Virtual Lightweight Snapshots for Consistent Analytics in NoSQL Stores

Fernando Chirigati
New York University
fchirigati@nyu.edu

Jérôme Siméon
IBM Watson Research
simeon@us.ibm.com

Martin Hirzel
IBM Watson Research
hirzel@us.ibm.com

Juliana Freire
New York University
juliana.freire@nyu.edu

Abstract—Increasingly, applications that deal with big data need to run analytics concurrently with updates. But bridging the gap between big and fast data is challenging: most of these applications require analytics' results that are fresh and consistent, but without impacting system latency and throughput. We propose virtual lightweight snapshots (VLS), a mechanism that enables consistent analytics without blocking incoming updates in NoSQL stores. VLS requires neither native support for database versioning nor a transaction manager. Besides, it is storage-efficient, keeping additional versions of records only when needed to guarantee consistency, and sharing versions across multiple concurrent snapshots. We describe an implementation of VLS in MongoDB and present a detailed experimental evaluation which shows that it supports consistency for analytics with small impact on query evaluation time, update throughput, and latency.

I. INTRODUCTION

The ability to leverage insights from large volumes of rapidly-evolving data is a requirement in an increasing number of applications. Decisions must be made on-the-spot for a given incoming event, but often, these decisions must also consider large volumes of historical data. Big data companies have acknowledged the need to effectively combine big data (analytics over retrospective data) with fast data (data that is constantly updated) for a wide range of problems [1], [2].

At IBM, a middleware called “Operational Decision Manager: Decision Server Insights” (ODM Insights) has been developed as a foundation for such applications [3]. Figure 1 depicts its architecture. An event processor receives multiple events that cause updates to the system’s NoSQL data store; simultaneously, it generates actions in response to these events informed by analytics over the same store to analyze historical data and discover new insights. Real-world scenarios targeted by ODM Insights include customer payment verification (e.g., detecting promotions and fraudulent activities), air traffic control (e.g., rerouting the closest flights to an airport), and health costumer care (e.g., predicting outbreaks and illness diagnosis). For most of these applications, data freshness is crucial, since incorrect decisions may be taken if analytics do not see a recent version of the data. But freshness is not the only concern: analytics results must also reflect a consistent view of the data. In these high-stake domains, a seemingly small change in the data can lead to radically different outcomes, and consistency is essential to ensure the integrity of results and predictability of the system’s behavior.

Traditionally, update operations (OLTP) and analytics (OLAP) have been handled by different systems. Transactional systems are optimized to support many concurrent updates, maximizing throughput and ensuring low latency. In contrast, OLAP systems, such as data warehouses, are optimized for long-running queries that involve expensive table scans. Although changes can be propagated from the transactional store to the data warehouse periodically, analytics end up being performed over an outdated version of the data. Using a single system for both workloads brings freshness to analytics, but leads to resource contention: analytics may hold long-running locks, thus negatively impacting update throughput and latency. To mitigate this problem, a lower level of isolation, such as cursor stability [4], can be used, but this has the undesirable effect that records may change while analytics are in progress, leading to inconsistent results.

In the context of the ODM Insights middleware, ensuring consistency is more challenging because the data resides in a NoSQL system. These systems have been widely used to handle both big and fast data, but they often do so without guaranteeing consistent results. One of the goals for ODM Insights is to support MongoDB as its data store for various reasons [5], including (i) its support to JSON, which brings interoperability and flexibility for the middleware, (ii) its ability to handle high update throughputs, (iii) its implementation of a non-versioned store that limits space requirements, (iv) its popularity, and (v) its availability on IBM’s Bluemix cloud platform. However, as many NoSQL systems, MongoDB lacks transactional capabilities and only guarantees weaker forms of consistency, such as atomicity on single record operations and cursor stability for queries. Indeed, despite the foregoing benefits, our prior work with ODM Insights could not use MongoDB directly due to its lack of consistency [5].

This paper introduces virtual lightweight snapshots (VLS), a new snapshotting technique that provides consistency for analytics in the presence of concurrent updates in disk-resident NoSQL systems. VLS was designed to require neither a transaction manager nor native support for versioning: snapshots are virtual and built on demand—as needed by ad-hoc queries—at a low cost, having negligible overheads both in terms of latency and throughput. Our approach focuses on non-distributed scenarios but we are extending the solution to address distributed cases as well (see Section VI).

The design of VLS was inspired by concurrent garbage collection (CGC) algorithms, which perform memory reclamation while applications concurrently modify the memory [6]:
in a database system, this is equivalent to supporting a query to execute over a consistent snapshot while updates are taking place. Therefore, VLS can be seen as a snapshot isolation technique [7] that is amenable for implementation in NoSQL stores. Our approach also shares some similarities with previous work on transient versioning [8]–[10], in the sense that snapshots are created on demand. There is, however, an important difference: VLS does not require a transaction manager. This is key to the applicability of VLS to ODM Insights, as many NoSQL systems, including MongoDB, lack support for transactions.

The core contribution of this paper is VLS. VLS is a new technique that enables consistent analytics in disk-resident NoSQL systems (e.g., MongoDB) while, at the same time, attaining low latency and high throughput for updates (Sections II and III). The overheads are kept low through the use of data structures that are compact and support fast operations to manage the snapshots. In addition, storage usage is minimized by sharing data across multiple queries and snapshots. As a proof of concept, we implemented VLS in MongoDB by modifying the native database operations to support the VLS algorithm (Section III-C). Furthermore, our detailed experimental evaluation shows that VLS supports consistent results with low memory usage and computational overheads for analytics and updates (Section IV).

II. BACKGROUND AND MOTIVATION

An important goal of our work is to provide efficient support for concurrent updates and analytics running in a NoSQL system, so as to have results that are both fresh and consistent. Our solution to this problem was inspired by techniques used in CGC. Although these techniques were designed for a different problem, issues arise during a table scan in the presence of updates that are similar to those that arise in CGC. When attempting to reclaim memory, the garbage collector scans the memory and must see a consistent version to avoid premature reclamation. Concurrently, one or more threads of an application may modify that same memory, e.g., by manipulating object pointers. Analogously, a table scan that is performed while multiple updates are concurrently applied should see a consistent snapshot of the table.

