# Update Maps – A new Abstraction for High-Throughput Batch Processing

Steffen Viken Valvåg Dag Johansen
Department of Computer Science
University of Tromsø, Norway†
{steffenv, dag}@cs.uit.no

## Abstract

*Key/value databases are popular abstractions for applications that require synchronous single-key look-ups. However, such databases invariably have a random* I/O *access pattern, which is inefficient on traditional storage media. To maximize throughput, an alternative is to rely on asynchronous batch processing of requests. As applications evolve, changing requirements with regard to scale or load may thus lead to a redesign to increase the use of batch processing. We present a new abstraction that we have found useful in making such transitions: the* update map*. It aims to combine the convenience of a key/value database with the performance of a batch-oriented approach. The interface resembles that of an ordinary key/value database, but its implementation can rely on batch processing and sequential* I/O*, for improved throughput. We evaluate our new abstraction by comparing three different implementations and their performance trade-offs. Specifically, we identify some conditions under which update maps significantly outperform commonly deployed key/value databases. Finally, we discuss ways to improve generic batch processing systems like MapReduce as well as traditional key/value databases based on our findings.*

## 1 Introduction

A recurring question in system design is how to store persisted state. Options include using a relational database system, a simple key/value based look-up database, or some application-specific storage scheme built on top of a regular file system. From a performance point of view, the right choice depends on the expected access pattern of the application. Applications that only do single-key look-ups, such as a simple directory service, will typically get better performance using a key/value database than using a full-fledged relational database. Many real-life services only require primary-key access, as evidenced for example by Amazon's Dynamo system [3] and its client services. Similarly, there are applications for which a key/value database would be overkill, such as a simple web server that only needs to serve a static set of files.

The most common storage media are hard disks, where large sequential I/O operations achieve significantly higher bandwidth than small random-access operations, given the mechanical nature of the disk head movements involved. A common approach is thus to accumulate storage requests and apply them in batches, aiming to amortize the cost of random I/O accesses and improve throughput. Taken to its extreme, the *batch processing approach* employs nothing but append-only files, without any supplementing index structures. A batch of queries is processed by a simple sequential traversal of the database. Updates are applied by rebuilding a new version of the entire database, including all updated values using a join algorithm such as hash join [4] or sort-merge join [6]. With sufficiently large batches, this can outperform the alternative of applying each update individually using a random-access data structure.

Many applications are distributed by nature, or handle data volumes that necessitate distributed processing. In a distributed setting, the complexity of implementing a hand-crafted storage solution can be discouraging, given the additional non-functional requirements that distribution entails; e.g., fault tolerance and scalability. One option is to use a high-level abstraction for distributed data processing, such as the widely adopted MapReduce [2] programming model. MapReduce applications have a *map* phase in which all input files are read sequentially, producing intermediate key/value

†This work is supported in part by the Research Council of Norway through the National Center for Research-based Innovation program.

pairs that are subsequently grouped by key and aggregated in a *reduce* phase. How to produce the intermediate key/value pairs, and how to aggregate them, is controlled by supplying user-defined *map* and *reduce* functions; through appropriate choices a basic sort-merge join algorithm can be implemented, for example to apply batches of updates. MapReduce execution environments typically include a distributed file system optimized for high-throughput sequential I/O [5].

In this paper we focus on how to redesign I/O bound applications from using a key/value database tailored for fast synchronous look-ups, to using the throughput-oriented batch processing approach. Such a transition can be motivated for example by a change in requirements or deployment scale, resulting in increased request volumes. Based on our experience from redesigning a real-life application, we present a new abstraction for batch processing, the *update map*. It mimics the familiar interface of a key/value database, but is entirely batch oriented, and can be implemented exclusively using sequential I/O. We present and evaluate three separate implementations of update maps; one that uses the MapReduce model for batch processing, a second handcrafted implementation, and a third reference implementation that is *not* batch-oriented, but instead relies on a traditional key/value database.

