Efficient Updates for OLAP Range Queries on Flash Memory

MITZI MCCARTHY AND ZHEN HE

Department of Computer Science and Computer Engineering, La Trobe University, VIC 3086, Australia
Email: m.mccarthy@latrobe.edu.au; z.he@latrobe.edu.au

This paper explores efficient ways to use flash memory to store OLAP data. The particular type of queries considered are range queries using the aggregate functions SUM, COUNT and AVG. The asymmetric cost of reads and writes for flash memory gives higher importance to how updates are handled in a flash memory environment. A popular data structure used for answering OLAP range-sum queries is the prefix sum cube. It allows the range-sum query to be answered in constant time. However, updating the prefix sum cube is very expensive. To overcome this the Δ-tree was proposed by Chun et al. [1]. The Δ-tree stores all updates to the prefix sum cube in a separate r-tree. This approach worked well for the hard disk where in-place updates are relatively cheap. However, for flash memory where in-place updates are very expensive the Δ-tree performs very poorly. We take a four pronged approach to overcome the problem of expensive in-place updates. The first is efficient caching of updates in RAM. The second is writing out *whole* trees from RAM to flash memory instead of incrementally updating a disk resident tree. The third is we allow users to trade bounded amounts of accuracy for less updates via lossy compression. Finally, we use a quadtree index structure instead of the R-tree. We prove the quadtree compression problem is NP complete. A greedy heuristic is proposed to find near optimal solutions in polynomial time. Various experiments were conducted to compare the proposed algorithms against the existing Δ-tree. The results show that our algorithms consistently outperformed Δ-tree by factors of between 10 and 100. This demonstrates the importance of designing flash memory customised algorithms for OLAP range queries. In addition, among our algorithms, the error bound solutions with a small error bound setting significantly outperform the accurate solution in terms of performance for a variety of parameter settings. This indicates the error bound algorithms offer users an effective trade off between execution time and accuracy.

Keywords: Relational Databases; OLAP; range queries; flash memory

1. INTRODUCTION

Decision support systems are used in many different industries to help inform the users of useful information for making business decisions. Online Analytical Processing (OLAP) applications and data warehouses are important parts of decision support systems. OLAP applications allow users to analyse multi-dimensional aggregates of data from data warehouses. The data is often stored in the form of multi-dimensional data cubes. To build a data cube, a certain subset of the attributes is selected. Some of the selected attributes are called measure attributes which are the metrics of interest, e.g. price, revenue, etc. The remainder of the selected attributes are called the functional attributes. The records which have the same functional attribute values are aggregated over the measure attributes. Aggregated data are stored as a data cube. For example, assume a data cube on salary with the following functional attributes: age, gender and job. Age is between 1 and 100, gender is male or female and job is {teacher, manager, clerk, none}. The data cube will have $100 \times 2 \times 4$ cells, with each cell containing the total salary for the corresponding combination of age, gender and job. In this paper, we considered range queries over these data cubes that use any of the following as the aggregation function: SUM, COUNT and AVG.

Summary structures have been proposed by many researchers [1, 2, 3, 4, 5, 6, 7, 8] for answering OLAP queries. These structures can greatly reduce the read cost of queries. However, due to the large amount of pre-computation needed to create these data structures, they are very expensive to keep up-to-date. However, keeping data warehouses up-to-date, is an important

Device	Seq. Read (ms)	Random Read (ms)	Seq. Write (ms)	Random Write (ms)
Memoright MR25.2-032S	0.3	0.4	0.3	5
Mtron SATA7035-016	0.4	0.5	0.4	9
Samsung MCBQE32G5MPP	0.5	0.5	0.6	18

TABLE 1. Time costs for NAND Flash

problem which has been extensively studied in the past [1, 9, 10, 11, 12, 13, 14, 15]. An example of a need for efficient data warehouse updates is tracking current sales data which requires frequent updates in an increasingly competitive environment. This highlights the importance of efficiently handling updates to data warehouses and OLAP data structures. However, these papers assume hard disk drives are the storage medium instead of flash memory.

Flash memory in the form of solid state drives (SSD) is becoming more prevalent. SSDs are secondary storage devices that have the same physical size as hard disk drives (HDD) but are made from flash memory and therefore have the same performance characteristics as flash memory. SSDs are designed to replace HDDs in the same sized slots. However, flash memory can be found in smaller sizes such as compact flash and USB sticks. Flash memory is very fast non-volatile RAM which has much lower read latency than HDDs. Table 1 [16] shows the cost of 32 KB IO operations on some typical SSDs. As shown in the table random writes are much more expensive compared to random or sequential reads due to the SSD's need to erase entire blocks of data before writing. Therefore if we can tap into the SSD's fast read performance while at the same time minimise the amount of writes we can potentially answer OLAP range queries much faster by storing data on the SSD.

In this paper we study the problem of OLAP range SUM, COUNT and AVERAGE queries. Ho et al. [8] proposed the prefix sum cube which allows range SUM, COUNT and AVERAGE queries to be answered very fast. However, keeping the data structure up-to-date is very expensive in terms of both IO and CPU costs. This led Chun et al. [1] to propose the use of a modified R*-tree, called the Δ-tree, to store the changes separate from the prefix sum cube. This was found to be effective at reducing update costs for a hard disk based system. However, updating the Δ-tree involves a lot of random writes. For flash memory where random writes are much more expensive than random reads we can do better. Our idea is to trade more random reads to drastically reduce random writes. We cache random writes in an in-RAM quadtree which is written out sequentially to flash when the buffer is full. Using this approach we produce a number of small quadtrees (we call these Δ-quadtrees) instead of one large Δ-tree. The downside of this approach is that reading becomes more expensive since it involves searching all Δ-quadtrees in order to answer queries. However, we take two approaches to keep reading costs low.

First, we use a cost based approach to decide the optimal merging of Δ-quadtrees to keep the total number of Δ-quadtrees low in order to minimise the overall reading and writing costs. Second, we compress the Δ-quadtrees to produce smaller trees so that both reading and writing costs are lowered. We propose a lossy compression approach. In order to give users more control on the error resultant from the compression, we allow users to bound the maximum amount of error they are willing to tolerate. Our approach for compression is to remove nodes from the tree until a minimum compression ratio has been reached. The nodes which can be removed are limited to those for which storing an average update value instead can be guaranteed to be within the error bound set. The compression problem is proven to be NP complete and as such, an optimal solution to the problem has exponential complexity. Therefore, we propose a greedy heuristic approach to finding a near optimal solution in polynomial time. We call our approach Quadtree Based Storage for Updates of OLAP (QUO).

There has been much recent work on building databases for flash memory [17, 18, 19, 20, 21, 22, 23]. However, none of the existing work targets OLAP applications which store data in the form of data cubes. The structure of data cubes and the way they are queried and updated is fundamentally different from relational tables. Hence, the existing solutions for flash memory databases are not applicable to this research.

We have conducted extensive experiments to compare our proposed algorithms against the Δ-tree by Chun et al. [1]. The results show that our algorithms consistently outperformed Δ-tree by factors of 10 and 100. The experiments also show that among our algorithms, the error bound solutions outperform the accurate solution for a variety of parameter settings. This indicates that if the users are willing to compromise accuracy to within a small error bound, then much performance can be gained.

This paper makes the following key contributions:

- It proposes a quadtree based storage algorithm for efficient handling updates to OLAP summary data structures for flash memory. The algorithm prevents random writes to flash memory by caching random writes in a RAM-resident Δ-quadtree and then writing the entire Δ-quadtree sequentially to flash when RAM is full. In order to minimise the reading costs of a potentially large number of

flash resident Δ-quadtrees we propose cost-based strategies to merge the trees.

- It further reduces read and write IO costs by giving users the option of trading a small bounded amount of accuracy for improved performance by way of lossy compression of the Δ-quadtrees.
- It performs a thorough experimental study comparing our algorithms against the existing state-of-the-art Δ-tree algorithm by Chun et al. [1]. In addition, the experiments also shed light on how much performance can be gained when users are willing to trade a small amount of accuracy for efficiency.

The paper is organised as follows: Section 2 describes the unique characteristics of NAND flash memory; Section 3 surveys the body of related work; Section 4 describes our quadtree-based storage algorithms for handling updates in OLAP applications running on flash memory; Section 5 describes the experimental setup used to conduct our experiments; Section 6 provides the experimental results and analysis of our experiments; and finally Section 7 concludes the paper and provides directions for future work.

2. FLASH MEMORY

There are two different types of flash memory, NAND and NOR. The fundamental difference between the two is the layout of the memory cells. In NOR flash, each cell is located at the intersection of one word line and one bit line, allowing for random access, whereas NAND cells must be sequentially accessed within a block of cells. The main use of NOR flash is for execution-in-place applications. Storing data in blocks allows data on NAND flash to be more compactly stored than for NOR flash, which is an important advantage for portable devices. Storing the data in blocks also gives NAND faster sequential write and erase speeds over NOR. NOR flash is also more expensive and has a lower storage capacity than NAND flash. Another advantage of NAND over NOR is the higher number of writes and erases that can be made before the cells become unreliable. Given these advantages of NAND over NOR flash and its suitability to our application scenario, we will concentrate on NAND flash memory and its characteristics in this paper.

