Lazy Analytics: Let Other Queries Do the Work For You

William Jannen, Michael A. Bender, Martin Farach-Colton†, Rob Johnson,
Bradley C. Kuszmaul‡, and Donald E. Porter
Stony Brook University, †Rutgers University, and ‡Massachusetts Institute of Technology

Abstract

We propose a class of query, called a derange query, that maps a function over a set of records and lazily aggregates the results. Derange queries defer work until it is either convenient or necessary, and, as a result, can reduce total I/O costs of the system.

Derange queries operate on a view of the data that is consistent with the point in time that they are issued, regardless of when the computation completes. They are most useful for performing calculations where the results are not needed until some future deadline. When necessary, derange queries can also execute immediately. Users can view partial results of in-progress queries at low cost.

1 Introduction

Queries on production databases have varying requirements for response time and data timeliness. Some transactions service end-user requests, and must minimize latency in order to minimize user-perceived delays. Other queries are not urgent, and hence can be scheduled opportunistically, but nonetheless need a specific point-in-time-consistent view of the data. Examples of the second class of queries include periodic reports and summary computations, such as issuing monthly bills, identifying patterns in online purchases, and monitoring trends in social media.

Long-running summary computations can starve other high-priority, latency-sensitive tasks, if both classes of operations are run on the same machine. To alleviate resource contention on production databases, it is common to maintain replicas or additional databases where summary computations are performed [2]. This may require additional physical resources, management effort, and/or licenses, and requires keeping multiple databases in sync.

We propose a new class of query for summary computations that can minimally impact other operations. A *derange query* maps a function over a range of records, and incrementally aggregates the result. Derange queries defer work until it is necessary (e.g., the result of the query is needed), or convenient (e.g., other necessary work has read the required data into memory). Thus, derange queries are most useful for calculations whose results are needed at some future deadline. However, once issued, derange queries can be scheduled immediately.

A key idea underlying the derange query model is to integrate background work with I/O scheduling. The goal of a derange query is to make maximum use of all I/Os in the system; when any query executes, we want to amortize the I/O cost of that query across as many active queries as possible. At the same time, we do not want background tasks to impact latency-sensitive operations negatively. The derange query model allows one to integrate these goals into one I/O scheduler.

Derange queries can be easily implemented as messages in a write-optimized dictionary (WOD), such as a B^{ε}-tree [7], a log-structured merge tree [15], or a log-structured merge tree variant [5, 18, 19, 21]. As the name implies, WODs are popular for high-performance databases and file systems [4, 9, 10, 11, 12, 13, 14, 16, 17, 19, 20] because of the very high insertion performance—typically less than 1 I/O per insertion or deletion. WODs are so fast because they buffer and batch writes. The primary focus of write-optimization has been on improving the efficiency of *writes* through batching.

This paper identifies an opportunity to integrate write batching in a WOD with background queries that access the same data. There are several benefits to implementing a derange query as a message in a WOD:

- Derange queries on overlapping input ranges can be transparently batched and processed together, requiring each input value be read only once.
- Repeated derange queries at multiple points in time on the same input range may complete by reading every version of the data exactly once.
- I/O required to ingest new data can contribute to completing a derange query, and I/O required to process a derange query can accelerate ingesting new data.

Derange queries can significantly reduce the cost of summary computations on highly volatile data sets, and could make data analytics possible on high performance production databases without harming update performance. In fact, the higher a data set's update rate, the faster a derange query would complete.

The remainder of this paper is organized as follows. Section 2 discusses WODs, and explains how the properties of B$^{\varepsilon}$-trees apply to derange query design. Section 3 outlines the proposed derange query implementation using a concrete example. Section 4 reasons about derange query performance. Section 5 presents scenarios where derange queries are particularly beneficial. Section 6 summarizes related work, and Section 7 discusses opportunities for future exploration.

