April 10, 2019

Stateful metadata for big data

Nagaraju Pothineni

Nitin Raut

Nikhil J. Joshi

Nikhil Menon

Stateful metadata for big data

ABSTRACT

Large volumes of data, characterized by large variety and high update velocities, pose challenges in terms of storage, application of concurrently occurring frequent updates, and serving processes that require the most accurate version of the data simultaneously. In most current schemes, it is not possible to guarantee all of these characteristics and a relaxing one or more requirements is necessary. The present disclosure describes a scalable, easy-to-maintain metadata mechanism that is fast and efficient to update, and can provide all the above guarantees on data. The metadata maintains lightweight validity markers, and simple algebra is performed thereof to surface the most up to date and accurate data while enabling constant updates to the data in a non-blocking fashion.

KEYWORDS

- Big data
- Stateful metadata
- Concurrent update
- Non-blocking update
- Validity marker

BACKGROUND

Most big data (BD) systems exhibit extremely large volumes of incoming data with immense variation, usually collected at enormous velocities. BD systems often have associated serving pipelines - such as e-commerce, advertising, and analytics or machine learning modeling - that utilize this data as input. Since the incoming data is received asynchronously from multiple uncorrelated sources, the data as received is not in a structured format that is readily suitable for serving and often requires separate offline data *preprocessing*. As a result, most functional data is usually stored in a structured format, which can make further mutations to the data computationally expensive and time consuming. On the other hand, the data-preprocessing logic may need frequent upgrades to mitigate issues and bugs encountered in the past, and may require some or all of the data previously processed to be *reprocessed* and be corrected. In addition, there can exist data *removal* processes - such as regulations that require that the users have control over their data - requiring BD systems to remove portions of stored data. Data mutation is a time consuming, and computationally expensive process in most BD systems, while serving pipelines, such as e-commerce and real-time bidding, require instantaneous provision of the most up to date and accurate data at all times. Fig. 1 illustrates the issue at hand.

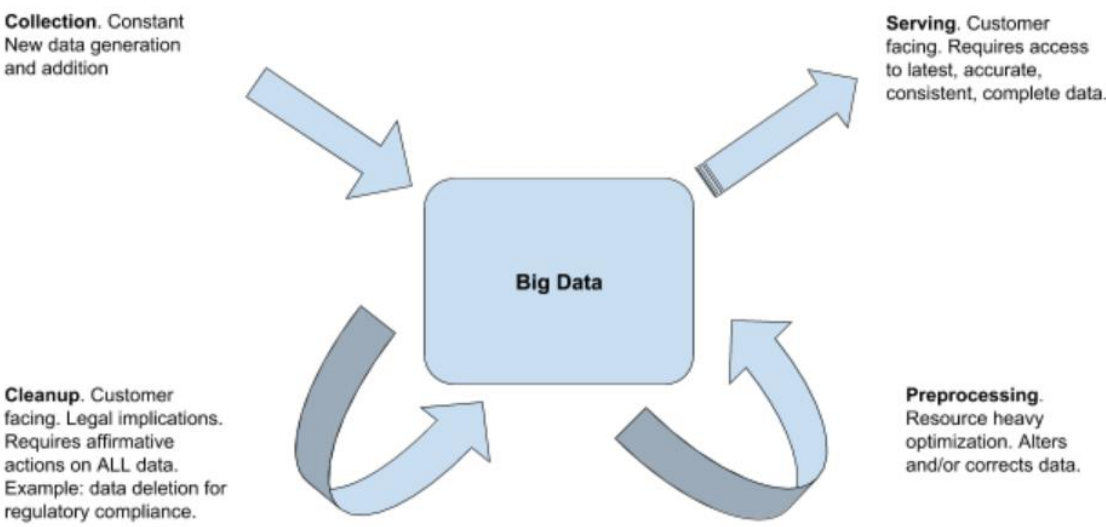

Fig. 1. Evolution of Big Data. Data needs to be in the most updated, easy-to-process form at all times, while many conflicting mutations occurring constantly and concurrently. Satisfying all requirements simultaneously is not be possible, unless provided with unlimited resources, often faced with one of the scenarios: either complete, but stale or inaccurate data or accurate but incomplete representation.