NAND flash is organised into blocks with each block containing multiple pages. The typical size of pages is 4 KB, with 64 pages contained within each block. A limitation of flash memory is that data cells can not be overwritten. To erase a cell on flash memory, the entire block must be erased. Due to the limited number of times cells can be erased before they are unusable, erasing an entire block for each update is undesirable. This means that when an update needs to be performed, new values are written to free areas of the memory and the locations of the old values are then considered dirty. Invalid areas of memory must be reclaimed later so that the space can be reused.

A flash translation layer is needed to map logical read and write operations to the physical read, write and erase operations. This involves maintaining a mapping from the logical to physical addresses and assigning free space for update operations to avoid the need to erase. Some research [24, 25, 26] has been done in finding effective ways to use the flash translation layer. Wu et al. [27] propose an additional layer on top of the existing flash translation layer to improve the performance of applications using index structures. Kim et al. [25] propose a flash translation layer which provides a buffer to store writes to minimise the need to perform erases. Lee et al. [26] extend the work by Kim et al. [25] to better utilise the space within the buffer.

Research by Bourganim et al. [16] found that the flash device is best considered as a black box when modeling its read and write performance. The reason is the flash translation layer plays a large part in determining the performance of the device and details of the flash translation layer for commercial devices are not published. Extensive experiments done by Bourganim et al. on many different flash devices reveal that sequential writes cost much less than random writes. This fact is shown in Table 1. Our system is designed to take advantage of the fact that sequential writes are much cheaper than random writes by converting random writes to sequential writes via clever caching and writing a series of small Δ-quadtrees to flash rather than continuously updating a single large Δ-quadtree. **In should be noted that although the figures in Table 1 are typical for a wide range of SSDs, some high-end SSDs exist which provide similar performance for random reads and random writes.**

Other research [27, 28, 29] has been done on how to implement a garbage collector to reclaim the space lost by updating out of place. These garbage collectors aim to reduce the number of erase operations needed and to evenly distribute the wear associated with erasing blocks. Wu et al. [27] use a greedy method for garbage collection which chooses sections of data to reclaim, based on the amount of invalid data. Chiang et al. [28] proposed a method to group data together which has similar write frequencies during garbage collection. Chiang et al. [29] propose a method for garbage collection which takes three things into account: the cost to clean the data, the age of the data and the number of erases that have been performed on that block.

3. RELATED WORK

In this section we survey the literature in two areas which are the most similar to our work. The first is efficient range query processing for OLAP data cubes and the second is log based approaches to index updates.

3.1. Efficient range query processing for OLAP data cubes

In this section we describe the existing work on efficient computation of queries in OLAP applications. None of the existing pieces of work in this area use flash memory as the storage medium, they all use the hard disk instead. One category of existing work focuses on selecting which queries to pre-compute. Deciding which and how many queries to pre-compute in order to get the most benefit has been explored by many researchers [2, 3, 4, 5, 6]. Another category of existing work concentrates on how best to pre-compute queries. Reducing the cost of computing and materalising high dimensional data cubes with large data sets by pre-computing aggregates has been explored by many researchers [3, 4, 7]. Some existing work [1, 4, 14, 15] focuses on providing estimated answers to queries. Much existing work [9, 10, 11, 12, 13] explores how to efficiently update materialised views.

In contrast to the above existing work, we are interested in data structures and algorithms that focus on efficient range query computation for a given set of dimensions rather than deciding which dimensions to pre-compute. **Substantial theoretical research exists on the development of efficient data structures for computing range SUM queries [8, 30, 31, 32]. Similarly, research exists for range MIN and MAX queries [8, 32, 33, 34, 35]. These papers are focused on reducing the amount of data access when executing queries and updates in a system independent way. In contrast, our paper is focused on reducing flash memory I/O update costs by making best use of a RAM buffer.**

We have chosen to build our algorithms on the prefix sum cube data structure proposed by Ho et al. [8] because of its simplicity and popularity. Ho et al. [8] propose two data structures for increasing the efficiency of range queries for OLAP data cubes. The first is the prefix sum cube which is used for SUM aggregation queries over ranges of dimensional values of data cubes. The second is a tree structure used for MAX aggregation queries. The space needed for storage of these data structures is also considered. Updates to the data structures are done as batch updates offline.

In considering range queries for SUM and MAX, the following five main SQL aggregation functions are covered: SUM, AVG, COUNT, MIN and MAX. A tree-based algorithm is proposed for the aggregation function MAX and similarly for MIN. The tree structure is constructed by having a root node which contains the maximum value of all cells of the data cube. Subsequent levels of the tree are constructed by dividing each parent node into disjoint subsets of similar size, each of these child nodes containing the maximum value of their subset of the data cube. Queries can then be answered by searching this tree for the related sections of the data cube.

The data structure for answering SUM range queries is called the prefix sum cube. This cube is the same size as the original data cube. It is constructed by each cell containing the sum of its value and all values with a lower index in all of the dimensions. An example of a two dimensional data cube and its corresponding prefix sum cube can be seen in Figure 1. Note that the prefix sum cube can be constructed for any $n \in \mathbb{N}^+$ dimensions.

2D Data Cube						
Index	0	1	2	3	4	5
0	3	5	1	2	2	3
1	7	3	2	6	8	2
2	2	4	2	3	3	5

(a) Data Cube

2D Prefix Sum Cube						
Index	0	1	2	3	4	5
0	3	8	9	11	13	16
1	10	18	21	29	39	44
2	12	24	29	40	53	63

(b) Prefix Sum Cube

FIGURE 1. Example 2D data cube and associated prefix sum cube

Queries are computed from the prefix sum cube by retrieving the value from the prefix sum cube of the highest index in each dimension. From this value, the indexes which should not have been included in the sum are subtracted. Any cells whose values were subtracted more than once are added back on. For example, a query on the data cube of Figure 1 could be to sum the values of the original data cube A from column 3 to 5 and row 1 to 2. This would be calculated using the prefix sum cube P as follows:
$A[1:2,3:5] = P[2,5] - P[2,2] - P[0,5] + P[0,2] = 63 - 29 - 16 + 9 = 27$.

The storage needs can be minimised by deleting the original data cube once the prefix sum cube has been computed. The problem with this is the difficulty it creates in answering queries which involve the MAX and MIN aggregation functions. The other way discussed is to keep the original data cube and construct a blocked prefix sum cube where the prefix sum is only stored for a fraction of the cells. This saves storage space but means that queries are less time efficient.

Chun et al. [1] proposed a new algorithm and data structure to improve the efficiency of updates for range-sum queries. The aim of the algorithm is to find an efficient method to reflect updates to the data cube and the prefix sum cube proposed by Ho et al. [8].

Chun et al. [1] improved the work by Ho et al. [8]

by reducing the update time without jeopardising the search performance. To this end, they proposed the Δ-tree. This tree structure contains the change in values which have been updated from the original data cube. The idea behind having a separate structure is that the prefix sum cube is very costly to update, more so than the original data cube. The Δ-tree is used together with the prefix sum cube to answer queries with updated values without the need to update the prefix sum cube. The performance results in Chun et al. [1] show that updates to the Δ-tree are much more efficient than to the prefix sum cube. The number of pages accessed to perform updates is $O(\log(N_u))$ where N_u is the number of changed cells. This compares well to the prefix sum cube where updates have time complexity $O(n^d)$ where n is the number of cells in each dimension and d is the number of dimensions. The drawback of the Δ-tree approach for flash memory is that insertions and deletions to the Δ-tree involve splitting and updating tree nodes which are very costly to perform in flash memory because it causes many random page writes. In contrast, our approach converts the many random flash page writes into in-cache updates followed by sequential writes when the cache is full. The results show our algorithms greatly outperform the Δ-tree approach.

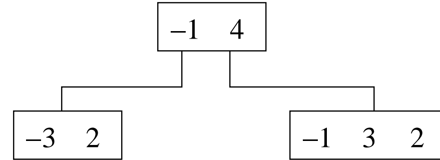
An example set of updates to the data cube shown in Figure 1(a) can be seen in Figure 2(a) and the corresponding Δ-tree in Figure 2(b). In Figure 2(b), the leaf nodes of the Δ-tree contain the individual updates from the update cube. The root node of the Δ-tree stores the sum of the updates for each of its child nodes (i.e. the leaf nodes). The same query as before would now be calculated using the prefix sum cube and the Δ-tree as follows:

$$\begin{aligned}\Sigma(A[1:2,3:5]) &= P[2,5]-P[2,2]-P[0,5]+P[0,2]\\ &+ \Sigma(\Delta\text{-}[1:2,3:5])\\ &= (63-29-16+9)+(3+2)=32\end{aligned}$$

where $\Sigma(\Delta\text{-}[1:2,3:5])$ is the sum of values inside the Δ-tree which cover the following range: columns 3 to 5 and rows 1 to 2.

2D Dynamic Update Cube						
Index	0	1	2	3	4	5
0	−3				−1	
1		2				3
2					2	

(a) Update Cube



(b) Δ-tree

FIGURE 2. Example of data cube updates and associated Δ-tree

3.2. Log based approaches to index updates

There has been widespread work in using a log based approach to handle frequent updates to primary index structures [21, 36, 37, 38, 39, 40]. The first of these is the Log-structured Merge (LSM) tree by O'Neil et al. [36]. The LSM-tree uses a primary index structure in memory as well as one or more index structures on disk to reduce the need to write to disk structures. The memory and disk structures are merged once enough updates or inserts have occurred. The LSM-tree, like other log based index structures [21, 37, 38, 39], is designed for variants of the B-tree.