Snapshot-at-the-beginning algorithms are a class of CGC algorithms that reclaim only objects that are unreachable at the beginning of a garbage collection scan. The graph of reachable objects is fixed at the moment garbage collection starts, and the collector only sees that snapshot, ignoring objects that become unreachable as it runs to ensure termination. In what follows, we describe one such algorithm, Yuasa’s algorithm [6], that allows applications to keep running and to make modifications to object pointers while the garbage collection is being executed. We begin by showing, through an example, how the algorithm can be adapted to support a single scan without blocking concurrent updates. Then, we discuss additional challenges that arise in the database context, notably how to support index scans and multiple concurrent analytics, and give an overview of how VLS addresses them.

Single Snapshot: Example. Consider a simple scenario from an air traffic control, which represents a typical ODM Insights use case: there is an airport closure due to a storm and flights must be rerouted. To identify the closest flights, a query is issued to the database that holds information about all flights for a given destination. This database is updated continuously, receiving events as planes move. Each record in the table has a unique id, the flight number, and the current distance to the destination airport. Finding the closest flights requires a scan S over the table. For the time being, we assume that there are no indices and S needs to read the entire table.

Figure 2a shows the state, or snapshot, of the table at the time S starts. Using transient versioning terminology, we call this snapshot the stable version of the table for the analytics, with the corresponding records being the stable versions as well. As in Yuasa’s algorithm, where the garbage collector sees the memory state at the time it begins its execution, a query sees the database state as of the time it starts and must not see subsequent changes.

As changes are applied, the snapshot is maintained using two data structures: (i) a bit that is assigned to each record in the base table to indicate whether it is stable (bit set to 0), or whether it has been changed or read by the scan (bit set to 1); and (ii) a remset (which stands for remembered set) that remembers the stable version for the records that have been changed but have yet to be read since the scan started. Similarly, in CGC, the bits indicate which object pointers the garbage collector should read and which pointers it should skip, while the remset stores the overwritten values for the object pointers skipped by the collector.

At the beginning of scan S, all bits are set to 0, and the remset is empty, as shown on Figure 2a. As S proceeds, it reads records and switches the bit from 0 (unread) to 1 (read). During the scan, updates may be applied to the table. Figure 2b shows S having read the first two records and three updates being applied to records that have yet to be read (ahead of S). Update U modifies the value for record 2. Since the current version of this record is the stable one, U first copies this...
version to the remset, flips its state to 1, indicating that this record has changed, and applies the update. Update D deletes record 4 from the base table, but before doing so, copies its current version to the remset. Finally, update I inserts a new record and sets the bit to 1 indicating that it does not belong to the stable version of the table, i.e., it must be skipped by \( S \).

Figure 2c shows three additional updates on records that have their bit set to 1. The first update is on record 1, which has already been read by \( S \), while the second and third updates are on records 2 and 7, ahead of \( S \) but with their bits already set to 1. These updates can be applied directly, requiring no bit operations or changes to the remset. As \( S \) proceeds, it skips records 2 and 7, since they have been changed/inserted after \( S \) started. Figure 2d shows, in the base table, read records in a gray background, and the skipped ones with a white background. After scanning the base table, \( S \) retrieves the stable version of records that have been skipped from the remset: record 2, which was modified, and record 4, which was deleted. The combination of the records read from the base table (gray records) and the remset records corresponds exactly to the stable version of the table and is what we call a virtual lightweight snapshot (VLS). The snapshot is virtual because it is simulated rather than materialized, and it is lightweight as the bits and the remset are the only additional pieces of information used to maintain the snapshot.

Note that this snapshot ensures consistent results: at the beginning of the scan (Figure 2a), the closest flight is the one in record 1, and since the stable version of the table is maintained, the result reflects that version. If cursor stability were used instead, individual records would be locked as they are read by the scan: after the scan fetches records 0 and 1, and after the updates in record 2 are applied in Figure 2b and Figure 2c, the scan would identify that record 2 contains the closest flight, which is incorrect. In addition, this result reflects neither the stable version of the table, nor the version of the table at the end of the scan: for both versions, the closest flight is still the one in record 1.

Challenges in the Database Context. While we can draw analogies between CGC and consistent evaluation of database queries, there are additional issues that need to be addressed in the database context. One challenge is that garbage collection works for a single scan, while databases need to support concurrent analytics corresponding to distinct, possibly overlapping, snapshots. A naive approach would be to simply allocate one bit and one remset per scan. However, the overlap between concurrent scans can be large, leading to many duplicated record versions across remsets and inefficient storage use. To avoid duplicates, we extend the bit logic described above to allow multiple, overlapping snapshots to share a single remset, reducing the amount of additional storage required. Our experimental results show that sharing remsets leads to storage savings of up to 60 times (Section IV).

Another challenge is that garbage collection always performs a full scan, while we need to support index scans, i.e., partial scans on the base table. For example, assume that the table from Figure 2 has a B-Tree index on the distance field and a query asks for flights that are at most 400 miles from the airport. When using the index, some records are skipped (e.g., flights with distance greater than 400 miles: records 3, 4, and 6). Updates to these records would still add them to the remset as the bit indicates that they are stable, leading to incorrect results: the delete operation in Figure 2b would maintain that version of record 4 in the remset, while this version should not be read by the index scan. The VLS algorithm ensures that these spurious records, i.e., records incorrectly added to the remset, are not retrieved by the index scans.

A third challenge is to ensure the integrity of the bits used for records as the system runs, in a way that minimizes the overheads and allows resources to be efficiently reused. As a simple example, consider again Figure 2: at the beginning of the scan, all records have their bit set to 0, and at the end of the scan, all records have their bit set to 1. A classic CGC implementation would do a pass to flip the bits to 0 again before a new scan starts. This is obviously expensive in a database context, in particular for large datasets with many millions (or billions) of records. We avoid the need to reset the bits by changing their semantics between two consecutive scans, i.e., in every other execution, 1 becomes the indicator for stable records. As a consequence, these bits can be reused without the need for an expensive operation to reset them. Index scans, on the other hand, require additional bookkeeping: since some records are not read, their bits remain unflipped. Thus, the algorithm needs to ensure that all the bits are properly set for a new scan. While for full scans the overheads for maintaining the bits are amortized, since all records are read, a naive solution for index scans can result in unacceptable costs, which we avoid in our approach.

III. THE VLS ALGORITHM

Figure 3 shows a high-level overview of the VLS architecture. The approach implements versioning at the record level: a virtual lightweight snapshot is created for each read-only query (i.e., analytics) and consists of (a subset of the) records from the base table and from the shared remset. A query runs over its snapshot and thus sees a consistent view of the data. Update operations applied to the base table maintain these snapshots by copying stable versions of modified records to the remset, where these versions can be shared across queries, limiting space overhead.