The remainder of this paper is structured as follows. Section 2 introduces the update map abstraction and its interface. Section 3 provides an in-depth example showing how a real-life web crawler application has been restructured for improved throughput using the abstraction. Section 4 describes our three implementations of update maps and how they differ from each other. Section 5 evaluates the implementations by comparing the performance of batch-oriented update maps to traditional key/value databases for various batch sizes. We discuss possible applications and adaptations of update maps in Section 6, and conclude in Section 7.

## 2 Update Maps

The drawback of traditional key/value databases is that they require random I/O accesses to handle synchronous look-ups. Our idea is to provide an abstraction that only supports *iteration* over all items, as well as *asynchronous updates*. With this restricted interface, our implementation is able to rely solely on sequential I/O and batch processing.

The basic interface of a key/value database is this:

| | |
|---|---|
| `Get(Key) ⇒ Value` | Synchronous look-up |
| `Set(Key, Value)` | Synchronous overwrite |

Keys can be mapped to values through the `Set` operation, and the value associated with a key is retrieved using the `Get` operation. Since the `Set` operation simply overwrites the old value, updates such as incrementing a value by one must be performed by 1) reading the old value using the `Get` operation, 2) calculating the new value and 3) writing back the new value using the `Set` operation. Not only does this constitute a race condition, it is also inefficient in terms of I/O and latency.

Our update map abstraction addresses this limitation by introducing an `Update` operation that asynchronously updates the value associated with a key, possibly as a function of the previous value. Rather than directly specifying a new value, the `Update` operation accepts an *updater* function which is used to determine the new value. The updater function accepts the previous value associated with a key and returns the new value. Note that the updater function can be evaluated lazily, i.e. its evaluation can be deferred until the new value is actually required.

Iteration is supported through the visitor pattern; the `Traverse` operation accepts a *visitor* function which is invoked once for each key/value pair in the database. By design, there is no facility to synchronously look up the value associated with a single key; a full traversal of the database is required whenever values are to be read. This allows implementations to collect batches of pending updates and apply them lazily at traversal time. For a given application, the suitability of an update map thus depends on the ratio of updates to traversals. The core interface of update maps is summarized below.

| | |
|---|---|
| `Update(Key, Updater)` | Asynchronous update |
| `Traverse(Visitor)` | Iteration |

The following example illustrates the use of the `Update` operation with various updater functions.

```
function two(key, value):
  return 2

function increment(key, value):
  return (value == null) ?  1 :  value + 1

function remove(key, value):
  return null

function multiply(key, value, factor):
  return value * factor

Update(''abc'', two)
Update(''abc'', increment)
Update(''abc'', remove)
Update(''x'', bind(multiply, factor=3))
Update(''y'', bind(multiply, factor=7))
```

The first updater function always returns the value 2, unconditionally overwriting any previous value associated with the key. The second function examines the previous value and increments it by one. If the key had no previously associated value, the special `null` value is passed to the updater function, which in this case sets the initial value to 1. By a similar convention, the third updater function serves to remove a key from the database by always returning `null`. The fourth updater function, `multiply`, is more generic and accepts an additional parameter whose value must be bound prior to invoking `Update`. In our example, it is used first to multiply one value by 3 and then another by 7. The value of the additional `factor` argument is specified using a hypothetical `bind` primitive. For functional languages, the natural way to implement parameter binding would be through partial function applications. For object-oriented languages, bound paramaters can be stored as fields in the function objects.

These examples all ignored the *key* argument, so it might seem superfluous. The main motivation for including it is that it allows updater functions to double as visitor functions, because they have the same type signature. When the `Traverse` operation is used to iterate over all key/value pairs, the visitor function can return a new value for each key that is visited, or remove the key by returning `null`. The same functions can thus be used as updaters to update the values associated with individual keys, or as visitors to update *all* values in the database. The following example illustrates the use of the `Traverse` operation.

```
function removeOdd(key, value):
  if value mod 2 == 1:
    value = null
  return value

function show(key, value):
  print key, value
  return value

Traverse(increment)
Traverse(removeOdd)
Traverse(show)
```