Although the LSM-tree appears to be using a similar approach to our work, there are a number of key differences. Firstly, the LSM-tree is designed for an environment where updates occur much more often than queries. This is not the norm for OLAP. Secondly, the LSM-tree is designed for hard drives rather than SSDs therefore it considers random reads to be much more expensive than sequential reads. Thirdly, our merge is fundamentally different from the LSM-tree because of the type of data stored. The LSM-tree merges trees of discrete rows of data which cannot be combined whereas we can combine the data stored in our trees because it is aggregation data. Lastly, we support lossy compression which would not be suitable for the LSM-tree given that it stores transactional rows of data.

There are several other works which are similar [21, 37, 40] to the LSM-tree or build on [38, 39] the LSM-tree. The work presented by Jagadish et al. [37] is similar to the LSM-tree (although it was developed independently). It differs from the LSM-tree in that it aims to reduce the number of I/O operations rather than LSM-tree's aim of reducing the dollar cost for supporting a given load by determining optimal amount of memory and disk to use. The work by Jagadish et al. [37] also differs from the LSM-tree by making optimised concurrency control schemes as one area of focus. The work in [38, 39] addresses the LSM-tree assumption that there is a steady rate of arrival of incoming data. The work by Li et al. [21] is similar to the both LSM-trees and the work by Jagadish et al. [37]. The difference being that Li et al. [21] is designed for flash memory. Although this is similar to our aim of using flash memory as the storage medium, the other differences listed between our work and the LSM-tree still hold with regard to the work by Li et al. [21]. The work in [40] is also similar to [36] and [37]. It differs in that it stores updates in a quadtree since it is designed for a dynamic spatial database. Although [40] uses a quadtree to store updates as our work does, it still has a number of differences: lossy compression is not

appropriate; no merge of aggregation data; designed for hard disk drives; target scenario has frequent updates.

4. QUADTREE-BASED STORAGE FOR UPDATES OF OLAP (QUO)

This section gives the details of the methodology proposed in this paper for storing and querying an OLAP cube in flash memory in the presence of updates.

A high level view of the QUO system is given in Figure 3. The system consists of a RAM-based read buffer, a RAM-resident Δ-quadtree (Δ-RAM), a flash-resident prefix sum cube and a sequence of flash-resident Δ-quadtrees (Δ-Flashes). The idea is to satisfy as much of the queries and updates as possible within RAM and go to flash only if necessary. The read buffer is used to cache read requests to both the prefix sum cube and the Δ-Flashes. The Δ-RAM is used to cache updates by incrementally building a RAM-resident Δ-quadtree until its RAM allocation has been exhausted. Once Δ-RAM is full, it is either flushed directly to flash memory or merged with an existing Δ-Flash or discarded with a new prefix sum cube being built. We use a cost model to decide which option to take. An intermediate step of compressing Δ-RAM is carried out if the user can tolerate an error bound (and Δ-RAM is not being discarded).

Our system dynamically adjusts the ratio of RAM allocated to the read buffer versus the Δ-RAM so that when RAM sizes are large and updates are infrequent more RAM can be allocated for the read buffer instead of Δ-RAM. This is done using the following simple yet effective mechanism. Allow the read buffer to grow beyond its pre-allocated size if the Δ-RAM has not fully utilised its pre-allocated size. If the read buffer has encroached on the Δ-RAM pre-allocated RAM space and the Δ-RAM needs to grow, then allow the Δ-RAM to grow up to its pre-allocated size by evicting from the read buffer.

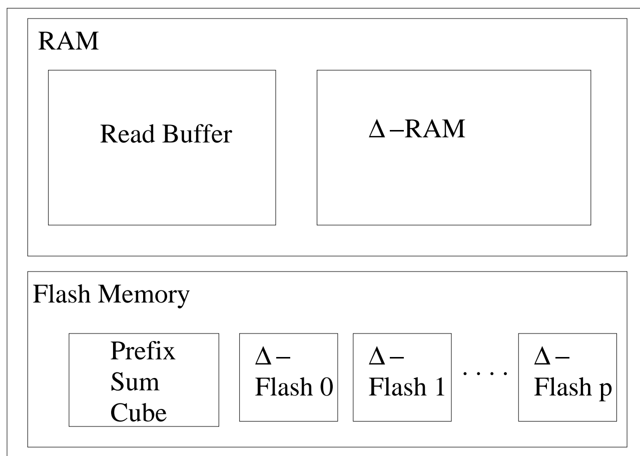


FIGURE 3. QUO System Diagram

QUO can be used to obtain both precise and imprecise answers to queries. Imprecise query answers allow the user to trade precision for query response time since it reduces the sizes of the Δ-Flashes and therefore less reads are required to answer a query. These answers are within some set bound for the maximum absolute error from the precise answer. The error is introduced by compressing each Δ-RAM before it is moved to flash. Each compression is done with a minimum compression ratio but with a maximum amount of error that can be introduced. These two conditions allow for more query answering while giving users confidence by guaranteeing that answers will be within the set error bound. The precise option is useful for users who are not willing to compromise accuracy for efficiency.

Throughout this section, the focus is on the process of compressing Δ-RAM, storing data updates and selecting the best option for storing Δ-RAM on flash. These are the components which are designed to accommodate the characteristics of flash memory.

4.1. Δ-Quadtree Structure

The Δ-quadtree is a tree structure which is used to store the updates that occur in the prefix sum cube. A tree structure is used for similar reasons as in Chun et al. [1]. An advantage of using the quadtree over the R*-tree used by Chun et al. [1] is that the quadtree can divide the space more efficiently since it assumes a fixed range of values. We assume a fixed range of values because typically, once the ranges of each dimension are set in the prefix sum cube, this remains constant. Another advantage of the quadtree over the R*-tree is that quadtrees can be merged more efficiently since many nodes of the trees may cover the same area, whereas R*-trees have no set of ranges for nodes. The problem of trying to minimise node overlap for R*-trees does not exist in quadtrees, as ranges between siblings never overlap.

Each node in the Δ-quadtree has either no children or 2^d children where d is the number of dimensions in the data. The individual updates (Δs) are stored on the leaf nodes. Each node in the tree contains the sum of the Δs for the area of the prefix sum cube that it covers. The Δ-quadtrees have a maximum size for each node as well as a maximum depth.

Figure 4 shows an example 2D Δ-quadtree and its associated updates in the prefix sum cube. In this section, we assume the aggregation function is SUM, however the technique will work equally well for COUNT and AVG since COUNT is simply SUM with each value equal to one and AVG is SUM divided by COUNT.

4.2. Insertion Into Δ-Quadtree

The Δ-quadtree initially has one node which covers all cells of the prefix sum cube. As updates occur, they

UPDATES TO PREFIX SUM CUBE:

		−1		6	−3		
2							
10	2	8	1				
−4	7	−3	−6				7

Δ −QUADTREE:

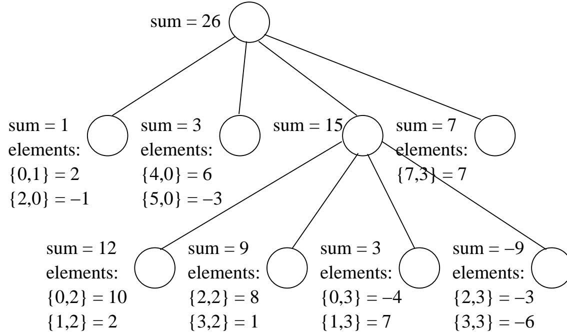


FIGURE 4. Δ-quadtree example

are added to the node of the tree whose range of cells includes the cell of the update. When data is inserted into a node that has reached its maximum size, the node is allocated 2^d children. The ranges of cells covered by this new parent is split into two even sections in each dimension and assigned to the children. Once the children's ranges are set, all the elements of the parent are assigned to the relevant child. Each update can have either a positive or negative value and is amalgamated with any existing update in the tree which is for the same cell. For each update, the sum of Δs is updated in every ancestor of the leaf node that the element belongs to, as well as the leaf node.

Figure 4 shows the updates to the prefix sum cube and corresponding updates to the Δ-quadtree. The root node of the tree covers the whole prefix sum cube. The cube is divided into four quarters by splitting each dimension down the middle. A child of the root node is created for each of these quarters. The bottom left quarter of the cube is further drilled down to create four child nodes in the Δ-quadtree because the maximum size of a node had been reached. Figure 4 also shows the sum of Δs for each node, as well as the elements for each node. The element {4,0} = 6 means that the cell with index 4 in the horizontal dimension and with index 0 in the vertical dimension has an update with a positive Δ of size 6.