Our goal is to support consistent analytics without negatively impacting the concurrency in NoSQL systems (e.g., MongoDB) so as to bring consistency to ODM Insights. These systems often lack transactional capabilities, and although some stores have native support for database versioning, not all of them do, and full-fledged versioning costs a lot of storage space. Thus, we make the following assumptions about the underlying store: (i) there is no support for ACID transactions, and as a consequence, updates are atomic and modify single
records; (ii) there is no database versioning; (iii) the data is disk-resident and non-distributed; and (iv) analytics fundamentally rely on a few key data access operations, namely scans, which can be either full scans over a given base table or index scans. The VLS algorithm supports any analytics that can be built on top of these scans. In ODM Insights, that includes both descriptive analytics [11] and predictive analytics [3].

The remainder of this section describes the VLS algorithm in detail, showing how it provides efficient support for (i) multiple scans and (ii) index access. The end of the section also discusses our current VLS implementation.

A. Support for Multiple Scans

The VLS algorithm extends the bit representation and the remset used in the single-scan approach from Section II. Instead of relying on a single bit, N-bit vectors are used to handle N concurrent analytics, so that each scan can set these bits independently. Bit vectors are also attached to the remset to indicate which record versions correspond to which active scans. Additional bookkeeping is required to: (i) allocate bits to incoming scans, (ii) maintain the meaning of bits (stable and changed/read), and (iii) remove records from the shared remset when they are no longer needed.

1) Data Structures: Our approach uses several masks that are represented as bit vectors. The algorithm and the choice of specific masks are such that we can always use efficient bitmap operations (Table I), making the solution practical and limiting the overhead added to the database operations (see Section IV). We note that bit vectors have been used before to efficiently deal with a variety of database problems, including concurrent query processing in data-centric engines [12] and multi-query optimization in data warehouses [13].

We first discuss the masks used for scans that are not tied to a specific record.

- **Active Mask.** When a scan starts, it is assigned a specific bit that is identified by its position. A scan uses this bit consistently, i.e., the same position, in all masks for its entire duration. To assign a position to an arriving scan, we need to know which positions are available. This is done by using the global N-bit vector active mask: if position i in active mask has bit 1, then position i is not being used by any active scan; otherwise, if the bit is 0, position i has already been assigned.

While active mask is global to the database and is used to control the assignments of scans to bits, the base table maintains a local copy of this mask to inform table updates if stable versions of the records must be maintained in the remset (i.e., if there are active scans while updates are in progress). This copy is stored in the N-bit vector local active mask. We show later, while describing the database operations, how these masks are synchronized.

- **Stable Mask.** The global N-bit vector stable mask stores which value in each bit means stable. For instance, if position i in stable mask is 0, then 0 corresponds to stable for that position. Initially, stable is represented by 0 in all positions, i.e., the stable mask is initialized as a vector of bits set to 0.

- **Scan Mask.** In addition to active mask, which globally keeps track of which bits have been allocated, each scan needs to know which bit it owns. Therefore, each scan keeps a mask indicating which position it uses, represented by the N-bit vector scan mask. In this mask, the position with bit value 1 is the one assigned to the scan—all the other positions are set to 0.

Figure 4 illustrates the use of these masks. Their values change only when a scan starts or completes. At time T₀, no scan is active and active mask only contains bits with value 1. Because two scans were executed to completion before T₀, stable mask indicates that 1 means stable for the first two bits (which were assigned to these two scans), and that 0 means stable for the last three bits. When scan S₁ starts at time T₁, it is given an available bit (here, the third bit), reflected in active mask by setting the third bit to 1, and a scan mask indicating that it should use the third bit for all its operations. Similarly, when scan S₂ starts at time T₂, it is allocated the first bit, and given a corresponding scan mask. Once S₁ completes at time T₃, its scan mask is released, the third bit in active mask reverts to 1, and the meaning of stable is switched from 0 to 1 in stable mask. The same occurs when scan S₂ completes at time T₄. At this point, all the bits in active mask have value 1, i.e., there are no scans running.

We now present the bit vectors tied to specific records. As described in the previous section, the remset is used to store the stable versions of the records that have been modified but that have yet to be read by a scan; note that one record may have different versions in the remset, since scans can start at different times. As snapshots for different active scans may overlap, to make efficient use of memory, we use a shared remset, along with additional bit masks that indicate which record versions a scan should fetch from it. The following masks are used in either the base table or the remset:

- **Status Mask.** Each record in the base table is associated with an N-bit vector status mask, where each bit corresponds to a different active scan.

- **Remset Status Mask.** The algorithm maintains a mask for each record version in the remset: the N-bit vector remset status mask. If position i in the remset status mask of a record version is 0, then the active scan S assigned to position i should read this version, i.e., this version belongs to the snapshot for S; otherwise, if the bit value is 1, S should skip it.

In addition to keeping track of which snapshot a record belongs to, the remset status mask is also used to support...
TABLE I: Efficient bitmap operations in the VLS algorithm.

<table>
<thead>
<tr>
<th>Name</th>
<th>Bitmap Operations for Scans and Updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>REMSET_MASK_INS</td>
<td>(stable mask XOR status mask) OR (local active mask)</td>
</tr>
<tr>
<td>REMSET_MASK_UPD</td>
<td>local active mask OR remset status mask</td>
</tr>
<tr>
<td>LCL_ACTIVE_MASK_UPD</td>
<td>active mask OR scan mask</td>
</tr>
<tr>
<td>ACTIVE_MASK_UPD</td>
<td>NOT (stable mask XOR local active mask)</td>
</tr>
<tr>
<td>STATUS_MASK_SCAN_UPD</td>
<td>status mask XOR scan mask</td>
</tr>
<tr>
<td>SCAN_TABLE_READ</td>
<td>(stable mask XOR status mask) AND scan mask</td>
</tr>
<tr>
<td>INDEX_ACTIVE_MASK_UPD</td>
<td>index status mask OR local active mask</td>
</tr>
<tr>
<td>SCAN_INDEX_MASK_U_UPD</td>
<td>index status mask OR index active mask</td>
</tr>
<tr>
<td>SCAN_INDEX_MASK_U_UPD</td>
<td>index status mask OR scan mask</td>
</tr>
<tr>
<td>IDX_INDEX_MASK_U_UPD</td>
<td>index status mask AND (NOT (scan mask))</td>
</tr>
</tbody>
</table>

memory reclamation of versions in the remset that will no longer be used: if all the bits in a remset status mask are 1, no other scan will read the corresponding version, so the scan can remove this version from the remset. This reclamation is immediate: a record version is removed as soon as it is no longer needed.