In the presence of such conflicting operations concurrently mutating the same data, it is non-trivial to maintain, at all times, the qualities of (a) *completeness* or provision of all the relevant data; (b) *correctness* or provision of only the relevant data; and (c) *consistency* or provision of causally connected copies of the data. The most commonly applied solutions involve performing modification to the data in a blocking manner, where one of the conflicting operations is allowed to proceed while other operations wait for their turn, thus artificially breaking concurrency. The only options available in such cases are to serve either incomplete but accurate data, with temporary hiding of the data being mutated, or complete but stale and inaccurate data, where older copies of data are provided for a period of time until the replacement is available. Another commonly applied solution is to allow these operations to proceed concurrently in a non-blocking fashion, but selecting only one of the copies in case of conflicting updates, rerunning operations on the new version of the data. This however causes a lot of throw-away data that needs to be rejected in order to incorporate recent changes, thus incurring significant costs in processing time and resources usage.

DESCRIPTION

The present disclosure describes a metadata mechanism that preserves correctness, consistency, and completeness guarantees in the surfaced data, while simultaneously permitting frequent and concurrent mutations to large volumes of the underlying data. This mechanism enables applying the data mutations in a non-blocking fashion, effectively saving significant time and processing power, which otherwise is not possible in most current systems. The disclosed metadata mechanism is efficient, fast to update, and cheap to maintain. It acts as an index over the data and provides an effective view of the most updated, complete, correct, and consistent data by way of incorporating *data-invalidity-markers* (masks here onwards) with a *commutative manipulation algebra* defined on the masks. This removes the need for blocking conflicting data updates,, since every update operation can independently introduce some locally latest version of the data by updating the metadata with new masks. The new masks carry the onus of presenting the most updated, complete, and concise view of the current data. The metadata being cheaper to modify and maintain can significantly reduce resource usage by avoiding throw-away work or artificial update latencies due to blocking. By incorporating a simple marker-addition-timestamp, an implicit versioning of the data is achieved, thereby providing rollback facilities in case need arises.

Metadata in its ordinary manifestation is an index over the data. It can include indicators or accessors for the data based on identifiers used to categorize data in a file. For example, if the data is organized by the ascending order of uniquely assigned user identifiers, say *user_id*s, and is spread across multiple files and directories, the corresponding metadata can simply specify paths to all the files for a given user. Such accessors are known collectively as *references* to data.

A mask *m* can be a combination of data attributes identifying the slice of the data (which needs masking and filtering before serving) from the reference on which it has been applied, or it may simply be a flag rendering the reference invalid.

In a simple form, a data mutation that results in metadata masking is as illustrated in Fig. 2.

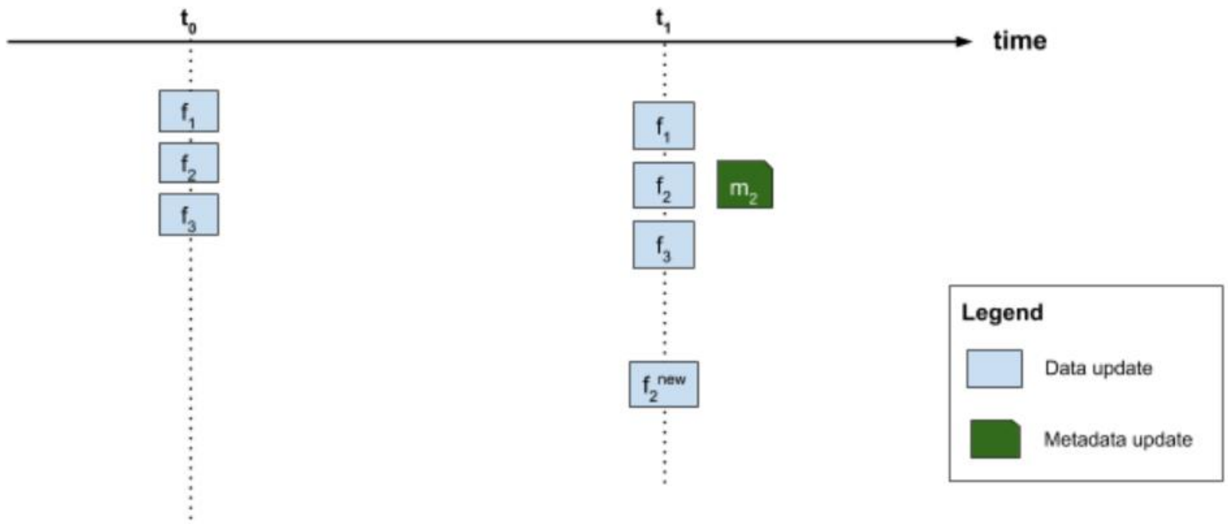

Fig. 2. Singleton data mutation. Boxes in blue represent data, boxes in green represent a metadata reference to a file indexed by the same integer, and are shown next to the data. An update f_2^{new} for some data in file f_2 is committed. A mask m_2 is added to metadata associated with an index to f_2, while adding a new reference to f_2^{new}.