Figure 5 shows the algorithm for inserting these updates. In lines 4 - 16 of the algorithm, we navigate down to the leaf node into which the update record maps. In lines 17 - 23, we look for an existing update in the leaf node for the same coordinates as the current update record. If an existing update for the coordinates is found, the sum of the existing and current updates is stored as the new Δ. In lines 24 - 26, if a matching existing update is not found, the current update record is inserted into the leaf node. Finally, in lines 27 - 30, if the leaf node is now greater than the maximum allowable size, 2^d children are created for the node.

```
Insert_Update_Record(u: update record to insert,
                     T: Δ-quadtree,
                     m: the max size for a quadtree node,
                     d: number of dimensions in data cube)
1. Let C be the current node in the traversal
2. Let CC be the set of children of C
3. Initialise C to be the root node of T
4. while ( leaf node not found )
5.    CC = children of C
6.    if ( CC is empty )
7.      exit while
8.    else
9.      For each c ∈ CC do
10.        Let R store the ranges of c
11.        if ( u ∈ R )
12.          C = c
13.        end if
14.     end for
15.   end if
16. end while
17. while ( previous update in C with same coordinates
           not found )
18.   Let p be the current update record in C
19.   if ( coordinates of p = coordinates of u )
20.     Δ of p = Δ of p + Δ of u
21.     exit while
22.   end if
23. end while
24. if ( no match in C for u )
25    Insert u into C
26. end if
27. if ( size of C > m )
28.   Let CN be 2^d new children of C
29.   Move the elements of C to the appropriate cn ∈ CN
30. end if
```

FIGURE 5. Insertion of an Element into a Δ-quadtree

4.3. Range Queries For Δ-Quadtree

When answering a range query on a Δ-quadtree, the tree is traversed starting from the root node. When traversing each node, the ranges that the node covers are compared to the ranges of the query. If the cells the node covers are a subset (not necessarily a strict subset) of the cells of the query, the sum of the Δs for the node is added to the query answer. Otherwise, the children of the node whose ranges intersect the query ranges are traversed. If the node is a leaf node, each element needs to be checked to see if its Δ should be added to the answer. Once the relevant nodes have been traversed, the sum for the tree is added to the sum for all the other trees, as well as the sum for the prefix sum cube. This gives the final answer to the query.

Figure 6 gives an example range query for a Δ-quadtree. The range query in the example is $\{2-3, 0-2\}$ which means the sum of the values whose indexes are between 2 and 3 (inclusive) in dimension 1 and between 0 and 2 (inclusive) in dimension 2. The range query is calculated as the sum of the element $\{2, 0\} = -1$ plus the sum for node G. So, the answer for the range query of the Δ-quadtree would be $-1 + 9 = 8$. The entire sum of node G is added since the ranges of node G are within the ranges of the query.

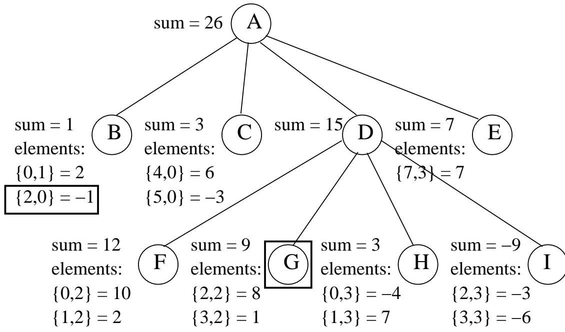


FIGURE 6. Δ-quadtree range query example

Figure 7 shows the range query algorithm. In lines 6 - 17 of the algorithm, we iterate through the set of nodes to traverse. Initially, the set only contains the root node. In lines 10 - 11, for each node we traverse, if the node's ranges are within the query ranges then the sum of the Δs of the node is added to the sum. In lines 12 - 13, if the node is a leaf node, the Δs of the node which reside inside R are added to the sum. Otherwise, in line 15, the children of the node which overlap the query are added to the set to traverse. Once the nodes have been traversed, the sum is returned.

Range_Query_Quadtree(R: ranges of read query,
T: Δ-quadtree)
1. Let C be the current node in the traversal
2. Let NT be the set of nodes to traverse
3. Let s be the sum
4. Initialise C to be the root node of T
5. Add C to NT
6. Initialise s to 0
7. while (NT is not empty)
8. Let CR be the ranges of C
9. Let CD be the sum of Δs of C
10. if ($CR \subseteq R$)
11. add CD to s
12. else if (C is a leaf node)
13. add Δs of elements of C which are inside R to s
14. else
15. add children of C which overlap with R into NT
16. end if
17. remove C from NT
18. end while
19. return s

FIGURE 7. Range Query for a Δ-quadtree

4.4. Δ-Quadtree Compression

If the error bound solution is being used, Δ-RAM is compressed before it is flushed to flash or merged with a Δ-Flash. The compression algorithm guarantees that the accumulated error from all previous compressions and the current compression for each cell does not exceed the error bound. Under this constraint, the compression algorithm also ensures the minimum compression ratio is reached at the end of each compression. Compression is performed by removing nodes from Δ-RAM until the minimum compression ratio has been reached. The minimum compression ratio is used to ensure that the tree is compressed to a sufficiently small size. When a node is removed, the average value of the elements covered by the node is stored instead. A node can only be compressed if the error bound can be guaranteed. We do this by ensuring that for each cell of the prefix sum cube (PSC), the difference between the average value which would be used for answering queries and the actual value of the update is within the absolute error bound. The amount of the error bound which has been used by compression of all previous Δ-quadtrees is remembered to ensure the error bound is preserved.

The Δ-quadtree compression problem is NP complete. Before this is proven, we give a formal definition of the problem. Property 1 below defines the property that for each cell, the accumulated error must be less than the error bound.

Property 1. After every compression $\Sigma_{i=1}^{p} error(c, i) \leq EB$ for every $c \in C$, where C is the set of all cells, p is the number of previous compressions and $error(c, i)$ is the amount of error introduced on cell c by the ith previous compression. Where error is defined as the absolute difference between the sum computed from non-compressed Δ-quadtree versus the compressed Δ-quadtree.

Below is a description of the problem of deciding how to best compress the Δ-RAM as well as two heuristic solutions and a proof that the problem is NP-complete.

Let EB be the total error bound, $sizeb(a)$ be the size in bytes for storing node a, $MEL(a)$ be the smallest error left for any cell covered by node a (this is equal to the minimum $EB - \Sigma_{i=1}^{p} error(c, i)$ where c is a cell of a that gives the minimum value), $NC(a)$ be the number of prefix sum cube cells covered by a, $CC(a)$ be the area covered by the prefix sum cells covered by a, ML be the memory limit for the Δ-quadtree in RAM, $TSD(S)$ be the total size after deleting the nodes contained in S and c be the minimum compression

ratio defined as the total size of the quadtree before compression divided by the total size after compression.

Δ-quadtree compression problem
Given a set A of possible nodes to delete, find $S \subseteq A$ such that the error bound consumed $EBC(S) = \Sigma_{s \in S}(EB - MEL(s))NC(s)$ is minimised and the following constraints are satisfied:

$(\forall a, b \in S)(CC(a) \cap CC(b)) = \emptyset$, $\frac{ML}{TSD(S)} \geq c$ and $MEL(s) \geq 0$ for each $s \in S$

If the constraints are met, then the set S is returned, otherwise the null set is returned, where returning the null set means the compression ratio cannot be achieved by selecting any subset of nodes of A.

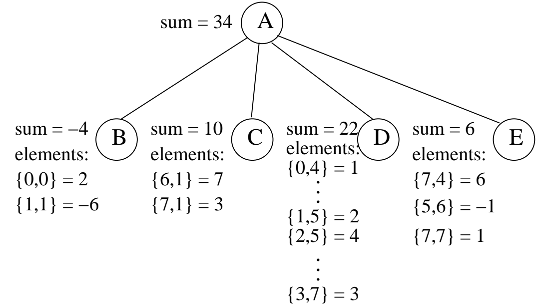
The above metric aims to minimise the total error bound which is consumed by all the nodes being deleted from the tree while ensuring the tree has been compressed enough so that the compression results in a significant efficiency gain. The set of nodes to delete does not contain any ancestors or descendants of each other. For each node to be deleted, the MEL must be greater than or equal to 0 after the compression (the third constraint). This ensures that Property 1 is met because if the minimum error left for a node is greater than 0, then the error for each cell of that node must be less than the error bound.

Figure 8 shows an example prefix sum cube and a Δ-RAM before and after compression. In the figure, node D is selected to be compressed since compressing only node D gives the solution with the minimum amount of error bound consumed while meeting the compression ratio constraint. In the example, an error bound of 3 is used. Once node D has been compressed, the new MEL is 0.375. Node D can be compressed because for each cell covered by the node, the total error stays within the bound.

UPDATES TO THE PREFIX SUM CUBE:

2							
	–6					7	3
1	1	1	1				6
1	2	4	1				
1	1	1	1		–1		
1	1	1	3				1

Δ –RAM:



Δ –RAM COMPRESSED:

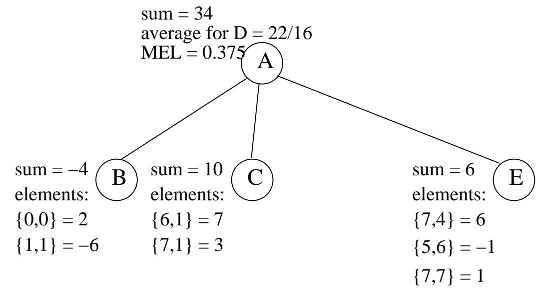


FIGURE 8. Δ-Quadtree compression example

THEOREM 4.1. *The Δ-quadtree compression problem is NP complete.*

To prove Theorem 4.1, we restrict the quadtree compression problem to only deleting leaf nodes and the domain of integers instead of the real numbers. The restricted problem is mapped to an equivalent node retention problem. The retention problem is mapped to the knapsack problem. Knapsack is known to be NP complete and therefore, so is the quadtree compression problem. A formal proof can be found in the Appendix.

The fact Δ-quadtree compression is NP complete means finding an optimal solution has exponential time complexity. Therefore, in the next subsection, we propose a greedy heuristic solution to find a near optimal solution in polynomial time.