Next, we look at how these bit masks are maintained by individual operations on the store: table updates and full scans. Since our focus is on NoSQL stores, we assume that table updates are atomic, and that there is no transaction manager. While describing the algorithm, we refer to the bitmap operations from Table I.

2) Update Operations: Let U be an update operation that inserts, deletes, or modifies a single record.

U modifies a record. First, U uses the bit operation REMSET_MASK_INS and stores its value in a temporary N-bit vector temp mask. If all bits in temp mask have value 1, it means that (i) this record has already been modified before (if so, stable versions were already placed in the remset), or (ii) all active scans have already accessed it (if so, these scans no longer need this record), or (iii) there are no active scans, or (iv) a combination of the previous; therefore, U simply modifies the record and no operations need to be applied to the masks. Otherwise, if there is at least one bit 0 in temp mask, the current version of the record is the stable one for at least one active scan; in this case, this version needs to be maintained in the remset prior to the update, and the following actions are executed in order: (1) the current version of the record is added to the remset, together with temp mask, which corresponds to the remset status mask of this version; (2) status mask is updated to the result of bit operation STATUS_MASK_UPD, to inform active scans to skip the record; and (3) the record is modified by U.

Note that, in the bitmap operation REMSET_MASK_INS, the expression stable mask XOR status mask checks if, for each bit position in status mask, the record is stable (outputting 0 for stable), while the remainder of the operation ensures that only bits having active scans are taken into account. Other operations perform similar XOR operations against the stable mask to capture the proper stable semantics, including STATUS_MASK_UPD.

U inserts a record. U inserts the new record with a status mask generated with the bit operation STATUS_MASK_UPD: this operation guarantees that active scans will skip the record, and that new scans include that record in their snapshots.

U deletes a record. The steps are the same as when modifying a record, except that U deletes the record from the base table, and there is no need to maintain the status mask of the record.

3) Full Scan Operations: We now describe the algorithm for a full scan S.

S begins. If all bits in active mask are 0, there is no position available for S and the scan is placed on a queue for future execution (see discussion at the end of this subsection). Otherwise, a free position i is assigned to S, and both active mask and local active mask are updated together to have the bit at position i set to 0. Finally, S generates its scan mask, and can start scanning the table.

S reads a record in the table. S stores the value of the bit operation SCAN_TABLE_READ in a temporary N-bit vector called temp mask. If all bits in temp mask have value 0, the current version of the record is the stable one for S. Then fetches the record and updates its status mask to the value of bit operation STATUS_MASK_SCAN_UPD: this operation updates (i.e., flips) the bit value in position i to reflect that S has already read this record, i.e., that S will not need it anymore. Otherwise, if position i of temp mask has value 1, the current version of this record is not the stable one for S, i.e., this record has already been updated (inserted / modified) after S started, so S skips this record. It is worth noting that, in SCAN_TABLE_READ, the use of scan mask ensures that only position i of S is taken into account, and the remainder of the bit operation takes care of capturing the stable semantics, as explained earlier.

S finishes scanning the table. S updates stable mask using the bit operation STABLE_FLIP, which ensures that the bit at position i is flipped to change the meaning of stable. Besides, bit operation LCL_ACTIVE_MASK_UPD is used by S to update local active mask. After scanning the table, S needs to scan the remset to get the stable version of records it had skipped before (see the next operation).

S reads a record version in remset. S stores the value of operation SCAN_RemSET_READ in a temporary N-bit vector temp mask. If the bit at position i in temp mask is 1, that record does not belong to the snapshot for S and is skipped. Otherwise, S accesses the record and updates remset status mask using the REMSET_MASK_UPD bit operation, which flips the bit at position i in remset status mask to record the fact that S has already fetched this version, i.e., that S will no longer use the record. At this point, an important part of the algorithm takes place, which is the memory reclamation of the remset: if all bits in the updated remset status mask have value 1, this version will no longer be used by any scan, and S removes it from the remset. This reclamation is immediate: as soon as a record version becomes unnecessary, it is removed from the remset.

S ends. S uses ACTIVE_MASK_UPD to update active mask, making position i available again for future scans, and synchronizing its value for bit i with local active mask.

4) 3-Scan Example: Figure 5 shows some of the stages of VLS using the same example presented in Section II. We assume that two analytic queries (S1 and S2) to find the closest flights from a given airport are executing concurrently, and both perform a full scan over the base table. To simplify the example, we assume that N is 3.

Suppose that active mask is 100, stable mask is 000, and bits 2 and 1 were assigned to S1 and S2, respectively (bit 1
being the rightmost bit in the masks). Figure 5a shows the state when S1 is reading record 5 and S2 is reading record 1, at which point U modifies record 2. In this case, the current version of record 2 must be stored in the remset for S2. Note that S1 has already read this record. Before applying the update, U copies the version to the remset, together with remset status mask equals to 110, indicating that this version belongs to the snapshot for S2. If another U modifies record 6, the current version needs to be stored for both scans, since it belongs to both snapshots. The record version is copied only once to the remset, and remset status mask is set to 100, indicating that both scans need to read this version, i.e., the version is shared between the snapshots.

After scanning the table, S1 scans the remset (Figure 5b). S1 only fetches record 6, and changes its remset status mask accordingly. When S2 starts scanning the remset, it reads the versions of records 2 and 6. At that point, both masks are set to 111, which means that no other scan will use them in the future, and S2 can remove these record versions from the remset (Figures 5c and 5d).

5) Final Considerations: It is worth mentioning that, conceptually, the VLS algorithm can be used to run any number of concurrent analytics, but in practice, our approach allocates bit vectors in the database with a given size N, and this value determines the maximum number of concurrent scans. The main reason for this design is that having fixed-size masks avoids the costs of dynamically allocating bit vectors as scans arrive and finish. Also, the algorithm is more efficient when bit operations are implemented using native machine instructions, i.e., N should be set according to the register width of the CPU for optimal performance, as we show in Section IV.

