Visionary Paper

Designing Access Methods: The RUM Conjecture

Manos Athanassoulis* Michael S. Kester* Lukas M. Maas* Radu Stoica†
Stratos Idreos* Anastasia Ailamaki‡ Mark Callaghan◇

*Harvard University †IBM Research, Zurich ‡EPFL, Lausanne ◇Facebook, Inc.

ABSTRACT

The database research community has been building methods to store, access, and update data for more than four decades. Throughout the evolution of the structures and techniques used to access data, *access methods* adapt to the ever changing hardware and workload requirements. Today, even small changes in the workload or the hardware lead to a redesign of access methods. The need for new designs has been increasing as data generation and workload diversification grow exponentially, and hardware advances introduce increased complexity. New workload requirements are introduced by the emergence of new applications, and data is managed by large systems composed of more and more complex and heterogeneous hardware. As a result, it is increasingly important to develop application-aware and hardware-aware access methods.

The fundamental challenges that every researcher, systems architect, or designer faces when designing a new access method are how to minimize, i) read times (R), ii) update cost (U), and iii) memory (or storage) overhead (M). In this paper, we conjecture that when optimizing the read-update-memory overheads, optimizing in any two areas negatively impacts the third. We present a simple model of the RUM overheads, and we articulate the *RUM Conjecture*. We show how the RUM Conjecture manifests in state-of-the-art access methods, and we envision a trend toward RUM-aware access methods for future data systems.

1. INTRODUCTION

Chasing Access Paths. Picking the proper physical design (through static autotuning [14], online tuning [13], or adaptively [31]) and access method [27, 49] have been key research challenges of data management systems for several decades. The way we physically organize data on storage devices (disk, flash, memory, caches) defines and restricts the possible ways that we can read and update it. For example, when data is stored in a heap file without an index, we have to perform costly scans to locate any data we are interested in. Conversely, a tree index on top of the heap file, uses additional space in order to substitute the scan with a more lightweight index probe. Over the years, we have seen a plethora of exciting and innovative proposals for data structures and algorithms, each one tailored to a set of important workload patterns, or for matching critical hardware characteristics. Applications evolve rapidly and continuously, and at the same time, the underlying hardware is diverse and changes quickly as new technologies and architectures are developed [1]. Both trends lead to new challenges when designing data management software.

The RUM Tradeoff. A close look at existing proposals on access methods[1] reveals that each is confronted with the same fundamental challenges and design decisions again and again. In particular, there are three quantities and design parameters that researchers always try to minimize: (1) the read overhead (**R**), (2) the update overhead (**U**), and (3) the memory (or storage) overhead (**M**), henceforth called the *RUM overheads*. Deciding which overhead(s) to optimize for and to what extent, remains a prominent part of the process of designing a new access method, especially as hardware and workloads change over time. For example, in the 1970s one of the critical aspects of every database algorithm was to minimize the number of random accesses on disk; fast-forward 40 years and a similar strategy is still used, only now we minimize the number of random accesses to main memory. Today, different hardware runs different applications but the concepts and design choices remain the same. New challenges, however, arise from the exponential growth in the amount of data generated and processed, and the wealth of emerging data-driven applications, both of which stress existing data access methods.

The RUM Conjecture: Read, Update, Memory – Optimize Two at the Expense of the Third. An ideal solution is an access method that always provides the lowest read cost, the lowest update cost, and requires no extra memory or storage space over the base data. In practice, data structures are designed to compromise between the three RUM overheads, while the optimal design depends on a multitude of factors like hardware, workload, and user expectations.