For brevity, all the data files are denoted by blue colored boxes, while references to these data files in metadata are shown in green boxes next to the data files. Only those metadata references are shown that receive updates in an operation. In the update shown in Fig. 2, a new version f_2^{new} of some of the data in file f_2 was added to the system. Consequently, a mask m_2 was added to all the metadata references to the older version f_2. For any metadata query made until t_1 the reference to f_2 is returned if relevant, while those immediately after t_1 surface f_2^{new} and the relevant part of f_2 after filtering out by m_2.

The addition of a mask re-establishes correctness and consistency by enabling the addition of updated data asynchronously while serving query responses that include the latest version of data at all times. A separate process can be used to consolidate f_2 modulo m_2.

While placing a mask to hide now-stale data solves the immediate problem of *output* invalidation, this process is insufficient for *input* invalidation: what if some newly added data invalidates an existing data that is being used in another as input? For example, a process p_1 initiated at time t_0, but at a later time t_1 before it finishes another process p_2 adds an alteration to data that results in invalidating inputs of p_1.

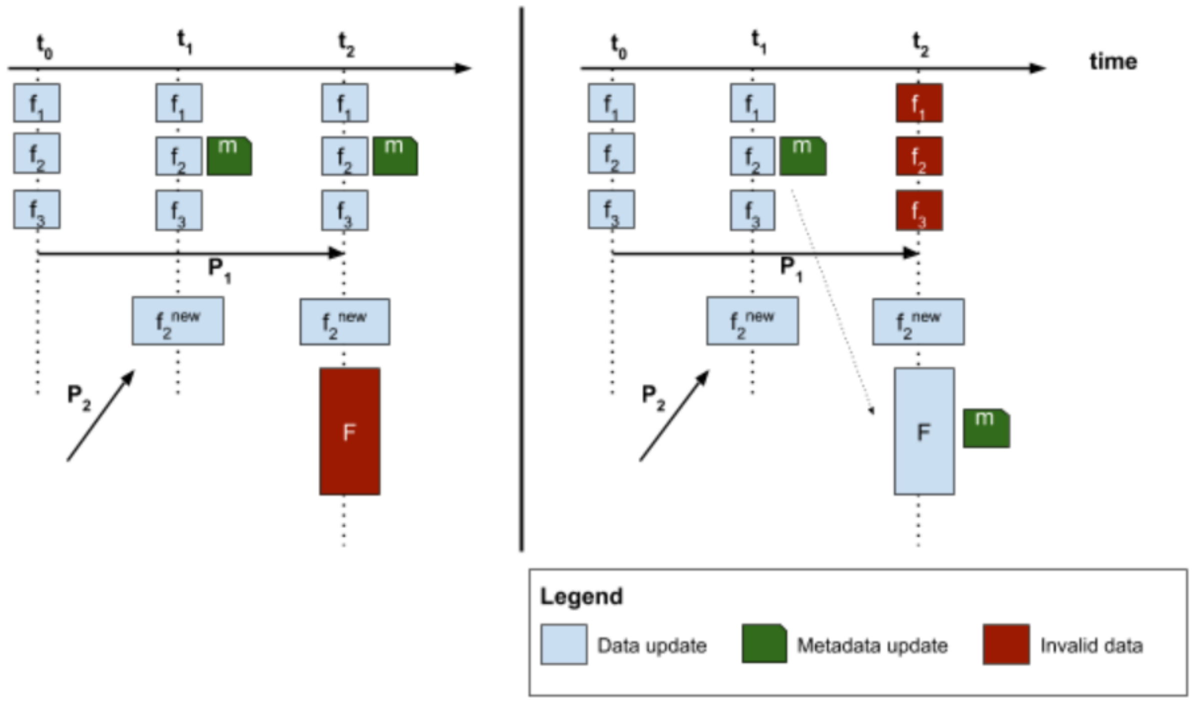

Fig. 3 Metadata versioning. (**Left**) A process p_2 added data invalidating input of an earlier instantiated process p_1. The output of p_1 must be discarded restarting p_1 to include modifications due to p_2. (**Right**) With notion of metadata *versioning*, the *new* masks can be moved to output of p_2 discarding the now-redundant input without explicit reference to p_2.