From a query planning point of view, if a new scan request arrives and there are no positions available, i.e., N scans are already active, the scan can be added to a scheduling queue for later execution. When an active scan completes, it signals to this queue so that the next awaiting scan can execute. Instead of waiting for a position on a per-scan basis, it is also possible to allow several queries to share an underlying snapshot. This requires some deeper integration with the query planner, which must decide which requests should share a scan to make best use of computational resources, as well as synchronize execution for the queries sharing a given scan. We plan to address this in future work.

B. Index Support

For the purpose of this discussion, we assume a simple B-Tree index, or a balanced search tree where non-leaf nodes contain a set of keys1. A leaf node keeps a set of pointers to records in the base table—the record pointers—based on its parent’s keys, and this set is also ordered by key. Figure 6a shows an example of a B-Tree index with rank 2 on the distance field from Figure 2a: record pointers are represented by dashed arrows, and record ids are enclosed in circles.

Index scans are handled in a way similar to full scans, but two additional issues must be taken into account. First, when using the index, only a subset of the index and corresponding base table are scanned, which means that updates operating over records not supposed to be read by the scan may place spurious records in the remset. Thus, we also need to ensure the consistency of the bit masks after the index scan finishes. Recall that bits are flipped as the index scan progresses, but records not read by it remain untouched, which may impact the correctness of an upcoming scan using the same bit position.

Second, the physical structure in the index, notably the order of record pointers, may change as updates occur, requiring additional bookkeeping. As an example, consider the B-Tree presented in Figure 6a, and assume a query is issued to retrieve flights that are at most 400 miles away from the airport. Such an index scan, based on the stable version of Figure 2a, must read records 0, 1, 2, and 5. Gray areas and arrows in Figure 6a show the progress of the scan: it starts at the root node and traverses down to the first record pointer, linking to record 1. If update U of Figure 2b is applied at this moment, and since the distance value in record 2 changes from 367.21 miles to 100.45 miles, the pointer to record 2 is removed from the index and inserted before record 1’s pointer (see Figure 6b). Since the scan follows the order in the B-Tree, it will read records 5 and 0 afterwards, thus missing record 2.

To address these issues and correctly support index scans, we make the following adjustments to the VLS algorithm:

- To avoid retrieving records added to the remset but which should not be read by the index scan (i.e., spurious records), we assign a set called record set to each index scan. This set contains the ids of records that a scan should read from the

1The approach can be extended to other index types.
To maintain stable versions of the index, our approach uses two additional masks:

- **Index Active Mask.** While active mask contains information regarding both full and index scans, the global $N$-bit vector index active mask only takes into account index scans. This mask is used exclusively for index scan operations.

- **Index Status Mask.** Each record pointer in a B-Tree has its own $N$-bit vector index status mask. As with the status mask, each bit in the index status mask corresponds to an active scan. A position $i$ is 1 if the index scan $S_i$ at position $i$ should skip the record pointer (e.g., the pointer has been either inserted after $S_i$ started, or removed before $S_i$ started); if position $i$ is 0, the corresponding record belongs to the stable version for $S_i$ (e.g., either the pointer has not been altered, or it has been removed after $S_i$ started).

To capture which record pointers have had their removal postponed, each record pointer also maintains one extra bit that we call index status bit: this bit is 1 if the pointer has been flagged for removal, and 0 otherwise. The information provided by index status bit is important to reclaim index entries, as well as to inform active index scans about the removal: if the bit is 1, the scan adds the record id to its record set and does not need to follow the record pointer to the table. Note that the removal of a record pointer is triggered by either a record update that changes the position of its pointer in the index, or a record deletion; in both cases, the stable version of the record for the scan is in the remset, and we add its id to record set so that the scan does not identify it as a spurious record.

It is important to note that if the update is a deletion, although the removal of the record’s corresponding pointers is postponed, the deletion of the record itself from the base table is not deferred. Therefore, every operation over the B-Tree (besides index scans) must check the index status bit to avoid following pointers that may have become invalid. Note also that the database can reclaim index entries every time an index scan finishes, thus avoiding keeping removed record pointers in the B-Tree for long periods of time. We describe this reclamation process later in this section.

2) **Index Update Operations:** We consider two kinds of updates $U_1$ to record pointers in a B-Tree index: insertion and deletion. The maintenance of the index does not change, except that, when reallocating record pointers, their masks and bits need to be reallocated as well.

- **$U_1$ inserts a record pointer in a B-Tree.** When a new record pointer is inserted into a leaf node, $U_1$ needs to ensure that active index scans will skip it. $U_1$ inserts the record pointer in the B-Tree with an index status mask set to the value of the bit operation IDX_STATUS_MASK_INS. In addition, it is assigned an index status bit of value 0.

- **$U_1$ deletes a record pointer in a B-Tree.** If there are no active index scans, i.e., if index active mask is a mask of bits set to 1, $U_1$ simply deletes the record pointer and no further steps are necessary. Otherwise, if there is at least one active index scan, $U_1$ does not delete the record pointer: the removal is postponed. The pointer is still maintained in the leaf node, and $U_1$ updates its index status mask to the result of bit operation IDX_STATUS_MASK_UPD; this update ensures that active index scans can still see the record pointer, while new upcoming scans will skip it. In addition, index status bit is set to 1 to indicate that the removal of the record pointer has been deferred.

3) **Index Scan Operations:** For an index scan $S_i$, the algorithm needs to (i) verify the index status mask of record pointers before reading records from the base table; (ii) make use of the record set when scanning both the index and the remset to deal with spurious records; and (iii) inform the database when to flip the bits in the status mask vectors and reclaim index entries.

- **$S_i$ begins.** The steps are the same as for a full scan, but $S_i$ also (i) initializes an empty record set before it starts scanning the index and the table, and (ii) updates index active mask by setting the position $i$ assigned to $S_i$ to 0.

- **$S_i$ reads a record pointer in a B-Tree.** Before reading and following the record pointer, $S_i$ uses bit operation IDX_SCAN_READ and stores its value in a temporary $N$-bit vector temp mask. If temp mask has one bit set to 1, it means that the record pointer was either inserted after $S_i$ started, or flagged...
for removal before $S_I$ started: $S_I$ simply skips the pointer, since it does not belong to its snapshot. Otherwise, if $temp\ mask$ is a vector of bits set to 0, the record pointer belongs to the snapshot for $S_I$, and $S_I$ adds the corresponding record id to $record\ set$; in addition, it checks the value of $index\ status\ bit$: if the bit is 1, the record pointer has been flagged for removal, so $S_I$ skips the pointer (the stable version for the record is in the $remset$); however, if the bit is 0, $S_I$ follows the pointer to read the record in the table (next operation).