We analyze the lower bounds for the three overheads (read - update - memory) given an access method which is perfectly tailored for minimizing each overhead and we show that such an access method will impact the rest of the overheads negatively. We take this observation a step further and propose the RUM Conjecture: *designing access methods that set an upper bound for two of the RUM overheads, leads to a hard lower bound for the third overhead which cannot be further reduced*. For example, in order to minimize the cost of updating data, one would use a design based on differential structures, allowing many queries to consolidate updates and avoid the cost of reorganizing data. Such an approach, however, increases the space overhead and hinders read cost as now queries need to merge any relevant pending updates during processing. Another example is that the read cost can be minimized by

[1] Access methods: *algorithms and data structures for organizing and accessing data* [27].

storing data in multiple different physical layouts [4, 17, 46], each layout being appropriate for minimizing the read cost for a particular workload. Update and space costs, however, increase because now there are multiple data copies. Finally, the space cost can be minimized by storing minimal metadata and, hence, pay the price of increased search time when reading and updating data.

The three RUM overheads form a competing triangle. In modern implementations of data systems, however, one can optimize up to some point for all three. Such optimizations are possible by relying on using inherently defined structure instead of storing detailed information. A prime example is the block-based clustered indexing, which reduces both read and memory overhead by storing only a few pointers to pages (only when an indexed tuple is stored to a different page than the previous one), hence building a very short tree. The reason why such an approach would give us good read performance is the fact that data is clustered on the index attribute. Even in this ideal case, however, we have to perform additional computation in order to calculate the exact location of the tuple we are searching for. In essence, we use computation and knowledge about the data in order to reduce the RUM overheads. Another example, is the use of compression in bitmap indexes; we still use additional computation (compression/decompression) in order to succeed in reducing both read and memory overhead of the indexes. While the RUM overheads can be reduced by computation or engineering, their competing nature manifests in a number of approaches and guides our roadmap for RUM-aware access methods.

RUM-Aware Access Method Design. Accepting that a perfect access method does not exist, does not mean the research community should stop striving to improve; quite the opposite. The RUM Conjecture opens the path for exciting research challenges towards the goal of creating RUM-aware and RUM-adaptive access methods.

Future data systems should include versatile tools to interact with the data the way the workload, the application, and the hardware need and not vice versa. In other words, the application, the workload, and the hardware should dictate how we access our data, and not the constraints of our systems. Such versatile data systems will allow the data scientist of the future to dynamically tune the data access methods during normal system operation. Tuning access methods becomes increasingly important if, on top of big data and hardware, we consider the development of specialized systems and tools to cater data, aiming at servicing a narrow set of applications each. As more systems are built, the complexity of finding the right access method increases as well.

Contributions. In this paper we present for the first time the RUM Conjecture as a way to understand the inherent tradeoffs of every access method. We further use the conjecture as a guideline for designing access methods of future data systems. In the remainder of this paper we charter the path toward RUM-aware access methods:

- Identify and model the RUM overheads (§2).
- Propose the RUM Conjecture (§3).
- Document the manifestation of the RUM Conjecture in state-of-the-art access methods (§4).
- Use the RUM Conjecture to build access methods for future data systems (§5).

Each of the above points serves as inspiration for further research on data management. Finding a concise yet expressive way to identify and model the fundamental overheads of access methods requires research in data management from a theory perspective. Similarly, proving the RUM Conjecture will expand on this line of research. Documenting the manifestation of the RUM Conjecture entails a new study of access methods with the RUM overheads in mind when modeling and categorizing access methods. Finally, the first three steps provide the necessary research infrastructure to build powerful access methods.

While we envision that this line of research will enable building powerful access methods, the core concept of the RUM Conjecture is, in fact, that there is no panacea when designing systems. It is not feasible to build the universally best access method, nor to build the best access method for each and every use case. Instead, we envision that the RUM Conjecture will create a trend toward building access methods that can efficiently morph to support changing requirements and different software and hardware environments. The remainder of this paper elaborates one by one the four steps toward RUM-aware access methods for future data systems.

2. THE RUM OVERHEADS

Overheads of Access Methods. When designing access methods it is very important to understand the implications of different design choices. In order to do so, we discuss the three fundamental overheads that each design decision can affect. Access methods enable us to read or update the main data stored in a data management system (hereafter called *base data*), potentially using auxiliary data such as indexes (hereafter called *auxiliary data*), in order to offer performance improvements. The overheads of an access method quantify the additional data accesses to support any operation, relative to the base data.