This example first increments all values by 1, then removes all keys with odd-numbered values, and finally displays all remaining keys and values (assuming the existence of a print statement for displaying values).

```
# process downloaded URLs
for url, status in downloads:
  meta = Get(url)
  if status != 200:
    # download error - retry later?
    if meta.retries++ > maxRetries:
      meta.status = gone
    else:
      meta.status = error
  else:
    # download ok - parse links
    meta.timestamp = now()
    meta.status = ok
    for link in parse(meta.content):
      linkMeta = Get(link)
      if linkMeta != null:
        # seen this URL before
        linkMeta.linkCount++
      else:
        # never saw it before
        linkMeta = newMeta(link)
      Set(link, linkMeta)
  Set(url, meta)

  # schedule more URLs for download
  for url, meta in database:
    if meta.status = new:
      if meta.linkCount > 3:
        schedule(url)
    else if meta.status in (ok, error):
      if meta.timestamp < yesterday():
        schedule(url)
```

**Figure 1. Crawler main loop using a key/-value database.**

## 3 Example Application

The update map interface, whose simplicity allows for very efficient implementations, might in return seem overly restrictive for real life applications. We therefore present a more complete example in this section: a large scale web crawler – the application that was our original motivation for the update map abstraction. Web crawlers download web pages and parse them to extract hyperlinks to additional pages, which are again downloaded, forming a continuous work cycle. In many deployments, the web crawler focuses on a narrowed subset of the web, for example a single top-level domain, and thus ignores many URLs. The original web crawler was designed primarily for such use cases, with very flexible policies for scheduling and filtering of URLs to be downloaded. We needed to redesign it for larger crawls; with the total number of available web pages on

```
function urlError(url, meta):
  if meta.retries++ > maxRetries:
    meta.status = gone
  else:
    meta.status = error
  return meta

function urlOK(url, meta):
  meta.timestamp = now()
  meta.status = ok
  for link in parse(meta.content):
    Update(link, addLink)
  return meta

function addLink(url, meta):
  if meta != null:
    meta.linkCount++
  else:
    meta = newMeta(url)
  return meta

function scheduleURL(url, meta):
  if meta.status = new:
    if meta.linkCount > 3:
      schedule(url)
  else if meta.status in (ok, error):
    if meta.timestamp < yesterday():
      schedule(url)
  return meta

for url, status in downloads:
  if status != 200:
    Update(url, urlError)
  else:
    Update(url, urlOK)

Traverse(scheduleURL)
```

**Figure 2. Crawler main loop using an update map.**

the order of a trillion, throughput becomes the main concern even when crawling narrow subsets.

The main bottleneck in the original crawler turned out to be the key/value database used to store various meta-data for each URL. Statistics such as the last download time, the number of failed download attempts, and the number of in-links encountered were being recorded for each unique URL, which entailed enough random I/O accesses to severely limit the overall throughput.

Figure 3 gives a somewhat simplified view of the main loop of the original crawler: a separate download engine manages a large set of concurrent downloads, and the main loop iterates over the downloaded URLs and their HTTP status codes. Failed downloads are retried for a number of times before the URLs are flagged as permanently missing. For successful downloads, a time stamp is recorded and the downloaded content is parsed for additional hyperlinks. For each link, the in-link count of the target URL is incremented, or a new meta-data entry is created if it is the first time the URL is encountered. Finally, the database is traversed in order to schedule additional URLs for download. In this example, new URLs are scheduled for download if they have been encountered in at least 3 links, while previously downloaded URLs are rescheduled for a new download after 24 hours.

Once the set of encountered URLs grows sufficiently large, the frequent meta-data look-ups of the crawler's main loop become a performance problem. The key to optimizing the loop is realizing that the URL meta-data is only ever updated as a function of its previous value. Synchronous look-ups are not required; for example, there is no need to synchronously read the previous in-link count of a URL in order to increment it. However, we are hampered by the interface of the key/value database, which forces us to do synchronous look-ups in order to modify the meta-data. Figure 3 shows how we overcame this in the new version of our crawler, by rewriting the inner loop to use an update map with three different updater functions and one visitor function. The update map interface is thus sufficiently flexible for this application; updates can be processed asynchronously and new batches of URLs for the download engine can be generated by an occasional traversal.