$S_I$ reads a record in the table. The steps are the same as for full scans. Remember that $S_I$ still needs to execute the VLS algorithm when reading records in the base table to update the $status\ mask$ of the records and to deal with updates that do not change the physical structure of the index.

$S_I$ finishes scanning the table. The steps are the same as for a full scan. The bit mask $index\ active\ mask$ is also updated with the value of $IDX\_ACTIVE\_MASK\_UPD$: this mask is updated before scanning the $remset$ so that no upcoming updates in a B-Tree erroneously keep stable version information for $S_I$.

$S_I$ reads a record version in $remset$. The steps are the same as for a full scan, except that $record\ set$ is also used. In the algorithm for a full scan, if the bit at position $i$ in $temp\ mask$ is 0, this record belongs to the snapshot for the scan. For index scans, the record could also be a spurious record. Given that the bit is 0, we need to: (1) verify if the id of the record is in $record\ set$: if this is the case, $S_I$ reads this record; otherwise, it skips it; and (2) update $remset\ status\ mask$ and, if all its bits are 1, do the memory reclamation. Note that (2) is always executed when the bit value is 0.

$S_I$ ends. $S_I$ discards its $record\ set$ and starts a background thread that is handled by the database itself and that executes the following operations in order: reclamation of index entries that were modified while $S_I$ was executing; and flipping of the bits in position $i$ for the $status\ masks$ of records neither touched by $S_I$ nor updated. After starting this thread, $S_I$ completes its execution. Note that $active\ mask$ is not updated at this point, since we need to make sure all the masks are consistent before allowing new upcoming scans to use the same bit position.

4) Operations for Finalizing an Index Scan: The following are operations started by an index scan $S_I$ right before its completion; they are executed in a background thread and handled by the database system.

Index reclamation. The reclamation process iterates over the record pointers of the B-Tree index structure. For each pointer, if the $index\ status\ bit$ is 0, the corresponding pointer may have been inserted while $S_I$ was running; in this case, the algorithm updates the $index\ status\ mask$ to the result of bit operation $scan\_IDX\_MASK\_1\_UPD$, so that new upcoming scans using the same bit position do not skip this pointer. Otherwise, if the value of $index\ status\ bit$ is 1, the pointer is flagged for removal; first, $index\ status\ mask$ is updated to the value of bit operation $scan\_IDX\_MASK\_1\_UPD$, to indicate that $S_I$ does not need this pointer anymore; then, its updated value is verified: if all its bits are set to 1, no other index scan will use this pointer, so it can be removed from the index, and its $index\ status\ mask$ and $index\ status\ bit$ are discarded.

Resetting the status masks. We iterate over all $status\ masks$ in the table, flipping the bit position used by $S_I$ as if the record is being read by $S_I$: bit operations $scan\_TABLE\_READ$ and $STATUS\_MASK\_SCAN\_UPD$ are used for the update process, similar to when a scan reads a record in the table. This guarantees that, even if the record was not read by $S_I$ (i.e., even if it does not satisfy its query predicate), its bit is properly flipped to not affect the results of a new scan that will use the same bit position. Next, $active\ mask$ is updated to the value of bit operation $ACTIVE\_MASK\_UPD$, freeing the bit position to new arriving scans.

C. Implementation

We implemented the VLS algorithm inside MongoDB, since one of the goals is to support this system as the main data store in the ODM Insights middleware. MongoDB is a document-oriented database that uses Binary JSON (BSON) as a storage format, and it stores collections of key-value pairs. The system, as many other NoSQL stores, neither supports versioning nor guarantees consistency for analytics in the presence of concurrent updates.

Our prototype, MongoDB-VLS, extends the built-in database operations from MongoDB with VLS capabilities. We modify the standard create (i.e., insert), update, and delete operations based on the algorithm from Section III. MongoDB-VLS also modifies the built-in scan operations: both full and index scans read and write bit masks, skip records from the base table if necessary, and read the $remset$ at the end of the execution. For index scans, as described in the previous section, a background thread is used to ensure that the bit masks are properly set. Our prototype can run MongoDB workloads composed of updates along with aggregate queries, which use the built-in scan operations internally.

Masks and $remsets$ are stored in memory by the system. For the bit masks, we maintain a mapping between the record locations on disk (composed of a disk file identification and an offset value) and their corresponding masks. Note that operations on $remsets$ as well as on bit masks, either global or associated with a record, must be atomic and thread-safe to avoid inconsistencies if two or more operations occur on the same record simultaneously. We rely on the concurrent data structures and atomic operations from the TBB library, which proved to be the most efficient in our experiments: we use a concurrent hash-map for the $remset$, while bit masks are represented as bit vectors using TBB’s atomic interface.

IV. EXPERIMENTAL EVALUATION

To assess the effectiveness of VLS when integrated into a NoSQL store and to better understand the costs associated with supporting consistent analytics in the presence of updates, in our experimental evaluation, we compare the performance of MongoDB-VLS against a standalone MongoDB installation. We study the impact of the algorithm on update throughput and latency, as well as on analytics completion time. We also examine the costs incurred when maintaining the data structures required by the snapshots, including memory consumption, and evaluate how the size of the bit vectors impacts our approach.

A. Experimental Setup

Experiments were run on a server running CentOS (4 16-core 2.3 GHz AMD Opteron 6276 CPUs, 256 GB of RAM, 2Source code is available at https://github.com/ViDA-NYU/mongodb-vls 3https://www.threadingbuildingblocks.org/
and 4 3TB-SATA drives), using MongoDB 2.5.5 and TBB 4.2 Update 3. We set N to 64, which corresponds to the instruction width of our server, unless otherwise noted.

We ran two different workloads: one for analytics using an aggregate query that counts the total number of records being read, and one for updates. These workloads were generated and run with YCSB [14], a popular open-source benchmark for NoSQL stores. We extended YCSB 0.1.4 to include aggregate queries (using both full and index scans), and ran the benchmark using MongoDB’s API. The aggregate query for index scans is the same as for full scans, but includes a condition on the indexed attribute, matching records with key greater or equal to a given input value.