Read Overhead (*RO*). The *RO* of an access method is given by the data accesses to auxiliary data when retrieving the necessary base data. We refer to *RO* as the read amplification: the ratio between the total amount of data read including auxiliary and base data, divided by the amount of retrieved data. For example, when traversing a B^+-Tree to access a tuple, the *RO* is given by the ratio between the total data accessed (including the data read to traverse the tree and the base data) and the base data intended to be read.

Update Overhead (*UO*). The *UO* is given by the amount of updates applied to the auxiliary data in addition to the updates to the main data. We refer to *UO* as the write amplification: the ratio between the size of the physical updates performed for one logical update, divided by the size of the logical update. In the previous example, the *UO* is calculated by dividing the updated data size (both base and auxiliary data) by the size of the updated base data.

Memory Overhead (*MO*). The *MO* is the space overhead induced by storing auxiliary data. We refer to *MO* as the space amplification, defined as the ratio between the space utilized for auxiliary and base data, divided by the space utilized for base data. Following the preceding example, the *MO* is computed by dividing the overall size of the B^+-Tree by the base data size.

Minimizing RUM overheads. Ideally, when building an access method, all three overheads should be minimal, however, depending on the application, the workload, and the available technology they are prioritized. While access time, optimized by minimizing read overhead, often has top priority, the workload or the underlying technology sometimes shift priorities. For example, storage with limited endurance (like flash-based drives) favors minimizing the update overhead, while the slow speed of main memory and the scarce cache capacity justifies reducing the space overhead. The theoretical minimum for each overhead is to have the ratio equal to 1.0, implying that the base data is always read and updated directly and no extra bit of memory is wasted. Achieving these bounds for all three overheads simultaneously, however, is not possible as there is always a price to pay for every optimization.

In the following discussion we reason about the three overheads and their lower bounds using a simple yet representative case for

base data: an array of integers. We organize this dataset consisting of N ($N >> 1$) fixed-sized elements in blocks, each one holding a *value*. Every block can be identified by a monotonically increasing ID, *blkID*. The workload is comprised of point queries, updates, inserts, and deletes. We purposefully provide simple examples of data structures in order to back the following hypothesis. The generality of these examples lies in their simplicity.

Hypothesis. *An access method that is optimal with respect to one of the read, update, and memory overheads, cannot achieve the optimal value for both remaining overheads.*

Minimizing Only *RO*. In order to minimize *RO* we organize data in an array and we store each *value* in the block with $blkID = value$. For example, the relation $\{1, 17\}$ is stored in an array with 17 blocks. The first block holds value 1, the last block holds value 17, and every block in-between holds a null value. *RO* is now minimal because we always know where to find a specific value (if it exists), and we only read useful data. On the other hand, the array is sparsely populated, with unbounded *MO* since, in the general case, we cannot anticipate what would be the maximum value ever inserted. When we insert or delete a value we only update the corresponding block. When we change a value we need to update two blocks: empty the old block and insert the new value in its new block, effectively, increasing the worst case *UO* to two physical updates for one logical update.

Prop. 1 $min(RO) = 1.0 \Rightarrow UO = 2.0$ *and* $MO \rightarrow \infty$

Minimizing Only *UO*. In order to minimize *UO*, we append every update, effectively forming an ever increasing log. That way we achieve the minimum *UO*, which is equal to 1.0, at the cost of continuously increasing *RO* and *MO*. Notice that any reorganization of the data to reduce *RO* would result in an increase of *UO*. Hence, for minimum *UO*, both *RO* and *MO* perpetually increase as updates are appended.

Prop. 2 $min(UO) = 1.0 \Rightarrow RO \rightarrow \infty$ *and* $MO \rightarrow \infty$

Minimizing Only *MO*. When minimizing *MO*, no auxiliary data is stored and the base data is stored as a dense array. During a selection, we need to scan all data to find the values we are interested in, while updates are performed in place. The minimum *MO*=1.0 is achieved. The *RO*, however, is now dictated by the size of the relation since a full scan is needed in the worst case. The *UO* cost of in-place updates is also optimal because only the base data intended to be updated is ever updated.