4.4.1. Greedy Heuristic
The greedy heuristic works by ranking all the nodes in the tree from lowest to highest by the error bound consumed per byte freed. The first node in the ranking is chosen to be deleted. These two steps are repeated until the minimum compression ratio has been reached. Note that whenever a non-leaf node is chosen to be deleted, all descendants of the node contained in either the set of nodes to keep or delete are removed from the relevant set. The advantage of this heuristic is that the nodes deleted are those that give the best compression amount while introducing relatively small error. The run time complexity of this heuristic is $O(n^2)$, where n is the number of nodes in the Δ-RAM.

Let $EBCF(a) = \frac{(EB-MEL(a))NC(a)}{min(size(a), ML-\frac{ML}{c}-\Sigma_{a \in CS} size(a))}$ denote the error bound consumed by node a per byte freed. This metric is used by the greedy heuristic to incorporate the fact that deleting larger nodes

while incurring the minimum amount of error is more desirable.

Figure 9 shows the algorithm for compressing the tree using the greedy heuristic. In lines 1 - 6 of the algorithm, the sets to retain and delete are initialised. In lines 7 - 9, the nodes are selected to be deleted one at a time in sorted order until the compression ratio is met. The set of nodes to retain is then returned. Selecting the nodes to be deleted is done by passing the sets to retain and delete by reference to the Move_Node_To_Delete_Set function which is called on line 8. The Move_Node_To_Delete_Set function is given in Figure 10. The function deletes the node which minimises $EBCF$ and updates the sets to reflect this.

```
Greedy_Heuristic(T: RAM-resident Δ-quadtree)
1. Let RS be the set of nodes to retain
2. Let CS be the set of nodes chosen to be deleted
3. Let DRS(a) be the set of descendants of node a
   contained in the set RS
4. Let DCS(a) be the set of descendants of node a
   contained in the set CS
5. Initialise RS to be the set of all nodes in T
6. Initialise CS to be the empty set
7. Begin repeat
8.     call Move_Node_To_Delete_Set(RS,CS)
9. Repeat from 7 if Σ_{a∈RS} size(a) > ML/c
10. return RS
```

FIGURE 9. Algorithm for Greedy Heuristic

```
Move_Node_To_Delete_Set(RS: current set of nodes to
                        retain passed by reference,
                        CS: current set of nodes to
                        delete passed by reference)
1. Let m be the node a ∈ RS which minimises EBCF(a)
2. RS = RS − m − DRS(m)
3. CS = CS ∪ m − DCS(m)
```

FIGURE 10. Function for moving a node to delete set

4.5. Range Queries

When a range query occurs, the answer needs to be constructed by obtaining the answer to the query for each of the Δ-Flashes, the Δ-RAM and the prefix sum cube. Once we have these answers for each of the components, they are added together to give the final answer to the query. Each flash page which contains a prefix sum cube cell needed for the query is loaded into the read buffer. Similarly, each flash page which contains a node traversed for a Δ-Flash is also loaded into the buffer. The buffer is not needed for Δ-RAM as the RAM-resident quadtree remains in RAM until the whole tree is moved to flash.

4.6. Cost-Based Eviction For Δ-Quadtree

Whenever the Δ-quadtree stored in RAM (Δ-RAM) becomes full, a decision needs to be made about where to store the information it contains. This section describes how this decision is made. Note that the cost-based eviction is the same for both the accurate and error bound solutions; the only difference is the size of the Δ-quadtree evicted. The error bound solution evicts smaller Δ-quadtrees since it first compresses the Δ-quadtrees before evicting them.

The following lists the three choices that can be made upon eviction of Δ-RAM:
1. Merge Δ-RAM with the Δ-Flash (where each Δ-Flash can have a different merge count) that has the lowest expected cost per operation, where ties are broken based on some heuristic (eg. degree of overlap).
2. Flush Δ-RAM directly to flash.
3. Discard the Δ-RAM and Δ-Flashes and rebuild the prefix sum cube.

The option which is selected depends on which has the lowest expected cost per operation. The cost per operation of the first option is calculated based on the cost of writing the merged tree to flash and the cost of reading the merged tree as well as all the other Δ-Flashes. The cost per operation of the second option is calculated based on the cost of writing Δ-RAM to flash and the cost of reading all Δ-Flashes. The cost per operation of the third option is based on the cost of reading the underlying updated data cube and then producing the prefix sum cube and writing it to flash memory to replace the old prefix sum cube. The cost of reading the prefix sum cube and updating the underlying data cube is not included in the formulas for any of the options because it is the same cost for all of them.

Let $pr(write)$ be the probability of a write operation, $pr(read)$ be the probability of a read operation. These two probabilities are calculated using tallies of the number of reads and writes during the online workload. Let $CPW(t)$ = cost of writing Δ-quadtree t into flash memory, $CPR(t)$ = cost of reading Δ-quadtree t from flash memory, $NoWB(\Delta - RAM)$ = number of writes operations since the beginning of Δ-RAM, $CPRMC(c)$ = cost per read for merge count c, $CR(t)$ = cost of reading Δ-quadtree t, $CW(t)$ = cost of writing Δ-quadtree t, $CM(t1, t2)$ = cost of merging Δ-quadtree $t1$ with Δ-quadtree $t2$, $NoWPSCU$ = number of writes since prefix sum cube was last rebuilt, $MC(t)$ = merge count for Δ-quadtree t, $CRDC$ = cost of reading updated data cube and $CWPSC$ = cost of writing new prefix sum cube.

The expected cost of each of the options is described as follows:

1. Expected cost per operation for merging Δ-RAM and $\Delta\text{-Flash}_i$

$$\begin{aligned} &= pr(write) \times CPW(merged(\Delta - RAM, \Delta - Flash_i)) \\ &\quad + pr(read) \times CPR(merged(\Delta - RAM, \Delta - Flash_i)) \\ &= pr(write) \times \frac{CM(\Delta - RAM, \Delta - Flash_i)}{NoWB(\Delta - RAM)} \\ &\quad + pr(read) \times (CPRMC(MC(\Delta - Flash_i) + 1) \\ &\quad + CR(all(\Delta - Flash) - (\Delta - Flash_i))) \end{aligned}$$

2. Expected cost per operation of flushing Δ-RAM directly to flash

$$\begin{aligned} &= pr(write) \times \frac{CW(\Delta - RAM)}{NoWB(\Delta - RAM)} \\ &\quad + pr(read) \times (CPRMC(1) + CR(all(\Delta - Flash))) \end{aligned}$$

3. Expected cost per operation of rebuilding the prefix sum cube.

$$= pr(write) \times \frac{CWPSC + CRDC}{NoWPSCU}$$

Offline training is used to obtain the values of CM and $CPRMC$ for different merge counts. This is done by running the algorithm offline, using a representative workload and forcing all evicted Δ-quadtrees to be merged progressively into one large Δ-quadtree. This way we get read and write costs for Δ-quadtree that have been merged 0 to n times, where n is the maximum merge count. The size of each of the resultant trees is used to calculate CM. Read queries are performed on merged trees and $CPRMC$ is calculated by using the number of pages loaded into RAM to obtain answers to these queries. Once we have these two values for the different merge counts, the expected costs per operation can be calculated during the online algorithm for the online workload.

It should be noted that the offline training can be avoided. This can be done by mimicking the eviction decisions of the offline training at the beginning of the online algorithm. Doing this would mean slightly lower online performance during the online training phase. However, once the online training is finished, the performance would be the same as if offline training were used.

5. EXPERIMENTAL SETUP

This section gives the details of the various aspects of the experimental setup.

5.1. Simulation Setup

The experiments were carried out on a simulation of NAND flash memory written in C++. The parameters used in the simulation are given in Table 2. The read and write costs are the same as those taken from Table 1 for the Memoright SSD. These are the default parameters used in the experiments.

Parameter	Default Value
Sequential read cost per page	0.0375 ms
Random read cost per page	0.05 ms
Sequential write cost per page	0.0375 ms
Random write cost per page	0.625 ms
Flash page size	4KB
Number of pages per block [41]	64
Number of queries	100,000
Error bound	10
Maximum size for each quadtree	1KB
Maximum Depth of a quadtree	6
Updates percentage of total queries	50%
Size of prefix sum cube	5000KB
Size of RAM	100KB
Percentage of RAM used for Δ-RAM	75%
Minimum compression ratio for Δ-RAM	4
Number of dimensions	4

TABLE 2. Parameters Used For Simulation

A real data set was used in the experiments. This is a data set constructed from a subset of US census data for the year 2000 [42]. The dimensions used are age, salary, social security salary and travel time. The data is normalised based on the number of dimensions used for the experiment and the amount of space available for storing the prefix sum cube. Once the data is normalised, each person whose data is in the subset is assigned to the relevant data cube cell. The number of data points generated is 2^{20}.

A data-centred query set was used in the experiments. This is a query set generated by selecting the centre of the query based on the magnitude of the underlying data. The probability of any one cell being chosen as the centre equals the proportion of the data that belongs to that cell. The span of the query is generated using the Gaussian distribution in each dimension in both directions. The standard deviation of the span is a quarter of the maximum range.

A data-centred update set was used in the experiments. This is a set of updates with the cell to be updated generated in the same way as the data-centred query set. The update amount is generated based on the value in the data cube for the selected cell. This is done by using the Gaussian distribution with the mean as the current value and the standard deviation as one hundredth of the current amount.