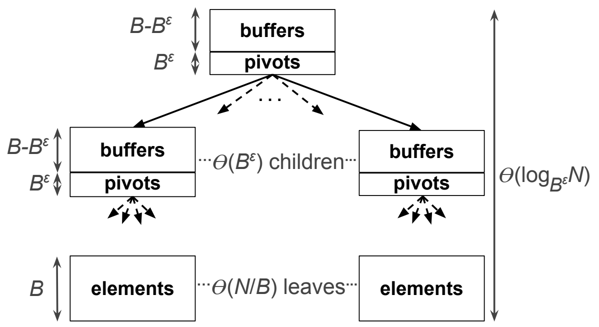

Figure 1: A B$^{\varepsilon}$-tree. Internal nodes store pivot keys and child pointers, and leaf nodes store key-value pairs. Internal nodes also allocate a buffer to store messages, which are flushed down the tree in batches.

2 Write-Optimized Dictionaries

This section explains B$^{\varepsilon}$-trees [7], an example of a write-optimized dictionary (WOD). Derange queries could be implemented in other WODs, including LSM-trees [15] and their variants [18, 19, 21]. However, our proposed implementation relies heavily on upsert operations, and B$^{\varepsilon}$-trees have asymptotically superior upsert performance.

We limit our discussion to the features of B$^{\varepsilon}$-trees that are most relevant to derange query design. Bender et al. [6] offer a more complete description of B$^{\varepsilon}$-trees, including comparisons with other WODs.

2.1 B$^{\varepsilon}$-Trees

A B$^{\varepsilon}$-tree, like a B-tree, is a search tree for organizing persistent data. Internal nodes store pivot keys and child pointers, and leaf nodes store key-value pairs. What sets a B$^{\varepsilon}$-tree apart from a standard B-tree is that internal B$^{\varepsilon}$-tree nodes also allocate a buffer to store *messages*. The structure of a B$^{\varepsilon}$-tree is illustrated in Figure 1.

Messages encode updates to key-value pairs. All messages are inserted into the B$^{\varepsilon}$-tree root, and when the message buffer fills in a root or other non-leaf node, messages in the full buffer are flushed to one or more children. Flushing moves messages from a parent to a child's buffer; flushes may cascade down the tree; and messages are ultimately applied to key-value pairs at a leaf. The flushing process estimates the children that would receive enough messages to amortize the cost of rewriting the parent and child buffers. Thus, messages make their way down a root-to-leaf path in batches, until they are eventually applied at a B$^{\varepsilon}$-tree leaf.

Upserts. B$^{\varepsilon}$-trees can effectively implement blind operations—operations on a key-value pair without first reading it—using *upsert* messages.

An upsert message specifies a key, a function, and a set of function arguments. When a key-value pair is queried, all upsert messages along the pair's root-to-leaf path are gathered, and their functions are applied in order.

Upserts can be used to compactly encode updates to ranges of bytes within a object, modifications to fields of structured data, or data-dependent computations. The flexibility of upsert messages is essential to the implementation of derange queries; as described in Section 3, upserts allow derange queries to incrementally and lazily aggregate the results of deferred work.

Temporal ordering. The relative position of messages within buffers of a B$^{\varepsilon}$-tree preserves the temporal order of updates. At any point in time, multiple versions of a key-value pair may exist in the tree (e.g. an insert message overwrites an existing key-value pair), and multiple in-flight messages may contain updates to a given value (e.g. two upserts target the same key-value pair). Node flushing preserves the message ordering until messages are applied to key-value pairs at B$^{\varepsilon}$-tree leaves.

Queries. All messages needed to answer a query reside in buffers on the root-to-leaf search path. Because non-leaf messages may contain outstanding updates, all messages along the root-to-leaf path must be searched, and updates are applied in reverse chronological order.

Message targets. A single message may apply to one key-value pair, all key-value pairs (broadcast), or a range of key-value pairs (rangecast). A rangecast message [22] is addressed to a contiguous range of keys, specified by a beginning and ending key, inclusive.