Prop. 3 $min(MO) = 1.0 \Rightarrow RO = N$ *and* $UO = 1.0$

3. THE RUM CONJECTURE

Achieving the optimal value for one overhead is not always as important as finding a good balance across all RUM overheads. In fact, in the previous section, we saw that striving for the optimal may create impractical access method designs. The next logical step is to provide a fixed bound only for one of the overheads, however, this raises the question of how to quantify the impact of such an optimization goal for the remaining two overheads. Given our analysis above, we conjecture that it is not possible to optimize across all three dimensions at the same time.

The RUM Conjecture. *An access method that can set an upper bound for two out of the read, update, and memory overheads, also sets a lower bound for the third overhead.*

In other words, we can choose which two overheads to prioritize and optimize for, and pay the price by having the third overhead greater than a hard lower bound. In the following section we describe how the RUM Conjecture manifests in existing access methods by showing that current solutions are typically optimized for one of the three overheads, while in Section 5 we discuss a roadmap for RUM access methods.

4. RUM IN PRACTICE

The RUM Conjecture captures in a concise way the tension between different optimization goals that researchers face when building access methods. We present here the competing nature of the RUM tradeoffs, as it manifests in state-of-the-art access methods. Different access method designs can be visually represented based on their RUM balance. The RUM tradeoffs can be seen as a three dimensional design space or, if projected on a two-dimensional plane, as the triangle shown in Figure 1. Each access method is mapped to a point or – if it can be tuned to have varying RUM behavior – to an area. Table 1 shows the time and size complexity of representative data structures, illustrating the conflicting read, update, and memory overheads.

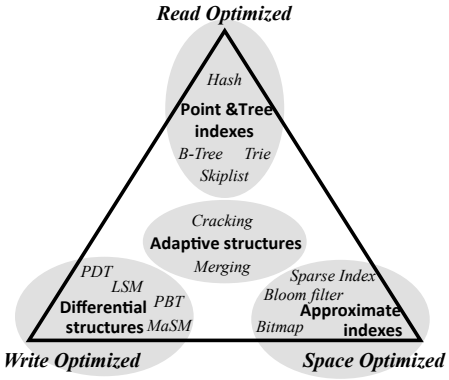

Figure 1: Popular data structures in the RUM space.

Read-optimized access methods (top corner in Figure 1) are optimized for low read overhead. Examples include indexes with constant time access such as hash-based indexes or logarithmic time structures such as *B-Trees* [22], *Tries* [19], *Prefix B-Trees* [9], and *Skiplists* [45]. Typically, such access methods offer fast read access but increase space overhead and suffer with frequent updates.

Write-optimized differential structures (left corner in Figure 1) reduce the write overhead of in-place updates by storing updates in secondary differential data structures. The fundamental idea is to consolidate updates and apply them in bulk to the base data. Examples include the *Log-structured Merge Tree* [44], the *Partitioned B-tree* (PBT) [21], the *Materialized Sort-Merge* (MaSM) algorithm [7, 8], the *Stepped Merge* algorithm [35], and the *Positional Differential Tree* [28]. *LA-Tree* [3] and *FD-Tree* [38] are two prime examples of write optimized trees aiming at exploiting flash while respecting at the same time its limitations, e.g., the asymmetry between read and write performance [6] and the bounded number of physical updates flash can sustain [7]. Typically, such data structures offer good performance under updates but increase read costs and space overhead.