For all experiments, the YCSB client ran on the same machine as the database server. Workloads were run using 10 threads for updates, and one thread per analytics. We evaluated the approach for different index rates, indicating how much of the table is being scanned: 5%, 10%, or 25% of the base table. The default attempted update rate was set to 10,000 operations per second (op/s), and keys chosen for the update operations were generated using a zipfian distribution originally provided by YCSB. Each experiment—except the one that verifies the size of the shared remset—was executed 10 times, and both the mean and standard deviation (error) were computed. All experiments started with a flushed cache, and we used different YCSB configurations to vary the table size (10 million and 100 million records) and the number of concurrent analytics. For all table sizes, each record used 240 bytes.

B. Experimental Results

Query Execution Time. To understand the overhead associated with the VLS algorithm, we measured the raw analytics performance with no concurrent updates for both MongoDB-VLS and MongoDB. Figure 7a shows, in log scale, the query duration for the two table sizes. For a query that performs a full scan, the relative overhead added by the VLS approach is about 5% and 1% for 10 million and 100 million records, respectively; in absolute numbers, MongoDB-VLS increases the scan duration from 24.87s to 26.33s (about 1.46s) for the smaller table, and from 252.85s to 255.37s (about 2.52s) for the larger table. For a query that uses an index scan, the overhead is under 10% for both tables. Consider, for example, the index scan that reads 5% of the 10 million-record table: the overhead added by VLS is 9.67% (from 2.09s to 2.29s); for the 100 million-record table, it is 5.67% (from 19.95s to 21.08s). As expected, the relative overheads for index scans are bigger than those for full scans since these require additional bookkeeping; however, when considering the total running times, the increase is negligible: in many cases, less than 1s. We also studied how VLS impacts analytics execution time in the presence of concurrent updates: not surprisingly, the duration of a query increases substantially in both systems, but VLS has a negligible impact for both full and index scans (Figure 7b). Overall, the experiments on query execution time indicate that the overheads incurred by VLS are acceptable, while providing consistent and correct results for analytics.

Update Throughput. Figure 8a shows, for the different table sizes, how the update throughput changes when the number of concurrent analytics using full scans increases. As a baseline, we include the update throughput when no analytics are running (i.e., number of analytics equals to 0); note that VLS has little or no impact over the raw update throughput. When running concurrent analytics, the average overhead added by MongoDB-VLS varies from 2.84% to 16.36% for different table sizes and number of concurrent analytic queries. Nonetheless, most of the error bars overlap, showing that the difference between the average throughput for the two systems is statistically small. For concurrent analytics using index scans, most of the overheads are also small. As Figure 9 shows, the overhead on update throughput for all index rates varies from 0.70% to 16.28%, again with many overlapping error bars. We also experimented with other update rates (e.g., 20,000 op/s, 30,000 op/s, and 100,000 op/s), and obtained similar and consistent results.

Figures 8a and 9 show interesting trends. First, as expected, as the number of running queries increases, the higher concurrency in the system results in longer updates and smaller throughput. One counter-intuitive feature is that the overall update throughput for both systems is higher for 8 concurrent queries than for a single query. We believe that this is due to the aggressive caching strategy used by MongoDB: with 8 concurrent scans, more records are read and put in memory, speeding up updates. This is consistent with the latency results in Figures 8b and 10: updates in the presence of 8 concurrent queries are faster than in the presence of a single query.

Another interesting behavior is that the performance in both MongoDB and MongoDB-VLS for 64 concurrent queries is significantly lower when compared to the other configurations. This happens mainly due to the higher concurrency in the database and the increase in resource consumption: for 64 concurrent queries, there is a total of 74 threads (1 per query, and 10 for updates), which is larger than the number of cores in our server (64), making experiments more sensitive to resource contention. Last, Figures 9 and 10 (the latter shows the latency results discussed later) unveil that higher index rates result in better performance (i.e., higher update throughput and faster updates) in both systems. The main reason for such behavior is the locking mechanism implemented by MongoDB, which yields locks more often for update operations when reading larger regions of the table; therefore, more updates are received and applied while the analytics are running. As the number of concurrent analytics increases, this is almost unnoticeable due to the higher concurrency in the database.

Update Latency. We measured the overhead that MongoDB-VLS adds to the duration of an update. We report these results in 95th percentiles, i.e., the maximum latency among 95% of the update operations. Figure 8b presents, in log scale, the experimental results for different
table sizes and numbers of concurrent analytics using full scans. The overhead is most significant when using MongoDB-VLS in the 10 million-record table with a single analytic query, where the percentile increases 27.23%, and with 8 concurrent queries, where there is an increase of 18.64%. However, note that the percentile latency is in a very fine time granularity: in absolute numbers, the latency increases from 3.87ms to 4.93ms for a single analytics, and from 2.34ms to 2.77ms for 8 queries, which can be acceptable for a number of application scenarios. For all other configurations, the overhead varies between 1.62% and 15.04% with overlapping error bars for most of the results, which makes the overhead not statistically significant. Similar results are obtained for concurrent analytics using index scans (Figure 10): most of the overheads are under 18% and not statistically significant. A higher overhead can be noticed for the smaller table when running 8 and 64 concurrent analytics that scan 25% of the table (Figure 10c). Nevertheless, besides the overlapping error bars, the latency increases are small in absolute numbers: from 5.70ms to 7.17ms for 8 analytics, and from 152.57ms to 191.81ms for 64 analytics. For other attempted update rates with which we experimented, ranging from 20,000 op/s to 100,000 op/s, the overheads are also negligible, increasing the update latency in at most tens of milliseconds.

Remset Sharing and Memory Reclamation. To minimize the additional storage required to support the virtual snapshots, the VLS algorithm shares a single remset among multiple concurrent analytics and performs immediate memory reclamation when possible. In practice, these features lead to substantial savings. We measured in the 100 million-record table, for different number of concurrent queries (8, 16, and 64) and attempted update rates (10,000 and 100,000 op/s), (i) the size of the shared remset, (ii) the total size of individual remsets (one remset per scan), and (iii) the size of a shared remset without immediate reclamation, i.e., records are only removed from the remset after the analytics finish. The results are summarized in Figure 11, which shows how the remset size (in log scale of number of records) varies over time.