5.2. Algorithm Setup

The experiments compared the results of three algorithms which are described as follows:

Accurate Merge Algorithm (Accurate) When Δ-RAM is full, no compression occurs. The Δ-RAM is either merged with an existing Δ-Flash, flushed directly to flash or discarded in the case that the

prefix sum cube is rebuilt.

Error Bound Greedy Heuristic (EBGreedy) The greedy heuristic is used to compress Δ-RAM before it is flushed or merged with a Δ-Flash. This is the greedy heuristic as presented in Section 4.4.1.

Δ-tree algorithm [1] (Δ-tree) The Δ-tree algorithm was proposed by Chun et al. [1]. This algorithm uses an R*-tree [43] to store the updates. For this we used the R*-tree implementation from the spatial index library [44].

For the first two algorithms above, a RAM cache was split into a read buffer and an area reserved for the Δ-RAM. This is our system as depicted in Figure 3. The read buffer cached both the Δ-Flashes and the prefix sum cube at the page grain and used the least recently used (LRU) algorithm for buffer replacement. The Δ-tree algorithm used the entire RAM cache as one buffer which cached both read and writes to the Δ-tree and the prefix sum cube at the page grain.

6. EXPERIMENTAL RESULTS

Four experiments were performed in order to compare the total execution time of the algorithms Accurate, EBGreedy and Δ-tree. The first experiment compares the four algorithms for varying RAM size. The second experiment compares the algorithms for varying prefix sum cube size. The third experiment compares the algorithms for varying error bound. Finally, the fourth experiment compares the algorithms for varying probability of update versus read queries.

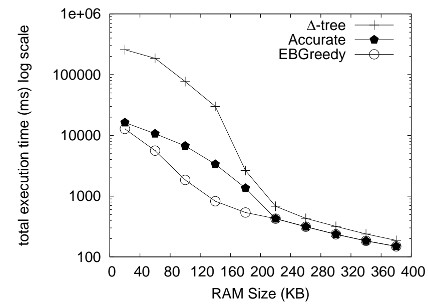
6.1. Varying RAM Size Experiment

This experiment compares the different algorithms for varying RAM size. The RAM sizes varies from 20KB up to 380KB in increments of 40KB. The default values are used for all other parameters.

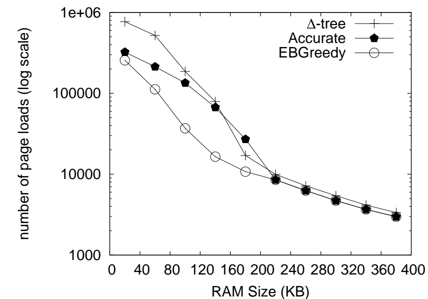
Figures 11 show the results of this experiment. The graphs show the results of the number of page loads from flash, the number of page writes to flash and the total execution time. The cost of erasing blocks of flash memory is also included in the total execution time metric. Note the y-axis of the graphs use log scales.

The results show that both of our algorithms outperform Δ-tree by a large margin (up to a factor of 10) for total execution time when RAM size is small (140 KB or less). This is because of the large number of writes incurred when the Δ-tree updates its R*-tree which is stored on flash memory. Insertions into the R*-tree can cause many node updates and splits which requires the concerned pages to be updated and new pages may also be inserted. The high cost of updates on flash memory makes Δ-tree much less efficient than our algorithms. When RAM size is large (above 220KB) all the algorithms perform about the same, since at that size, all popular pages can fit into the RAM buffer. The results show that EBGreedy generates less page loads than the accurate algorithm between RAM sizes of 20KB and 180KB. The reason EBGreedy generates less page loads is because it generates smaller Δ-Flashes which can be cached better in the read buffer. The smaller Δ-Flashes are created from compressing the Δ-RAM before flushing or merging it with previous Δ-Flashes. When the RAM size is large (220KB or above), the read buffer is able to fit almost all commonly used pages in the Δ-Flashes for both our algorithms, hence they perform about the same.

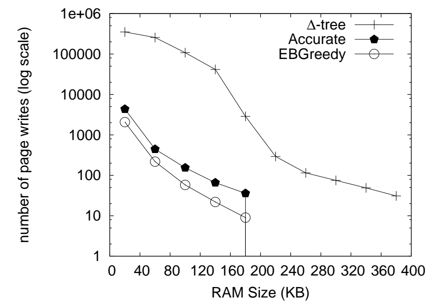
The results show that the number of writes goes to zero for our algorithms when the RAM size goes beyond 180 KB. The reason is when at these larger RAM sizes the Δ-RAM is large enough to fit all updates and therefore the Δ-RAM never needs to be flushed.



(a) Total Execution Time



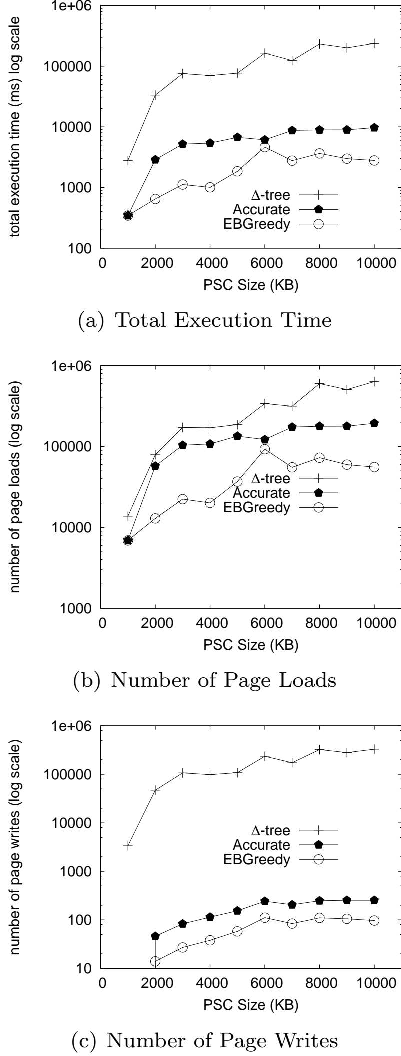
(b) Number of Page Loads



(c) Number of Page Writes

FIGURE 11. Varying RAM size results for data centred updates

6.2. Varying Prefix Sum Cube Size Experiment



(a) Total Execution Time

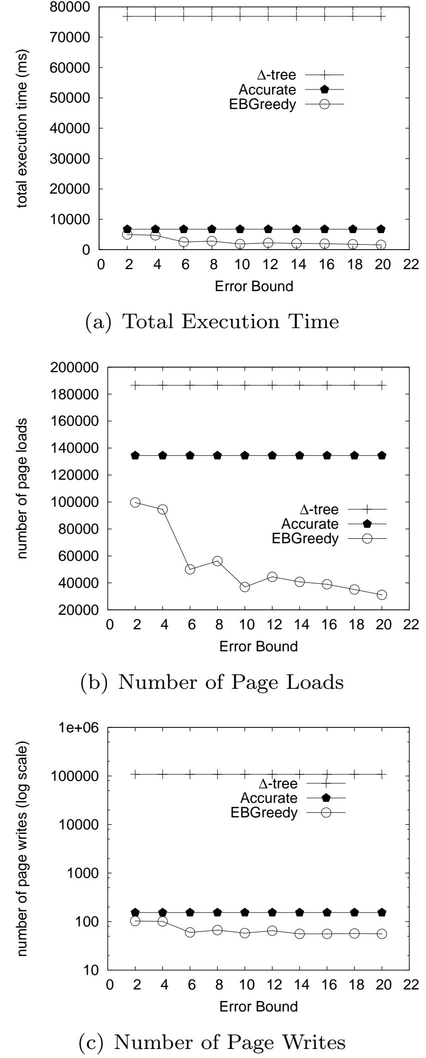
(b) Number of Page Loads

(c) Number of Page Writes

FIGURE 12. Varying prefix sum cube size results for data centred updates

This experiment compares the different algorithms for different prefix sum cube sizes. Therefore, this experiment compares the scalability of the algorithms by increasing data set size. The prefix sum cube size is varied from 1000KB to 10000KB in increments of 1000KB. The other parameters all use the default values.

The results are shown in Figure 12. Note the y-axis of the graphs use log scales. The results show that the total execution time of our algorithms scale well as the prefix sum cube grows in size. In addition, our algorithms consistently outperform the Δ-tree throughout the whole range of PSC size values. This is again due to the fact that our algorithms are much more efficient at handling updates.



(a) Total Execution Time

(b) Number of Page Loads

(c) Number of Page Writes

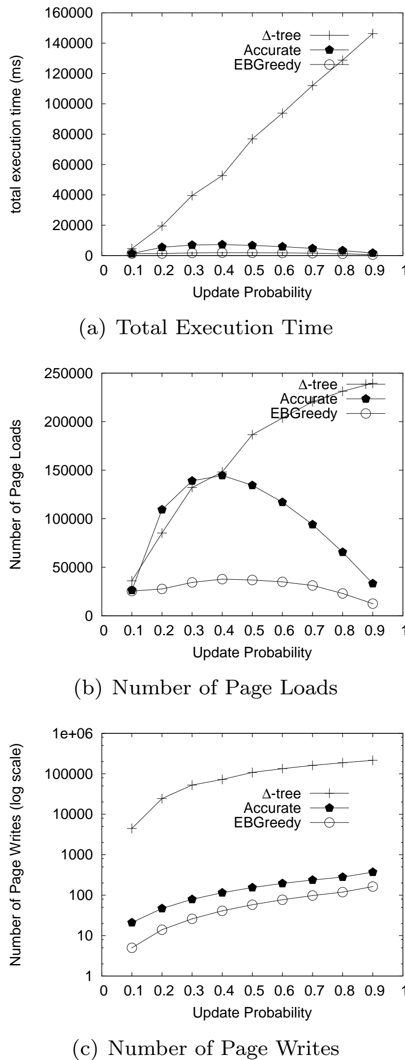
FIGURE 13. Varying error bound for data centred updates

6.3. Varying Error Bound Experiment

This experiment compares the algorithms for different error bounds. **In line with our problem definition, this is the absolute error not the percentage of error.** The error bound was varied from 2 to 20 in increments of 2. The other parameters all use the default values.