Since broadcast and rangecast messages may apply to many key-value pairs, these messages may *split* during a node flush. When a message splits, the original message is discarded, and new messages with appropriate subranges are created in its place.

3 Derange Query Design

A derange query can be implemented as a rangecast upsert message. A derange query has the form

$$\textsc{Derange}(R, \textsc{Filter}, \textsc{Map}, \textsc{Fold}, k)$$

where

1. R is an input range.
2. FILTER is a predicate to remove records that do not meet appropriate criteria.
3. MAP is a function to apply to each record in the input range that meets the filter criteria.
4. FOLD is a function to propagate the results.
5. k is a key specifying where results are accumulated.

The aggregation record associated with key k is incrementally updated as a derange query lazily completes. After each application of MAP to an input record, outputs are accumulated by inserting a message of the form:

$$\textsc{Upsert}(k, \textsc{Fold}, \text{result}_{\textsc{Map}})$$

Upsert messages offer a flexible means to propagate MAP results. Upserts can encode complex data-dependent operations as well as simple operations like incrementing a counter. Inserting small upsert messages into the root of a B$^\varepsilon$-tree imposes little I/O overhead.

3.1 Derange Query Example

To get a feel for how a derange query works, we will show how a fictional online retailer, called "Marketplace", could use derange queries for data analytics.

Suppose Marketplace manages its inventory using a product database with records of the form:

```
Item {
        productId    :   num
        warehouse    :   address
        quantity     :   num
        value        :   num
        price        :   num
}
```

Every hour, Marketplace would like to calculate the cumulative value of all products in its New York warehouses in order to identify trends and make inventory decisions. Marketplace could perform these calculations with a derange query where:

R	=	$(-\infty, \infty)$
FILTER	=	return Item.warehouse == NY
MAP	=	return Item.quantity * Item.value
FOLD	=	totalValue += result
k	=	InventoryAt||TIMESTAMP

Marketplace would first start by initializing its aggregation record, k. In this example, the value of k is a simple integer, totalValue, initialized to 0.

The range $R = (-\infty, \infty)$ means that this query will examine every record in the database. But since the query should only track items in warehouses located in NY, the FILTER function is used to exclude records that do not match this criteria. Note that, if the primary index for the database used geography, the range could select for only records in NY warehouses and avoid reading irrelevant data; the FILTER function can select data based on criteria that is not included in the indexing schema.

When a derange query message reaches a leaf of the tree, the value of each record it observes is the value that existed when the derange query was first issued. At that point, the MAP function is called on all records that fall within R and satisfy the FILTER function. The output of each MAP function—here the total value of a single product in the warehouse's inventory—is propagated to the aggregation record, k, using an upsert where the FOLD function updates k's running total.

This simple example demonstrates the utility that derange queries provide. Marketplace's inventory calculations are performed on views of the data at fixed timestamps, but query results are not needed right away. If a particular region of the tree remains unchanged between two derange queries, then a single I/O will satisfy both operations. However, even when the tree is updated frequently, all derange queries see a point-in-time-consistent view of the data, regardless of when the actual calculation is performed.

3.2 Query Completion

One challenge that arises when lazily executing independent, distributed computations is determining what fraction of the total work has completed. To solve this problem, we add a small amount of bookkeeping to the aggregation record: one required field, *outstandingMessages*, and one optional field, *recordsProcessed*.

The outstandingMessages field is a simple counter. A derange query message may apply to many records in the tree, and as explained in Subsection 2.1, a node flush may cause a rangecast message to split. Each time a derange query message splits, we issue an upsert message to the derange query's aggregation record to increment the outstandingMessages counter. To complete the bookkeeping, we issue an upsert message that decrements the counter when a derange query message reaches a B$^\varepsilon$-tree leaf. The outstandingMessages counter is initialized to 1 in order to account for the initial derange query message inserted at the root of the B$^\varepsilon$-tree.