As shown in Fig. 3 (left panel), the simple addition of mask on the input file f_2 at the arrival of process p_2 at time t_1 resolves only the local conflicts. However if the file f_2 was being used as an input to a separate process p_1 initiated before t_1 the output of p_1 needs to be abandoned, and p_1 needs to be restarted to consume the updated state of the data as input.

It should be noted that this *throw-away-and-restart* mechanism does not provide a failsafe solution, since in most cases it cannot be guaranteed that another update would not be applied until the restarted p_1 finishes, unless the system is blocked for p_1.

To circumvent this, two concepts, metadata versioning and masks manipulation algebra, are introduced. Fig. 3 (right panel) demonstrates metadata versioning and a simple mask manipulation in action. If a process p_1 was associated with the state of the metadata versioned at the time of initiation, i.e. when its inputs were materialized, say ,$\mathcal{M}^{\mathcal{I}(p_1)}(t_0)$ where $\mathcal{I}(p_1)$ is the set of references in input of p_1, and similarly, $\mathcal{M}^{\mathcal{I}(p_1)}(t_2)$ versioned at its completion, then the difference

$$\delta\mathcal{M}^{\mathcal{I}(p_1)} = \mathcal{M}^{\mathcal{I}(p_1)}(t_2) - \mathcal{M}^{\mathcal{I}(p_1)}(t_0)$$

can be used to determine any new masks applied to inputs of p_1 that can consequently be ported over to outputs of p_1.

The metadata versioning in combination with masks manipulation algebra generates the same effect that is achieved by designing a blocking mechanism to allow only one of the mutations at any given time. Fig. 3 shows scenarios where either of p_1 or p_2 is blocked for completion of the other. The left panel shows the metadata states if p_1 was completed before p_2. The metadata state after t_2 is exactly the same as the one in Fig. 3 right panel in presence of metadata versioning and mask algebra, but without explicit process blocking. On the other hand if p_1 was initiated after p_2 updates were applied, the state of the data and metadata individually looks different. However, when viewed in combination the effective state of the system is equivalent to that in the right panel in Fig. 3, i.e. with metadata versioning and masks algebra. In the former, the file F^{new} explicitly contains the updated data from p_2 while invalidating f_2^{new}, which would be represented as the file F with some data masked as per m and complemented with yet valid file f_2^{new} in the latter case.