Figure 13 shows the results of the experiment. The results show that even with a low error bound of 2, EBGreedy outperforms Accurate. The results show the reason EBGreedy outperforms Accurate is because EBGreedy performs much less reads than Accurate. The lower number of reads is due to the smaller compressed Δ-RAMs that are flushed or merged by EBGreedy when Δ-RAM has reached its pre-allocated size limit.

6.4. Varying Update Probability



(a) Total Execution Time

(b) Number of Page Loads

(c) Number of Page Writes

FIGURE 14. Varying Update Probability results for data centered updates

This experiment compares the algorithms for different percentages of updates of the total queries. The other parameters used are set to the default values.

Figure 14 shows the results of the experiment. The results show that our algorithms outperform Δ-tree by a larger margin for total execution time as the update probability increases. The reason for this is that the Δ-tree is more efficient for reading than updating. When the percentage of updates increases, the high update cost weakness of Δ-tree becomes more pronounced, although Δ-tree is more efficient for reads that gets dwarfed by the high update costs due to the asymmetric read versus write cost for flash memory. It is encouraging to note that our algorithms outperform Δ-tree when update probability is only 0.2 which means 80% of queries are reads.

The results show that for our algorithms, when the probability is either high or low, the total execution time decreases. This is because if the probability of an update is low, then not many trees will be constructed, causing the cost of reading and writing the trees to be reduced. When the probability of an update is high, the probability of a read query is low and hence not many pages are loaded into RAM.

7. CONCLUSION

The aim of this paper was to find efficient algorithms and data structures for OLAP applications on flash memory. We achieve this by exploiting the fast random read characteristic of flash memory while minimising slow random writes. To do this we build quadtrees to store updates in RAM and write whole trees sequentially to flash memory once the RAM limit is reached. Each of these trees is either merged with an existing tree stored on flash memory or flushed as an additional tree. This approach minimises random writes by caching the writes in RAM and then converting them to sequential writes when RAM is full by writing whole trees. The cost of this approach is that we end up producing a large number of quadtrees which increases read time when answering queries. Here is where we leverage the fast random read performance of flash memory to produce lower overall execution time. In addition we try to minimise the amount of read by merging the quadtrees and compressing the quadtrees to produce smaller trees.

An error bound solution was also proposed. This solution gave a trade-off between efficiency and accuracy. Answers to queries are given within a user-defined error bound. The increased efficiency came from compressing the trees by deleting nodes before moving the trees to flash memory. A greedy heuristic was proposed to compress the trees, as this compression problem was proven to be NP complete.

A variety of experiments were conducted to compare our algorithms against the Δ-tree algorithm proposed by Chun et al. [1]. The results show our algorithms outperformed Δ-tree for total execution time for a wide variety of situations by orders of magnitude. Among our algorithms, the error bound solution was shown to outperform the accurate solution for total execution time and could therefore be used to trade-off accuracy for efficiency.

Possible future work includes exploring effective ways of answering range queries for the aggregate functions MIN and MAX. Another area of future work is to alter the error bound problem so that answers to range queries are within a percentage of error rather than absolute error.

8. ACKNOWLEDGEMENTS

This work is supported under the Australian Research Council's Discovery funding scheme (project number

DP0985451). We would also like to thank Michele Mooney for her careful proof reading of this paper.

REFERENCES

[1] Chun, S.-J., Chung, C.-W., Lee, J.-H., and Lee, S.-L. (2001) Dynamic update cube for range-sum queries. *Proceedings of International Conference on Very Large Data Bases*, San Francisco, CA, USA, pp. 521–530. Morgan Kaufmann Publishers Inc.

[2] Shukla, A., Deshpande, P., and Naughton, J. F. (1998) Materialized view selection for multidimensional datasets. *Proceedings of the International Conference on Very Large Data Bases*, San Francisco, CA, USA, pp. 488–499. Morgan Kaufmann Publishers Inc.

[3] Han, J., Pei, J., Dong, G., and Wang, K. (2001) Efficient computation of iceberg cubes with complex measures. *SIGMOD Record*, **30**, 1–12.

[4] Beyer, K. and Ramakrishnan, R. (1999) Bottom-up computation of sparse and Iceberg CUBE. *SIGMOD Record*, **28**, 359–370.

[5] Harinarayan, V., Rajaraman, A., and Ullman, J. D. (1996) Implementing data cubes efficiently. *SIGMOD Record*, **25**, 205–216.

[6] Baralis, E., Paraboschi, S., and Teniente, E. (1997) Materalized view selection in a multidimensional database. *Proceedings of International Conference on Very Large Data Bases*, San Francisco, CA, USA, pp. 98–112. Morgan Kaufmann Publishers Inc.

[7] Ross, K. A. and Srivastava, D. (1997) Fast computation of sparse datacubes. *Proceedings of International Conference on Very Large Data Bases*, San Francisco, CA, USA, pp. 116–125. Morgan Kaufmann Publishers Inc.

[8] Ho, C.-T., Agrawal, R., Megiddo, N., and Srikant, R. (1997) Range queries in OLAP data cubes. *SIGMOD Record*, **26**, 73–88.

[9] Gupta, A., Mumick, I. S., and Subrahmanian, V. S. (1993) Maintaining views incrementally. *Proceedings of SIGMOD International Conference on Management of Data*, New York, NY, USA, pp. 157–166. ACM.

[10] Griffin, T. and Libkin, L. (1995) Incremental maintenance of views with duplicates. *Proceedings of SIGMOD International Conference on Management of Data*, New York, NY, USA, pp. 328–339. ACM.

[11] Mumick, I. S., Quass, D., and Mumick, B. S. (1997) Maintenance of data cubes and summary tables in a warehouse. *SIGMOD Record*, **26**, 100–111.

[12] Labio, W., Yang, J., Cui, Y., Garcia-Molina, H., and Widom, J. (2000) Performance issues in incremental warehouse maintenance. *Proceedings of International Conference on Very Large Data Bases*, San Francisco, CA, USA, pp. 461–472. Morgan Kaufmann Publishers Inc.

[13] Folkert, N., Gupta, A., Witkowski, A., Subramanian, S., Bellamkonda, S., Shankar, S., Bozkaya, T., and Sheng, L. (2005) Optimizing refresh of a set of materialized views. *Proceedings of International Conference on Very Large Data Bases*, New York, NY, USA, pp. 1043–1054. ACM.

[14] Burdick, D., Deshpande, P. M., Jayram, T. S., Ramakrishnan, R., and Vaithyanathan., S. (2005) OLAP over Uncertain and Imprecise Data. *Proceedings of International Conference on Very Large Data Bases*, New York, NY, USA, pp. 970–981. ACM.

[15] Burdick, D., Deshpande, P. M., Jayram, T. S., Ramakrishnan, R., and Vaithyanathan., S. (2006) Efficient Allocation Algorithms for OLAP over Imprecise Data. *Proceedings of International Conference on Very Large Data Bases*, New York, NY, USA, pp. 391–402. ACM.

[16] Bourganim, L., Jonsson, B., and Bonnet, P. (2009) uFLIP: understanding flash IO patterns. *Fourth Biennial Conference on Innovative Data Systems Research (CIDR)*. Online Proceedings www.crdrdb.org.

[17] Nath, S. and Kansal, A. (2007) FlashDB: dynamic self-tuning database for NAND flash. *Proceedings of International Conference on Information Processing in Sensor Networks*, New York, NY, USA, pp. 410–419. ACM.

[18] Kim, G.-J., Baek, S.-C., Lee, H.-S., Lee, H.-D., and Joe, M. J. (2006) LGeDBMS: a small DBMS for embedded systems with flash memory. *Proceedings of International Conference on Very Large Data Bases*, New York, NY, USA, pp. 1255–1258. ACM.

[19] Lee, S.-W. and Moon, B. (2007) Design of flash-based DBMS: an in-page logging approach. *Proceedings of SIGMOD International Conference on Management of Data*, New York, NY, USA, pp. 55–66. ACM.

[20] Tsirogiannis, D., Harizopoulos, S., Shah, M. A., Wiener, J. L., and Graefe, G. (2009) Query processing techniques for solid state drives. *Proceedings of SIGMOD International Conference on Management of Data*, New York, NY, USA, pp. 59–72. ACM.

[21] Li, Y., He, B., Luo, Q., and Yi, K. (2009) Tree indexing on flash disks. *Proceedings of International Conference on Data Engineering*, pp. 1303–1306. IEEE.

[22] Shah, M. A., Harizonpoulos, S., Wiener, J. L., and Graefe, G. (2008) Fast scans and joins using flash drives. *Proceedings of the Fourth International Workshop on Database Management on New Hardware*, New York, NY, USA, pp. 17–24. ACM.