The recordsProcessed field counts the number of key-value pairs that have folded their MAP results to the aggregation record. Due to the laziness of flushing and the opacity of the internal B$^\varepsilon$-tree structure, an application has no control over the progress of a derange query without manually triggering message flushes. By querying the recordsProcessed field, an application can reason about the meaningfulness of a partially completed result.

4 Derange Query Cost

This section explains how derange queries improve the performance of summary computations in much the same way that WODs improve the performance of inserts and updates.

As explained in Subsection 2.1, a B$^{\varepsilon}$-tree node is only dirtied when a substantial amount of new data is written—enough to amortize the cost of rewriting the parent and child nodes. For a tree with a node size of B, a branching factor of B^{ε}, and a buffer size of $B - B^{\varepsilon}$, the amount of new data written during each node flush is at least $\frac{B-B^{\varepsilon}}{B^{\varepsilon}} = B^{1-\varepsilon}$. We call this the *batching factor*. Batching is why inserts and upserts in a B$^{\varepsilon}$-tree are $B^{1-\varepsilon}$ times faster than in a B-tree.

Derange queries bring the benefits of batching to queries. A derange query spanning a range of ℓ items touches $O(\frac{\log_B N}{\varepsilon} + \frac{\ell}{B})$ nodes during its execution. Derange query messages are flushed along with other messages in batches of size at least $B^{1-\varepsilon}$. Hence the amortized I/O cost of a derange query spanning ℓ items is $O(\frac{\log_B N}{\varepsilon B^{1-\varepsilon}} + \frac{\ell}{B^{2-\varepsilon}})$. In contrast, a normal range query spanning ℓ items requires $O(\frac{\log_B N}{\varepsilon} + \frac{\ell}{B})$ I/Os. The batching factor *divides* the cost; as a result, derange queries have the potential to provide as much speedup for queries as write optimization provides for inserts.

5 Derange Query Opportunities

In this section, we discuss the types of environments where derange queries would be particularly useful.

Mixed workload environments. A typical web-scale database serves at least two kinds of queries: small random queries that must be answered quickly, and large analytic queries that might take several hours in the best case but can be delayed by many more hours without hurting their value to the business. An example might be a credit-card database where customer purchases create many high-priority inserts, and large queries are performed overnight to find new fraud patterns.

If most of the I/O's needed by the big query can be piggybacked onto the small queries, then both types of queries can be performed without increasing the cost of the database or slowing down the small queries.

Point-in-time computations. In the common case, instances of the same derange query, repeated at multiple points in time, would be satisfied by reading each version of the data exactly once. Thus, derange queries can be used to increase the granularity of reporting.

Queries on overlapping ranges. Derange queries can make it easy to batch otherwise unrelated queries. For example, consider a system that performs one summary computation every 24 hours, and another summary computation ever 12 hours. Manually batching these computations would essentially require writing two versions of the 12-hour computation—one that runs on its own and another that runs as part of the 24-hour computation. With derange queries, developers need to write only one version of each computation, and the system will batch them automatically when possible.

6 Related Work

Amvrosiadis et al. observed that common file system maintenance tasks (e.g. backup, defragmentation, virus scanning, etc.) are frequently executed independently despite their largely overlapping working sets. The Duet [3] framework places hooks in the page cache to notify processes when requested data is available. This lets background tasks leverage the I/O performed by foreground work. Derange queries similarly leverage the internal work done by the B$^{\varepsilon}$-tree when it flushes messages to apply updates, piggybacking on I/O.

In the MapReduce [8] programming model, users filter and sort input data, independently process the filtered data, and combine the computations' outputs into a final result. MapReduce makes these types of operations easy to program for distributed data sets. Derange queries provide a similar programming model, but can optionally defer execution. The motivating use cases of this paper have been single-node, high performance, production databases, but derange queries could also be extended to work on a distributed storage system.

LINQ [1] features deferred execution, which delays the evaluation of an expression until its value is required. However, from the time an expression tree is created to the time the query is executed, the database may change. A derange query defers execution until the message is applied, but the message is always applied to the value of the data at the time the message is inserted.