The maximum size of the shared remset in the presence of 8 concurrent queries with an attempted update rate of 10,000 op/s (Figure 11a) is around 4.3 million records (1.04 GB). In contrast, when using 8 individual remsets, the number of records reaches almost 34 million (8.20 GB), i.e., 7.9 times bigger than the shared remset. As expected, for a larger update...
rate (Figure 11b), the queue size increases, but the effectiveness of our solution is similar: the shared remset reaches 4.5 million records (1.08 GB), while, for 8 individual remsets, the size is 35 million records (8.44 GB). When running more concurrent queries, the savings are even greater: the number of records when using 16 individual remsets is about 15.9 times bigger compared to the shared remset (Figure 11c and 11d); for 64 analytics, the shared remset reduces the size by a factor of about 60 (results not depicted due to lack of space). Note that the remset size for 16 concurrent analytics is smaller than for 8 queries, which is an expected result: the higher the number of analytics, the higher the concurrency, and as a consequence, less updates are applied to the database. Figure 11 also shows additional savings (the shaded section) attained by the early reclamation of unused records compared to a reclamation that only takes place after a scan finishes.

**Varying the Size of Bit Vectors.** Since the VLS algorithm relies on bit operations and bit vectors, it is important to understand how the size of such vectors \(N\) impacts the performance. We compared, for the 100-million record table, the results on query execution time, throughput, and latency for two values of \(N\): 64 bits (value used in the aforementioned experiments) and 128 bits; our server’s registers are 64 bits wide; due to lack of space, we do not present plots for the comparison. When using 128-bit vectors, the duration of an analytics without concurrent updates increases by 9% compared to the performance for 64-bit vectors, while it increases by 6% in the presence of concurrent updates. Overall, the overheads on update throughput when using 128-bit vectors increases by at most 9% for different number of concurrent analytics when compared to the 64-bit results; with respect to update latency, the overhead is bigger—at most 30%—but in absolute numbers, the latency increases by no more than 1.20ms. These results show that, as expected, setting \(N\) to the same number as the CPU’s register and instruction width (in our case, 64) provides the best results, since native machine instructions can be used more efficiently. Note that, although the overheads are higher for 128-bit vectors, they are still negligible: most of these results, in particular for update throughput and latency, have overlapping error bars when compared to MongoDB. It is worth mentioning that there are servers with up to 512-bit registers\(^4\), thus enabling the maintenance of up to 512 snapshots concurrently; to run a number of concurrent analytics larger than \(N\), an approach that allows snapshots to be shared by multiple queries could be used, as discussed in Section III-A.

\(^4\)Modern Intel CPUs have a width of up to 256 bits, and CPUs that have the AVX-512 extension support 512-bit operations.

**V. RELATED WORK**

The impact of long-running queries on updates has long been acknowledged as an important issue. Replication can be used to segregate workloads, with long-running queries executing on a replica. This reduces the impact on updates at the cost of higher space requirements and analytics computed on stale information. In regard to this issue, Krishnan et al. [15] propose an alternative model to get approximate (yet fresh) results for analytics from stale views using techniques akin to data cleaning. DBMSs can also trade consistency for performance using cursor stability [4], which avoids blocking updates by locking one row at a time during scans. Our approach can be thought of as a way to provide consistent results with performance approaching that of cursor stability.

Snapshot semantics can be implemented with MVCC (Multi-Version Concurrency Control) either by storing multiple versions of the database or by dynamically reconstructing an earlier snapshot of the data using the database log [16]–[18]. In contrast, our approach focuses on enabling snapshot semantics for analytics without the need of full-fledged versioning to limit storage usage. VLS shares similarities with transient versioning techniques [8]–[10], [19], [20], since they rely on versions created specifically to support consistency for OLAP queries. However, these approaches often assume the presence of a transaction manager, making it hard to adapt to current NoSQL stores. In addition, some of these techniques [19] are designed in the context of main-memory databases, while others [9], [20] have significant initialization and communication costs, and implement versioning at the page—instead of record—level, which potentially creates unnecessary duplicates and space management problems. VLS can be seen as a snapshot isolation technique [7], [21] that is lightweight and amenable for implementation in NoSQL systems, since it does not require any transaction manager; differently from Padhye and Tripathi [21], our goal is to neither bring the entire transaction semantics to NoSQL stores, nor rely on native database versioning.

There has been renewed interest in systems that support both OLAP and OLTP workloads. HyPer [22] is an in-memory database that supports snapshots for OLAP-style queries through the fork system call. SAP HANA [23] propagates records through different stages, including a write-optimized storage format (for OLTP) and a read-optimized storage format (for OLAP). Hekaton [24] is a memory-optimized OLTP engine for SQL Server that uses an MVCC approach to isolate read-only transactions from updates. HYRISE [25] is a main-memory store that focuses on providing different
physical designs for different workloads, but it does not handle aspects related to consistency for hybrid workloads. Note that these systems are specific to in-memory stores, which have fundamentally different requirements from disk-resident storage [26]. Last, R-Store [27] relies on database versioning to support real-time OLAP with OLTP in the same system; it extends HBase as their versioned storage backend, and uses HStreaming to maintain a real-time data cube.

VI. CONCLUSION AND FUTURE WORK

As a step towards closing the gap between big and fast data, we proposed virtual lightweight snapshots (VLS), a technique that supports consistent analytics without blocking concurrent updates. VLS enables analytics and updates to run in the same system, leading to fresh results that are also correct and consistent. VLS does not require native support for transactions or database versioning, and thus, it can be combined with a number of NoSQL systems. We integrated VLS with MongoDB and performed a detailed experimental evaluation that showed that the overheads incurred by VLS are low and have little impact on query evaluation time, update throughput, and latency. These results suggest that VLS can bring consistency to analytics—while producing correct results—at a low cost, being a practical and efficient solution to support OLTP and OLAP in the ODM Insights middleware.

As future steps, we have plans to include MongoDB–VLS in ODM Insights so as to provide consistency in a future release of the middleware. Although many current use scenarios are non-distributed, we are extending our approach to provide support for distributed data: each node would have their own remset and copies for the global masks, and these copies would have to be synchronized whenever a scan begins and ends. However, the synchronization costs and the impact on the index structure are still being investigated. Finally, we are considering additional improvements to the underlying data structures (e.g., using compression to further reduce memory usage) and to the algorithm design (e.g., adapting concepts behind parallel garbage collection to parallelize the approach, and allowing several queries to share a single snapshot).

ACKNOWLEDGMENTS

This work was supported in part by a Google Faculty Award, an IBM Faculty Award, the Moore-Sloan Data Science Environment at NYU, DARPA, and NSF awards CNS-1229185 and CNS-1405927.

REFERENCES