[23] Koltsidas, I. and Viglas, S. D. (2008) Flashing up the storage layer. *Proceedings of the VLDB Endowment*, **1**, 514–525.

[24] Wu, C.-H., Kuo, T.-W., and Chang, L. P. (2007) An efficient B-tree layer implementation for flash-memory storage systems. *ACM Transactions on Embedded Computing Systems (TECS)*, **6**, 19.

[25] Kim, J., Kim, J. M., Noh, S. H., Min, S. L., and Cho, Y. (2002) A space-efficient flash translation layer for compact flash systems. *IEEE Transactions on Consumer Electronics*, **48**, 366–375.

[26] Lee, S.-W., Park, D.-J., Chung, T.-S., Lee, D.-H., Park, S., and Song, H.-J. (2007) A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, **6**, 18.

[27] Wu, M. and Zwaenepoel, W. (1994) eNVy: a non-volatile, main memory storage system. *ASPLOS-VI: Proceedings of the sixth International Conference on Architectural support for programming languages and operating systems*, New York, NY, USA, pp. 86–97. ACM.

[28] Chiang, M.-L., Lee, P. C. H., and Chang, R.-C. (1999) Using data clustering to improve cleaning performance for flash memory. *Software: Practice and Experience*, **29**, 267–290.

[29] Chiang, M.-L., Lee, P. C. H., and Chang, R.-C. (1997) Managing flash memory in personal communication devices. *Proceedings of International Symposium on Consumer Electronics*, pp. 177–182. IEEE.

[30] Yao, A. (1985) On the complexity of maintaining partial sums. *SIAM Journal on Computing*, **14**, 277–288.

[31] Chazelle, B. and Rosenberg, B. (1989) Computing partial sums in multidimensional arrays. *Proceedings of the fifth annual symposium on computational geometry*, New York, NY, USA, pp. 131–139. ACM.

[32] Poon, C. (2003) Dynamic orthogonal range queries in OLAP. *Theoretical Computer Science*, **296**, 487–510.

[33] Gabow, H., Bentley, J., and Tarjan, R. (1984) Scaling and related techniques for geometry problems. *Proceedings of the sixteenth annual symposium on theory of computing*, New York, NY, USA, pp. 135–143. ACM.

[34] Lee, S., Ling, T., and Li, H. (2000) Hierarchical compact cube for range-max queries. *Proceedings of International Conference on Very Large Data Bases*, San Francisco, CA, USA, pp. 232–241. Morgan Kaufmann Publishers Inc.

[35] Yuan, H. and Atallah, M. (2010) Data Structures for Range Minimum Queries in Multidimensional Arrays. *Twenty-First ACM-SIAM Symposium on Discrete Algorithms*, New York, NY, USA, pp. 150–160. ACM.

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

[37] Jagadish, H., Narayan, P. P. S., Seshadri, S., Sudarshan, S., and Kanneganti, R. (1997) Incremental organization for data recording and warehousing. *Proceedings of International Conference on Very Large Data Bases*, San Francisco, CA, USA, pp. 16–25. Morgan Kaufmann Publishers Inc.

[38] Muth, P., O'Neil, P., Pick, A., and Weikum, G. (1998) Design, implementation, and performance of the LHAM log-structured history data access method. *Proceedings of International Conference on Very Large Data Bases*, San Francisco, CA, USA, pp. 452–463. Morgan Kaufmann Publishers Inc.

[39] Muth, P., O'Neil, P., Pick, A., and Weikum, G. (2000) The LHAM log-structured history data access method. *The VLDB Journal*, **8**, 199–221.

[40] Hjaltason, G. and Samet, H. (2002) Speeding up construction of PMR quadtree-based spatial indexes. *The VLDB Journal*, **11**, 109–137.

[41] Park, S.-Y., Jung, D., Kang, J.-U., Kim, J.-S., and Lee, J. (2006) CFLRU: a replacement algorithm for flash memory. *Proceedings of International Conference on Compilers, architecture and synthesis for embedded systems*, New York, NY, USA, pp. 234–241. ACM.

[42] United States of America 2000 census data. http://www2.census.gov/census_2000/datasets /PUMS/FivePercent.

[43] Beckmann, N., Kriegel, H., Schneider, R., and Seeger, B. (1990) The R*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Record*, **19**, 322–331.

[44] Hadjieleftheriou, M. (Downloaded: August 2008). Spatial index library. http://www2.research.att.com/~marioh/spatialindex/.

APPENDIX A. NP COMPLETENESS PROOF

We prove by restriction that the quadtree compression problem is NP complete. We first define a restricted version of the quadtree compression problem to aid in proving that the quadtree compression problem is NP complete. We can prove that our problem is NP complete by showing that we can restrict our problem, and that this restricted problem is equivalent to a known NP complete problem.

Restricted Δ-quadtree compression problem
Given a set A of possible nodes to delete, find $S \subseteq A$ such that the error bound consumed $EBC(S) = \Sigma_{s \in S}(EB - MEL(s))NC(s)$ is minimised and the following constraint is satisfied:
$\frac{ML}{TSD(S)} \geq c$

We make two restrictions. The first is restricting our problem to only leaf nodes. The second is restructuring the problem to compressing quadtrees which only include nodes where $MEL(s) \geq 0$. We make another restriction that $EBC(s)$ (which is mapped to the value metric in the knapsack problem) is restricted from the domain of real numbers to the integers. These are valid restrictions for proving NP completeness because they represent possible instances of the Δ-quadtree compression problem and therefore can be used to prove NP completeness.

We next define the Δ-quadtree nodes retension problem which is equivalent to the restricted quadtree compression problem of selecting the nodes to retain. By showing a restricted version of our problem is equivalent to the knapsack problem, which is known to be NP complete, we can prove that the Δ-quadtree compression problem is NP complete.

Δ-quadtree nodes retension problem
Given a set A of possible nodes to keep, find $T \subseteq A$ such that $\Sigma_{t \in T}(EB - MEL(t))NC(t)$ is maximised and the following constraint is met:
$\Sigma_{t \in T} sizeb(t) \leq \frac{ML}{c}$

To prove that the Δ-quadtree compression problem is NP complete using knapsack, we first have to prove that the restricted version of the quadtree compression and quadtree node retension problems are equivalent. In order to do this, we define and prove the following two lemmas, which are then used to prove

that the two problems are equivalent.

LEMMA A.1.: *$\frac{ML}{TSD(S)} \geq c$ is equivalent to $\Sigma_{t \in T} \leq \frac{ML}{c}$*

Proof: $\frac{ML}{TSD(S)} \geq c \Leftrightarrow TSD(S) \leq \frac{ML}{c}$
So Lemma A.1 is true iff $TSD(S) = \Sigma_{t \in T} sizeb(t)$
and $TSD(S) = \Sigma_{a \in A} sizeb(a) - \Sigma_{s \in S} sizeb(s) = \Sigma_{t \in T} sizeb(t)$
Therefore, Lemma A.1 is true. □

LEMMA A.2.: *Minimising $\Sigma_{s \in S}(EB - MEL(s))NC(s)$ is equivalent to maximising $\Sigma_{t \in T}(EB - MEL(t))NC(t)$*

Proof: Each node in the Δ-quadtree has an associated value $(EB - MEL(a))NC(a)$, so we have $\Sigma_{a \in A}(EB - MEL(a))NC(a) = \Sigma_{s \in S}(EB - MEL(s))NC(s) + \Sigma_{t \in T}(EB - MEL(t))NC(t)$. So, minimising $\Sigma_{s \in S}(EB - MEL(s))NC(s)$ is equivalent to minimising $\Sigma_{a \in A}(EB - MEL(a))NC(a) - \Sigma_{t \in T}(EB - MEL(t))NC(t)$. Now because EB, A, $MEL(a)$ and $NC(a)$ are all constant for any given Δ-quadtree, $\Sigma_{a \in A}(EB - MEL(a))NC(a)$ is just some constant q. So minimising $\Sigma_{s \in S}(EB - MEL(s))NC(s)$ is equivalent to minimising $q - \Sigma_{t \in T}(EB - MEL(t))NC(t)$ which is equivalent to maximising $\Sigma_{t \in T}(EB - MEL(t))NC(t)$. Therefore, Lemma A.2 is true. □

THEOREM A.1. *The restricted quadtree compression problem and quadtree nodes retension problem are equivalent.*

Proof: By Lemma A.1, we have that the constraint of the quadtree nodes retension problem is equivalent to the constraint of the quadtree compression problem. By Lemma A.2, we have that the metrics of the two problems are also equivalent. Therefore, Theorem A.1 is true. □

THEOREM A.2.: *The Δ-quadtree compression problem is NP complete.*

Proof: We prove that the Δ-quadtree compression problem is NP complete by restricting our problem and mapping an equivalent converse problem to the knapsack problem which is known to be NP complete. In order to find a mapping between knapsack and our problem, by Theorem A.1, we can state the restricted problem as its equivalent corresponding problem of finding the set of nodes to keep rather than delete.

We map the quadtree nodes retension problem to the knapsack problem as follows:
t is equivalent to the kth item in the knapsack problem,
$EBC(t)$ is equivalent to the value of the kth item, where t is the selected non-compressed node and k is the selected item in the knapsack problem,
$sizeb(t)$ is equivalent to the size of the kth item in the knapsack problem and
size limit in the knapsack problem is equivalent to $\frac{ML}{c}$.

We have proven by restriction that the Δ-quadtree compression problem is NP complete using the above mappings and restricitions. □