Space-efficient access methods (right corner in Figure 1) are designed to reduce the storage overhead. Example categories include compression techniques and lossy index structures such as *Bloom filters* [12], lossy hash-based indexes like *count-min sketches* [16], *bitmaps with lossy encoding* [51], and *approximate tree indexing* [5, 40]. *Sparse indexes*, which are light-weight secondary indexes, like *ZoneMaps* [18], *Small Materialized Aggregates* [42] and *Col-*

Parameter	N	m	B	P	T	MEM
Explanation	dataset size (#tuples)	query result size (#tuples)	block size (#tuples)	partition size (#tuples)	LSM levels ratio	memory (#pages)

Access Method	Bulk Creation Cost	Index Size	Point Query	Range Query (size: m)	Insert/Update/Delete
B$^+$-Tree	$O(N/B \cdot log_{MEM/B}(N/B))$	$O(N/B)$	$O(log_B(N))$	$O(log_B(N)+m)$	$O(log_B(N))$
Perfect Hash Index	$O(N)$	$O(N/B)$	$O(1)$	$O(N/B)$	$O(1)$
ZoneMaps	$O(N/B)$	$O(N/P/B)$	$O(N/P/B)$	$O(N/P/B)$	$O(N/P/B)$
Levelled LSM	N/A	$O\left(\frac{N \cdot T}{T-1}\right)$	$O(log_T(N/B) \cdot log_B(N))$	$O\left(log_T(N/B) \cdot log_B(N) + \frac{m \cdot T}{T-1}\right)$	$O(T/B \cdot log_T(N/B))$
Sorted column	$O(N/B \cdot log_{MEM/B}(N/B))$	$O(1)$	$O(log_2(N))$	$O(log_2(N)+m)$	$O(N/B/2)$
Unsorted column	$O(1)$	$O(1)$	$O(N/B/2)$	$O(N/B)$	$O(1)$

Table 1: The base data typically exist either as a *sorted column* or as an *unsorted column*. When using an additional index we (i) spend time building it, (ii) allocate space for it, and (iii) pay the cost to maintain it. The I/O cost [2] (time and space complexity) of four representative access methods (B$^+$-Trees, Hash Indexes, ZoneMaps, and levelled LSM) illustrates that there is no single winner. ZoneMaps have the smaller size – being a sparse index, but Hash Indexes offer the fastest point queries, while B$^+$-Trees offer the fastest range queries. Similarly, the update cost is best for Hash Indexes, while LSM can support efficient range queries having very low update cost as well.

umn Imprints [50] fall into the same category. Typically, such data structures and approaches reduce the space overhead significantly but increase the write costs (e.g., when using compression) and sometimes increase the read costs as well (e.g., a sparse index).

Adaptive access methods (middle region in Figure 1) are comprised of flexible data structures designed to gradually balance the RUM tradeoffs by using the workload access pattern as a guide. Most existing data structures provide tunable parameters that can be used to balance the RUM tradeoffs offline, however, adaptive access methods balance the tradeoffs online across a larger area of the design space. Notable proposals are *Database Cracking* [31, 32, 33, 48], *Adaptive Merging* [22, 25], and *Adaptive Indexing* [23, 24, 26, 34], which balance the read performance versus the overhead of creating an index. The incoming queries dictate which part of the index should be fully populated and tuned. The index creation overhead is amortized over a period of time, and it gradually reduces the read overhead, while increasing the update overhead, and slowly increasing the memory overhead. Although much more flexible than traditional data structures, existing adaptive data structures cannot cover the whole RUM spectrum as they are designed for a particular type of hardware and application.

The RUM tradeoffs of four representative access methods as well as two data organizations are presented in Table 1 in order to illustrate the need to balance them. The memory overhead is represented in the form of space complexity (index size), and the read and the update overheads in the form of the I/O complexity of the operations. We differentiate between point queries and range queries, to allow for a more detailed classification of access methods. We examine B$^+$-Trees[2], Hash Indexing, ZoneMaps[3], and levelled LSM [36], assuming that each LSM level is a B$^+$-Tree with branch factor B. Typically, sparse indexing like ZoneMaps gives the lowest access method size, however, it delivers neither the best point query performance, nor the best range query performance. The lowest point query complexity is provided by Hash Indexing, and the lowest range query complexity by B$^+$-Trees. In addition, even without any additional secondary index, maintaining a sorted column allows for searching with logarithmic cost, with the downside of having linear update cost. Hence, even without an auxiliary data structure, adding structure to the data affects read and write behavior. We envision RUM access methods to take this a step further, and morph between data structures and different data organizations in order to build access methods that have tunable performance and can change behavior both adaptively and on-demand.