## 4 Implementation

We have developed two batch-oriented implementations of update maps. Our first implementation employs Hadoop, an open-source Java implementation of the MapReduce programming model, designed for simple and efficient batch processing on large data sets. Our second implementation, also in Java, is a hand-crafted version that relies on a basic hash-merge algorithm to apply batches of updates.

We also created a third reference implementation based on the Java version of Oracle Berkeley DB (OBDB), a widely deployed key/value database. OBDB uses a B-* tree and employs a custom log-structured storage for tree nodes. The storage is built on top of the native file system and serves to minimize disk seeks as new tree nodes are created. Nonetheless, look-ups in the tree will inevitably cause random I/O access patterns. The reference implementation is not batch-oriented; it applies each `Update` synchronously,

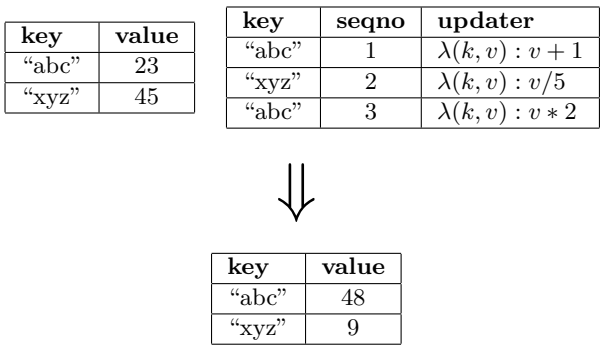


**Figure 3. Applying a batch of updates.**

by looking up the previous value in the B-* tree, invoking the updater function, and writing back the new value. The `Traverse` operation is implemented by iterating over the database using its built-in iteration facilities. We do not use OBDB transactions to apply updates atomically.

Our two batch-oriented implementations are based on the same core idea: implementing the `Update` operation by appending the updates to a log, deferring their actual evaluation until it is required by a `Traverse` operation. With appropriate buffering this only incurs an occasional burst of sequential I/O, which gives the `Update` operation a low amortized cost. The resulting log is a sequence of update records, each of which contains an updater function, its bound parameters, the key to which the update applies, and a sequence number. Updates to the same key should be applied in order of increasing sequence numbers. Since it may be practical or even necessary to split the log into several files, we decided to include the sequence numbers explicitly rather than relying on file offsets for ordering of updates.

In addition to the update log, there is a set of files containing data records, each of which contains a key and its most recently computed value. To implement the `Traverse` operation, all pending updates must first be applied, by joining the update log with the set of data records as illustrated by the example in Figure 3. The figure displays updater functions using lambda notation. There are two pending updates for the key ``abc'': Incrementing its value by one, and multiplying the value by two, in that order. The ``xyz'' key has one pending update: Dividing its value by five. The set of data records is joined with the set of update records, producing a new set of data records; once this is done, the two input sets can be discarded. Our two implementations differ in how this join is implemented, and in the details of how update and data records are stored.

Hadoop features a distributed file system, similar to the Google File System [5], optimized for sequential access and large files. Our Hadoop-based implementation of update maps stores data records in one set of files, and update records in another set. To perform the join, all of the files are used as input to a MapReduce job, whose map and reduce functions are shown in Figure 4. Since the input will include both data records and update records, the map function must handle both kinds. Either way, all it does is emit the record unchanged. The MapReduce framework then does the actual work of grouping the records by key and proceeds to invoke the specified reduce function once for each unique key. The first argument to the reduce function is the key, and the second argument is an iterator for all records having that key. There is at most one data record for each key, which holds the most recently computed value. The remaining records are pending update records for the key. The updates are sorted by sequence number and applied in turn. Finally, the visitor function is invoked, as required by the `Traverse` operation. Unless the final value is `null`, the resulting key/value pair is emitted by the reducer function, and thus ends up in the new version of the database.