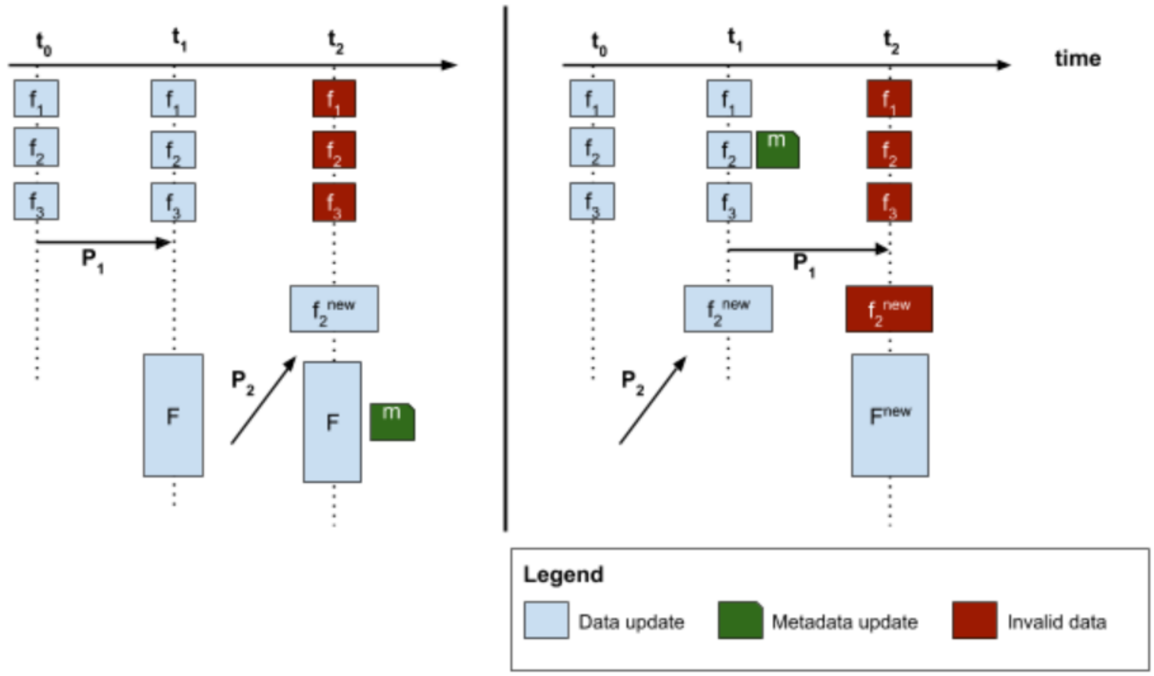

Fig. 4 Updates by blocking. (**Left**) The process p_1 completed before p_2 committed the updates. The state of the system at t_2 is exactly same as if masks were moved from p_2. (**Right**) p_1 started after p_2 updates were committed. While the individual states of data and metadata differ, the combined state of the system remains identical.

Again, by way of metadata versioning and masks manipulation algebra, the need for explicit blocking of the processes or occasional throw-away-and-restart of processes is eliminated. Also eliminated are artificial latencies and servings from stale or incorrect data. More importantly, because metadata only contains references and is usually significantly smaller in size compared to the data, metadata modifications are computationally cheaper and can be done instantaneously, making the latest data available for serving at all times, which is not possible otherwise in any of the scenarios discussed in Fig. 4.

A more involved example of *masks manipulation algebra* is shown in Fig. 5. At its completion the process p_2 adds a new mask m_2^1 to the existing set of masks on f_2 respecting the existing masks. This is done by computing the difference between two metadata versions, one taken at the initiation and the other at the completion of p_2. At the completion of p_1 this change in the metadata for inputs of p_1 is determined by computing yet another difference between metadata versioned at p_1's initiation at t_{n-1} and that at t_{n+1} and is moved to the output of p_1, that is to the references of the file F.

