of web search engines. *Parallel Comput.* 36, 4 (2010), 153–168. https://doi.org/10.1016/j.parco.2010.02.001

[45] Hasan Al Maruf, Yuhong Zhong, Hongyi Wang, Mosharaf Chowdhury, Asaf Cidon, and Carl A. Waldspurger. 2021. Memtrade: A Disaggregated-Memory Marketplace for Public Clouds. *CoRR* abs/2108.06893 (2021). arXiv:2108.06893 https://arxiv.org/abs/2108.06893

[46] Alistair Moffat, William Webber, Justin Zobel, and Ricardo A. Baeza-Yates. 2007. A pipelined architecture for distributed text query evaluation. *Inf. Retr.* 10, 3 (2007), 205–231. https://doi.org/10.1007/s10791-006-9014-4

[47] Jacob Nelson and Roberto Palmieri. 2020. Performance Evaluation of the Impact of NUMA on One-sided RDMA Interactions. In *International Symposium on Reliable Distributed Systems, SRDS 2020, Shanghai, China, September 21-24, 2020*. IEEE, 288–298. https://doi.org/10.1109/SRDS51746.2020.00036

[48] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5895)*, Raghunath Othayoth Nambiar and Meikel Poess (Eds.). Springer, 237–252. https://doi.org/10.1007/978-3-642-10424-4_17

[49] Vijayshankar Raman, Lin Qiao, Wei Han, Inderpal Narang, Ying-Lin Chen, Kou-Horng Yang, and Fen-Ling Ling. 2007. Lazy, adaptive rid-list intersection, and its application to index anding. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou (Eds.). ACM, 773–784. https://doi.org/10.1145/1247480.1247566

[50] Berthier A. Ribeiro-Neto and Ramurti A. Barbosa. 1998. Query Performance for Tightly Coupled Distributed Digital Libraries. In *Proceedings of the 3rd ACM International Conference on Digital Libraries, June 23-26, 1998, Pittsburgh, PA, USA*. ACM, 182–190. https://doi.org/10.1145/276675.276695

[51] Berthier A. Ribeiro-Neto, Edleno Silva de Moura, Marden S. Neubert, and Nivio Ziviani. 1999. Efficient Distributed Algorithms to Build Inverted Files. In *SIGIR '99: Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August 15-19, 1999, Berkeley, CA, USA*, Fredric C. Gey, Marti A. Hearst, and Richard M. Tong (Eds.). ACM, 105–112. https://doi.org/10.1145/312624.312663

[52] Yacine Taleb, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. 2018. Tailwind: Fast and Atomic RDMA-based Replication. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin C. Reed (Eds.). USENIX Association, 851–863. https://www.usenix.org/conference/atc18/presentation/taleb

[53] Shin-Yeh Tsai and Yiying Zhang. 2017. LITE Kernel RDMA Support for Datacenter Applications. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 306–324. https://doi.org/10.1145/3132747.3132762

[54] Jianguo Wang, Chunbin Lin, Ruining He, Moojin Chae, Yannis Papakonstantinou, and Steven Swanson. 2017. MILC: Inverted List Compression in Memory. *Proc. VLDB Endow.* 10, 8 (2017), 853–864. https://doi.org/10.14778/3090163.3090164

[55] Qi Wang, Constantinos Dimopoulos, and Torsten Suel. 2016. Fast First-Phase Candidate Generation for Cascading Rankers. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval, SIGIR 2016, Pisa, Italy, July 17-21, 2016*, Raffaele Perego, Fabrizio Sebastiani, Javed A. Aslam, Ian Ruthven, and Justin Zobel (Eds.). ACM, 295–304. https://doi.org/10.1145/2911451.2911515

[56] Qing Wang, Youyou Lu, and Jiwu Shu. 2022. Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 1033–1048. https://doi.org/10.1145/3514221.3517824

[57] Ruihong Wang, Jianguo Wang, Prishita Kadam, M. Tamer Özsu, and Walid G. Aref. 2023. dLSM: An LSM-Based Index for Memory Disaggregation. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*. IEEE, 2835–2849. https://doi.org/10.1109/ICDE55515.2023.00217

[58] Jeong-Min Yun, Yuxiong He, Sameh Elnikety, and Shaolei Ren. 2015. Optimal Aggregation Policy for Reducing Tail Latency of Web Search. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in*

Information Retrieval, Santiago, Chile, August 9-13, 2015, Ricardo Baeza-Yates, Mounia Lalmas, Alistair Moffat, and Berthier A. Ribeiro-Neto (Eds.). ACM, 63–72. https://doi.org/10.1145/2766462.2767708

[59] Jiangong Zhang, Xiaohui Long, and Torsten Suel. 2008. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*, Jinpeng Huai, Robin Chen, Hsiao-Wuen Hon, Yunhao Liu, Wei-Ying Ma, Andrew Tomkins, and Xiaodong Zhang (Eds.). ACM, 387–396. https://doi.org/10.1145/1367497.1367550

[60] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. 2020. Understanding the Effect of Data Center Resource Disaggregation on Production DBMSs. *Proc. VLDB Endow.* 13, 9 (2020), 1568–1581. https://doi.org/10.14778/3397230.3397249

[61] Tobias Ziegler, Viktor Leis, and Carsten Binnig. 2020. RDMA Communciation Patterns. *Datenbank-Spektrum* 20, 3 (2020), 199–210. https://doi.org/10.1007/s13222-020-00355-7

[62] Tobias Ziegler, Jacob Nelson-Slivon, Viktor Leis, and Carsten Binnig. 2023. Design Guidelines for Correct, Efficient, and Scalable Synchronization using One-Sided RDMA. *Proc. ACM Manag. Data* 1, 2 (2023), 131:1–131:26. https://doi.org/10.1145/3589276

[63] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 741–758. https://doi.org/10.1145/3299869.3300081

[64] Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *ACM Comput. Surv.* 38, 2 (2006), 6. https://doi.org/10.1145/1132956.1132959

[65] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 15–29. https://www.usenix.org/conference/atc21/presentation/zuo

Received October 2023; revised January 2024; accepted February 2024