7 Future Work

Even when derange queries cannot be delayed arbitrarily, they can provide significant speedups. Part of our future work is to analyze and empirically evaluate the performance opportunities created by derange queries.

When executing a derange query with a fixed deadline, the ability to systematically execute portions of the query would be useful. Otherwise, a burst of deferred work might need to be scheduled at the query deadline, eroding the benefits of batching. Derange queries create opportunities for I/O scheduling and workload management.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Cindy Rubio Gonzalez, for their insightful comments on earlier drafts of the work. This research was supported in part by NSF grants CNS-1409238, CNS-1408782, CNS-1408695, CNS-1405641, CNS-1149229, CNS-1161541, CNS-1228839, IIS-1247750, CCF-1314547, CNS-1526707, Sandia National Laboratories, and VMware.

References

[1] LINQ and deferred execution. `https://blogs.msdn.microsoft.com/charlie/2007/12/10/linq-and-deferred-execution/`, 2007. Viewed March 10, 2016.

[2] How to perform summary analytics on production databases? `http://goo.gl/lQXW7V`, 2016. Viewed March 8, 2016.

[3] G. Amvrosiadis, A. D. Brown, and A. Goel. Opportunistic storage maintenance. In *SOSP*, pages 457–473, 2015.

[4] Apache. HBase. `http://hbase.apache.org`, Last Accessed May 16, 2015, 2015.

[5] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In *SPAA*, pages 81–92, 2007.

[6] M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan, and Y. Zhan. An introduction to B^e-trees and write-optimization. *;login: Magazine*, 40(5):22–28, Oct 2015.

[7] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *SODA*, pages 546–554, 2003.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[9] J. Esmet, M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. The TokuFS streaming file system. In *HotStorage*, page 14, 2012.

[10] Google, Inc. LevelDB: A fast and lightweight key/value database library by Google. `http://github.com/leveldb/`, Last Accessed May 16, 2015, 2015.

[11] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. BetrFS: A right-optimized write-optimized file system. In *FAST*, pages 301–315, 2015.

[12] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. BetrFS: Write-optimization in a kernel file system. *ACM TOS*, 11(4), Nov. 2015.

[13] A. Lakshman and P. Malik. Cassandra - a decentralized structured storage system. *OS Rev.*, 44(2):35–40, 2010.

[14] MongoDB. *The MongoDB 2.6 Manual*, 2014. `http://docs.mongodb.org/manual/`, Viewed May 27, 2014.

[15] P. O'Neil, E. Cheng, D. Gawlic, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[16] K. Ren and G. A. Gibson. TABLEFS: Enhancing metadata efficiency in the local file system. In *USENIX ATC*, pages 145–156, 2013.

[17] RocksDB. `rocksdb.org`, 2014. Viewed April 19, 2014.

[18] R. Sears and R. Ramakrishnan. bLSM: a general purpose log structured merge tree. In *SIGMOD*, pages 217–228, 2012.

[19] P. Shetty, R. P. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok. Building workload-independent storage with VT-trees. In *FAST*, pages 17–30, 2013.

[20] Tokutek, Inc. TokuDB v6.5 for MySQL and MariaDB. `http://www.tokutek.com/products/tokudb-for-mysql/`, 2013. See `https://web.archive.org/web/20121011120047/http://www.tokutek.com/products/tokudb-for-mysql/`.

[21] X. Wu, Y. Xu, Z. Shao, and S. Jiang. LSM-trie: An LSM-tree-based ultra-large key-value store for small data items. In *USENIX ATC*, pages 71–82, 2015.

[22] J. Yuan, Y. Zhan, W. Jannen, P. Pandey, A. Akshintala, K. Chandnani, P. Deo, Z. Kasheff, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter. Optimizing every operation in a write-optimized file system. In *FAST*, pages 1–14, 2016.