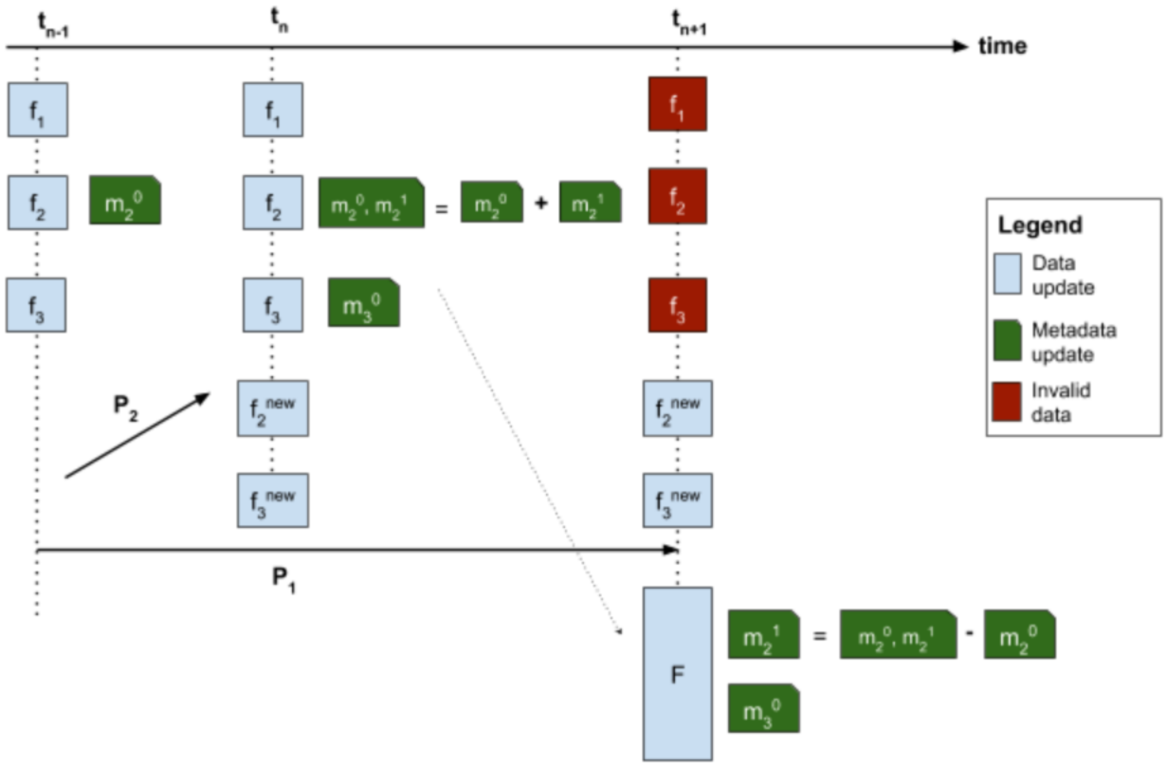

Fig. 5 Example of *Mask manipulation algebra*. The process p_2 at completion adds a new mask m_2^1 to the existing set of masks on f_2 while the process p_1 identifies this change with respect to its initiation and applies the difference to its output.

Throughout the discussion so far, it is assumed that the metadata update is a computationally inexpensive and efficient operation when compared to the actual data mutations, offering near instantaneous updates. This is a reasonable assumption, given that in most cases, metadata is simply a specialized index over the data, responsible for surfacing relevant parts of the data to be processed for information queried. For example, metadata can be maintained in SQL-like relational databases with searchable data attributes as index and data references as value columns.

Metadata versioning can be implemented in numerous ways. Per the techniques described herein, two such methods that are relatively easy to implement are as follows: (a) *Explicit snapshotting*, and (b) *Timestamping masks*. In the former, a metadata snapshot is explicitly included along with inputs to any data mutation process *p*. The process then upon completion can obtain the latest snapshot and determine the changes with respect to the one in input. In *timestamping masks* of implicit metadata versioning, a timestamp is included with every mask applied to the metadata references, enhancing the information of *when* that specific mask was added to the set of masks on a reference. The metadata snapshot at any given time in the past then contains the set of only those masks that were applied prior to the given time. In this case, any data mutating process maintains the start and completion times - usually the current time - and from this, the metadata changes can be derived by calculating the difference between the two timed versions. The latter technique not only eliminates the need to preserve metadata snapshots with every process input (thereby reducing input sizes), but also reduces the complexity of taking explicit metadata snapshot differences. Further, because the versioning information is persisted within the metadata itself, the latter technique automatically provides means of metadata version rollback functionality. Limiting the rollback operations to metadata only, one can achieve similar benefits of fast, efficient, inexpensive mutation to views of the data as in the case of regular mutations.

Masks with manipulation algebra can be maintained as a set of combination of data attributes - that define the filters - with normal set-algebra in its simplest form, although further compaction by considering each individual mask as a set of attributes is possible.

In summary, the disclosed metadata versioning and masks manipulation algebra accrues the following benefits to BD systems, which otherwise would have been available only in systems with static immutable data:

1. *Access to the consistent, complete, and correct data* at all times, irrespective of the amount or frequency of mutations performed simultaneously on the underlying data, which otherwise cannot be guaranteed all at the same time;
2. *Efficient resource usage*, by elimination of the need for either explicit blocking of some of the data mutations or generating frequent throw-away work. On the contrary metadata operations are highly efficient. In addition multiple mutations touching the same data can now be delayed and batched proving further savings on computational resource usage.
3. *Fast, efficient, and inexpensive versioning* of the data, with rollback facility.

CONCLUSION

Large volumes of data, characterized by large variety and high update velocities, pose challenges in terms of storage, application of concurrently occurring frequent updates, and serving processes that require the most accurate version of the data simultaneously. In most current schemes, it is not possible to guarantee all of these characteristics and a relaxing one or more requirements is necessary. The present disclosure describes a scalable, easy-to-maintain metadata mechanism that is fast and efficient to update, and can provide all the above guarantees on data. The metadata maintains lightweight validity markers, and simple algebra is performed thereof to surface the most up to date and accurate data while enabling constant updates to the data in a non-blocking fashion.