While our MapReduce-based implementation of update maps is quite simple, it has some drawbacks. First of all, MapReduce is not ideally suited for implementing joins, as has also been noted elsewhere [10]. The input to a MapReduce job must be modelled as a single homogeneous set, so there is no concept of a left-hand and right-hand side to facilitate a join. For each key, the reduce function must consequently iterate over all input records to locate the single data record that holds the previous value. Meanwhile, the set of updates for the key must be buffered and subsequently sorted. If there is a large number of updates for a single key this could even require external buffering and sorting.*

Another problem is that of data partitioning. The most expensive part of a MapReduce job is to group the intermediate key/value pairs by key [2]. The framework will process each input file sequentially and hash each record key in order to group it. This is done both for data records and update records, ensuring the updates for a given key end up being processed by the same reduce task as its current data record. Since our update map implementation uses an *identity map function* that emits all records unchanged, the only purpose of this initial pass over the data is to group records correctly.

*In practice, this does not occur in any of the experiments that we used for evaluation.

```
function Map(record):
  emit (record.key, record)

function Reduce(key, records, visitor):
  value = null
  updates = [] # empty list
  for record in records:
    if record is a data record:
      value = record.value
    else if record is an update record:
      updates.add(record)
  updates.sort() # by sequence numbers
  for update in updates:
    value = update.updater(key, value)
  if value != null:
    value = visitor(key, value)
  if value != null:
    emit (key, value)
```

**Figure 4. MapReduce-based Implementation of Update Maps**

Our second batch-oriented implementation, which we will refer to as the Hash-Merge implementation, addresses the above problem by ensuring that all records are grouped correctly at the time they are first written to disk. The key space is hashed into a number of buckets, and each bucket contains a separate file for data records, as well as a separate update log. `Update` operations are implemented by hashing the key to be updated, and appending an update record to the corresponding update log. Data record files are maintained in sorted order. To apply the updates in a given bucket, the update log is sorted in-memory and the updates are merged with the data records, reading the old data record file sequentially while writing a new one sequentially. This procedure can be applied separately for each bucket, whenever an update log is about to grow too large to fit in main memory. When executing a `Traverse` operation, each bucket can be processed in turn, so the main memory only needs to accomodate one update log at a time.

For larger data volumes, and regardless of the implementation on a single node, a distributed update map can always be constructed as follows. The key space is partitioned, for example using range partitioning or by hashing keys, mapping each key to a partition. Each partition is assigned to a separate node and implemented exactly as a local (single-node) update map. Updates to a distributed update map are implemented by selecting the partititon to which the key maps, dispatching the update to the corresponding node, and applying the update to the local update map of that node. Traversals are implemented by traversing all local update maps in parallel. Since this simple distribution scheme is applicable for all implementations of single-node update maps, we only focus on how to implement update maps locally on a single node, and will evaluate our implementations based on their single-node performance.

## 5 Performance Evaluation

In this section we compare the performance of our three update map implementations and attempt to identify the circumstances, if any, under which each is preferable to the others. The motivation for the update map abstraction is to improve throughput for I/O-bound applications by avoiding random accesses. By applying updates in batches, rebuilding the entire database for each batch, only sequential I/O accesses are required. In return, the data *volume* that is read or written is much larger for a complete rebuild than using a random-access data structure such as a B-* tree. As such, the relative performance of our implementations depends critically on the batch size, i.e. the number of `Update` operations that are performed on average in between each `Traverse` operation.

To compare our implementations we populated three sample update maps (one instance of each implementation) with identical data sets. While synthetic, the data set was constructed to mimic the meta-data that is maintained by a web crawler such as the one we described in Section 3. It contains $42$ million keys and values; the keys are integers drawn from a key space of $100$ million, so when updates are applied to random keys, $42\%$ of them will on average update old values while the rest will insert new values into the database. The values are variable length, chosen randomly to be from $0$ to $500$ bytes long.