[2]Bulk loading requires sorting. The best sorting algorithm depends on type of storage. Here we assume external multi-way mergesort.
[3]We consider the best case for ZoneMaps; only a single partition needs to be read or updated.

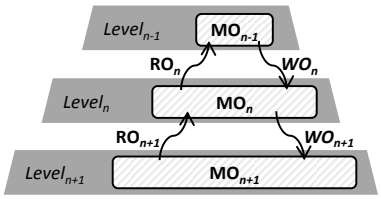

Figure 2: RUM overheads in memory hierarchies.

The Memory Hierarchy. For ease of presentation, the previous discussion assumes that all data objects are stored on the same storage medium. Real systems, however, have a more complex memory/storage hierarchy making it harder to design and tune access methods. Data is stored persistently only at the lower levels of the hierarchy and is replicated, in various forms, across all levels of the hierarchy; each memory level experiences different access patterns, resulting in different read and write overheads, while the space overhead at each level depends on how much data is cached.

Several approaches leverage knowledge about the memory hierarchy to offer better read performance. *Fractal Prefetching B$^+$-Trees* [15] use different node sizes for disk-based and in-memory processing in order to have the optimal for both cases. *Cache-sensitive B$^+$-Trees* [47] physically cluster sibling nodes together to reduce the number of cache misses, and decrease the node size using offsets rather than pointers. *SB-Trees* [43] operate in an analogous way when the index is disk-based, while *BW-Tree* [37] and *Masstree* [41] presents a number of optimizations related to cache memory, main memory and flash-based secondary storage. *SILT* [39] combines write-optimized logging, read-optimized immutable hashing, and, a sorted store, careful designed around the memory hierarchy to balance the tradeoffs of its various levels.

The RUM tradeoffs, however, still hold for each level individually as shown in Figure 2. The fundamental assumption that data has a minimum access granularity holds for all storage mediums today, including main memory, flash storage, and disks; the only difference is that both access time and access granularity vary. The RUM tradeoffs can also be viewed vertically rather than horizontally. For example, the RO_n read and the UO_n update overheads at memory level n can be reduced by storing more data, updates, or meta-data, at the previous level $n-1$, which results, at least, in a

higher MO_{n-1}. Overall, we expect this interaction of hardware and software to become increasingly more complex as hierarchies become deeper and as hardware trends shift the relative performance of one level compared to the others, resulting in the need for more fine tuning of access methods.

Viewing the hierarchy from the bottom towards the top, we first have cold storage which today may be shingled disks [29], or traditional rotational disks. They are followed typically by flash storage for buffering and read performance. The next levels are main memory and the different levels of cache memory. In the future an additional layer of non-volatile main memory will be added or it will replace main memory altogether [30]. Different layers of this new storage and memory hierarchy have different requirements. More specifically, shingled disks are similar to flash devices regarding the need to minimize update cost to respect their internal characteristics. On the other hand, when designing access methods for traditional rotational disks – sitting in-between shingled disks and flash in the hierarchy – we need to minimize the read overhead. As we move higher in the hierarchy, read performance and index size is typically more important than update cost.

Cache-Oblivious Access Methods. A different way to build access methods is to completely remove the memory hierarchy from the design space, using cache-oblivious algorithms [20]. Cache-oblivious access methods, however, achieve that by having a larger constant factor in read performance [11]. In addition, cache-oblivious access methods have a larger memory overhead because they require more pointers to guarantee that search performance will be orthogonal to the memory hierarchy [10]. Finally, cache-oblivious designs are less tunable. In order to tune a data structure and be able to balance between read performance, update performance, and memory consumption we need to be cache-aware [10]. As a result, in order to build RUM-tunable access methods we have to use a cache-aware design and take into account the exact shape of the memory hierarchy.

5. BUILDING RUM ACCESS METHODS

For several decades, the database research community has been redesigning access methods, trying to balance the RUM tradeoffs with every change in hardware, workload patterns, and applications. As a result, every few years new variations of data management techniques are proposed and adaptation to new challenges becomes increasingly harder. For example, most data management software is still unfit to natively exploit solid-state drives, multi-cores, and deep non-uniform memory hierarchies, even though the concepts used to adapt past techniques and structures rely on ideas that were proposed several decades ago (like partitioning, and avoiding random access). In other words, what changes is how one *tunes* the structures and techniques for the new environment.

Both hardware and applications change rapidly and continuously. As a result, we need to frequently adjust data management software to meet the evolving challenges. In the previous sections we laid the groundwork for RUM access methods, by providing an intuitive analysis of the RUM overheads and the RUM Conjecture. Moving further, here we propose the necessary research steps to design versatile access methods that have variable balance between the RUM overheads. In Figure 3 we visualize the ideal RUM access method, which will be able to seamlessly transition between the three extremes: read optimized, write optimized, and space optimized. In practice, it may not be feasible to aim for building a single access method able to cover the whole spectrum. Instead, an alternative approach is to build multiple access methods able to navigate partly in the RUM space, however, covering the whole space in aggregate.

Studying The RUM Tradeoffs. The first step towards building RUM access methods is to extend the discussion in Section 4. A detailed study of the nature of RUM tradeoffs in practice, will lead to a detailed classification of access methods based on their RUM balance. Most existing access methods can be depicted as a static point in the RUM space (Figure 3). The exact position of the point may differ based on some parameters (for example, the fan-out of B^+-Trees, the number of partitions in PBT, the number of sorted runs in MaSM). Moreover, adaptive indexing techniques like cracking behave in a dynamic manner yet are not tuneable: as they touch more data they add structure to the data and gradually reduce the read overhead at the expense of update overhead.

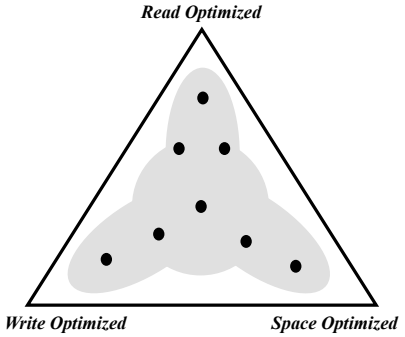

Figure 3: Tunable behavior in the RUM space.

A concrete outcome of this analysis is the gleaning of all the fundamental building blocks and strategies of access methods. For example, logarithmic access cost (trees, exponentially increasing logs), fixed access cost (tries, hash tables), trading-off computation for auxiliary data size (hashing for hash tables, compression for bitmaps), and lazy updates (log-structure approaches).

Tunable RUM Balance. Using the above classification and analysis we can make educated decisions about which access method should be used based on the application requirements and the hardware characteristics, effectively creating a powerful access method wizard. In addition to that, we investigate how to build access methods that have tunable behavior. Such access methods are not single points in the RUM space; instead they can move within an area in the design space.

We envision future data systems with a suite of access methods that can easily adapt to different optimization goals. For example:

- B^+-Trees that have dynamically tuned parameters, including tree height, node size, and split condition, in order to adjust the tree size, the read cost, and the update cost at runtime.
- Approximate (tree) indexing that supports updates with low read performance overhead, by absorbing them in updatable probabilistic data structures (like quotient filters).
- Morphing access methods, combining multiple shapes at once. Adding structure to data gradually with incoming queries, and building supporting index structures when further data reorganization becomes infeasible.
- Update-friendly bitmap indexes, where updates are absorbed using additional, highly compressible, bitvectors which are gradually merged.
- Access methods with iterative logs enhanced by probabilistic data structures that allows for more efficient reads and updates by avoiding accessing unnecessary data at the expense of additional space.