As described in Section 4, the on-disk data structures of our two batch-oriented implementations are simple append-only files containing sequences of key/-value pairs. For our sample database, this amounted to about $11$ GB of data on disk. In contrast, the same data set had a footprint of about $24$ GB when stored in an OBDB database. The difference in footprint is unsurprising, since our batch-oriented implementations employ no auxiliary data structures, whereas the OBDB implementation is based on a B-* tree, and needs additional space for node pointers. Furthermore, OBDB uses a log structured storage for tree nodes which needs to be compacted periodically to collect garbage and reclaim disk space.

We used a machine with $8$ GB of main memory and two quad-core Intel Xeon E5335 $2$ GHz processors with

8 MB level 2 cache, running the CentOS 5 Linux distribution, Java 1.6, version 3.3.74 of the Java OBDB, and Hadoop version 0.18.1. Our experiments used a single Java process that was limited to 2 GB in heap space, but the remainder of main memory was available for disk block caching.

Once our databases were initialized, we ran a series of experiments on each of them, applying a varying number of random updates. Each update appended some random data to the value associated with a key, or inserted a new value if the key was missing. The two batch-oriented implementations process updates by appending them to a log, so in order to evaluate their true amortized cost, we subsequently traversed the database once to read all the new values. The workloads thus consisted of a variable number of `Update` operations followed by a single `Traverse` operation. In the OBDB implementation all updates are applied synchronously, so we omitted the traversal and did not include it in the cost of processing the updates.

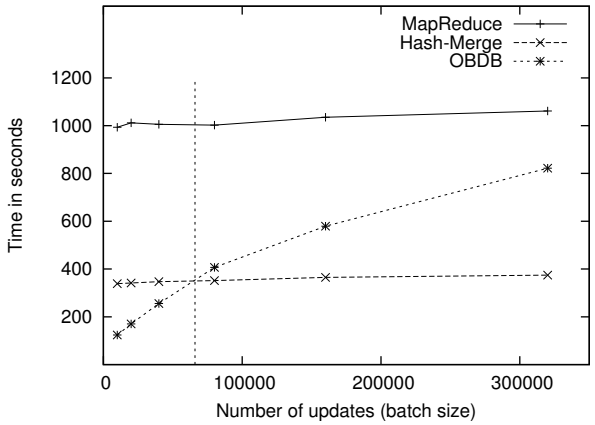


**Figure 5. Time to apply a number of updates to our sample database, for each of the implementations.**

Figure 5 shows the results, with the number of updates applied on the X axis and the number of seconds elapsed on the Y axis. The time required by the OBDB implementation increases with the batch size, since each update requires a look-up in the underlying B-* tree. Main memory caching helps, but there is only room for the topmost levels of the tree so the benefit is limited, as shown by the near-linear increase in execution time. In contrast, the time required for the Hash-Merge implementation is nearly constant, independent of batch size. The only varying factor is the number of updates that are logged, and their size on disk is small compared to the total size of the database. The `Traverse` operation will read and write the entire database, merging in the updates. It is surprising how few updates are required before the Hash-Merge implementation outperforms the OBDB implementation. The crossover point, marked by a vertical dashed line in the figure, is at around 66000 updates. In other words, if a mere 0.15% of the values are to be updated in between each traversal, the Hash-Merge implementation is preferable.

We were also surprised to see how poorly the MapReduce-based implementation performed. Compared to the Hash-Merge implementation we observed a constant overhead of about 250%. This certainly justifies hand-crafting the Hash-Merge implementation rather than relying on the higher-level MapReduce model implemented by Hadoop. The main advantages of our hand-crafted version is the ability to do all sorting in-memory, and to correctly partition the data when it is first written to disk. The MapReduce-based implementation needs extra passes over the data for partitioning, and may resort to external sorting.

## 6 Discussion

The update map abstraction takes batch processing to the extreme. The fundamental premise of the abstraction is to avoid random I/O accesses, to maximize throughput. Our Hash-Merge implementation achieves this by using append-only files which are only ever read sequentially, while merging in updates. Still the update map interface resembles that of a key/value database and aims to ease the transition from using synchronous look-ups. The web crawler example from Section 3 shows that the restricted interface can be sufficiently expressive for complex real-life applications.

Our approach of logging all updates and applying them asynchronously has certain additional benefits, besides performance. When the update log is flushed to disk, updates are guaranteed to be persisted and visible to future traversals. In other words, batches of updates can be committed atomically without additional complexity. Various semantics for concurrent updates can be implemented, depending on how sequence numbers are assigned; for example, they can be assigned by a centralized component, to implement a total ordering. In short, the update logs serve not only to batch updates but also as a potential write-ahead log.

Recent years have seen the emergence of several batch-oriented programming models [1, 2, 7, 10], such as MapReduce, and even higher-level abstractions, such

as Pig [8], that builds on top of MapReduce, and Oivos [9], which compiles high-level declarative programs into low-level dataflow graphs. In these systems, there is a central *table* abstraction which encapsulates a data set, and tables are transformed using various operators, supplemented by user-defined functions. Table operations are generally executed using sequential I/O, and multiple tables can be merged to implement joins.

In our experience, rewriting existing applications to fit into one of these frameworks can pose a greater obstacle than adapting to the semantics of an update map, for applications that originally rely on key/value databases. Table-oriented systems could address this by adding support for asynchronous updates of individual records, in additon to their existing operators, which transform entire tables. The implementation would rely on the pre-existing support for merging tables, and simply merge in an update log whenever a table is read.

Another approach is to include an asynchronous `Update` operation in the interface of a key/value database. Pending updates could be buffered in memory or stored in a separate B-* tree, to be applied in batches, amortizing the cost of modifying the main tree structure. While a number of clever implementations may be possible, the first step must be to offer the support for asynchronous updates as part of the database interface (akin to supporting SQL update statements).

## 7 Concluding Remarks

In this paper, we set out to explore the feasibility of restructuring real-life applications to rely exclusively on sequential I/O, and the performance gains that could result from doing so. Applications that rely on synchronous look-ups in a key/value database abstraction would seemingly be hard pressed to make do without random I/O accesses. However, many synchronous look-ups can be rewritten as asynchronous updates, paving the way for batch processing. The large scale web crawler we have described exemplifies this, as it was successfully rewritten to use our update map abstraction.

The exact performance characteristics of an update map depend on several factors, such as the size and disposition of keys and values, the hardware employed, and not least the access pattern of the application. Intuitively, a complete rebuild of the entire database for every batch of updates seems prohibitively expensive. As such, we were surprised to discover how few updates were required to justify it: for our example database, if a mere $0.15\%$ of the items are to be updated, our batch processing approach outperforms a conventional key/value database. What appears to be brute force is actually a powerful technique to improve application throughput.

## Acknowledgements

We thank our colleagues in the iAD project, in particular Johannes Gehrke and Åge Kvalnes, for their helpful input on earlier drafts of this paper.

## References

[1] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation*, pages 137–150, 2004.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 205–220, 2007.

[4] D. J. DeWitt and R. H. Gerber. Multiprocessor hash-based join algorithms. In *Proceedings of 11th International Conference on Very Large Data Bases*, pages 151–164, 1985.

[5] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 29–43, 2003.

[6] G. Graefe. Sort-merge-join: An idea whose time has(h) passed? In *Proceedings of the Tenth International Conference on Data Engineering*, pages 406–417, 1994.

[7] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 EuroSys Conference*, pages 59–72, 2007.

[8] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1099–1110, 2008.

[9] S. V. Valvåg and D. Johansen. Oivos: Simple and efficient distributed data processing. In *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications*, 2008.

[10] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker, Jr. Map-Reduce-Merge: Simplified relational data processing on large clusters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1029–1040, 2007.