Dynamic RUM Balance. We envision access methods that can automatically and dynamically adapt to new workload requirements

or hardware changes, like a sudden increase or decrease of availability of storage or memory. For example, in the case of access methods based on iterative merges, by changing the number of merge trees dynamically, the depth of the merge hierarchy and the frequency of merging, we can build access methods that dynamically adapt to workload and hardware changes.

Compression and Computation. Orthogonally to the tension between the three overheads, when accessing data today compression is often used to reduce the amount of data to be moved. This trade-off between computation (compressing/decompressing) and data size does not affect the fundamental nature of the RUM Conjecture. Compression is seldom used only for transferring data through the memory hierarchy. Rather, modern data systems operate mostly on compressed data and decompress as late as possible, usually when presenting the final answer of a query to the user.

6. SUMMARY

Changes in hardware, applications and workloads have been the main driving forces in redesigning access methods in order to get the read-update-memory tradeoffs right. In this paper, we show through the RUM Conjecture that creating the ultimate access method is infeasible as certain optimization and design choices are mutually exclusive. Instead, we propose a roadmap towards data structures that can be tuned given hardware and application parameters, leading to the new family of *RUM-aware* access methods.

Although building RUM access methods represents a grand new challenge, we see it as the natural next step inspired by our collective past efforts. Past research in areas such as data structures, tuning tools, adaptive processing and indexing, and hardware-conscious database architectures is the initial pool for concepts and principles to be generalized.

Acknowledgements. We thank Margo Seltzer, Niv Dayan, Kenneth Bøgh, the DIAS group at EPFL, and the DASlab group at Harvard for their valuable feedback. This work is partially supported by the Swiss National Science Foundation and by the National Science Foundation under Grant No. IIS-1452595.

7. REFERENCES

[1] D. J. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Found. Trends Databases*, 5(3):197–280, 2013.

[2] A. Aggarwal and J. S. Vitter. The Input/Output Complexity of Sorting and Related Problems. *CACM*, 31(9):1116–1127, 1988.

[3] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. *PVLDB*, 2(1):361–372, 2009.

[4] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A Hands-free Adaptive Store. In *SIGMOD*, 2014.

[5] M. Athanassoulis and A. Ailamaki. BF-Tree: Approximate Tree Indexing. *PVLDB*, 7(14):1881–1892, 2014.

[6] M. Athanassoulis, A. Ailamaki, S. Chen, P. B. Gibbons, and R. Stoica. Flash in a DBMS: Where and How? *IEEE DEBULL*, 33(4):28–34, 2010.

[7] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. MaSM: Efficient Online Updates in Data Warehouses. In *SIGMOD*, 2011.

[8] M. Athanassoulis, S. Chen, A. Ailamaki, P. B. Gibbons, and R. Stoica. Online Updates on Data Warehouses via Judicious Use of Solid-State Storage. *TODS*, 40(1), 2015.

[9] R. Bayer and K. Unterauer. Prefix B-trees. *TODS*, 2(1):11–26, 1977.

[10] 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*, 2007.

[11] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent Cache-Oblivious B-Trees. In *SPAA*, 2005.

[12] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *CACM*, 13(7):422–426, 1970.

[13] N. Bruno and S. Chaudhuri. An Online Approach to Physical Design Tuning. In *ICDE*, 2007.

[14] S. Chaudhuri and V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*, 1997.

[15] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefetching B+-Trees. In *SIGMOD*, 2002.

[16] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. of Algorithms*, 55(1):58–75, 2005.

[17] J. Dittrich and A. Jindal. Towards a One Size Fits All Database Architecture. In *CIDR*, 2011.

[18] P. Francisco. The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. *IBM Redbooks*, 2011.

[19] E. Fredkin. Trie memory. *CACM*, 3(9):490–499, 1960.

[20] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. In *FOCS*, 1999.

[21] G. Graefe. Sorting And Indexing With Partitioned B-Trees. In *CIDR*, 2003.

[22] G. Graefe. Modern B-Tree Techniques. *Found. Trends Databases*, 3(4):203–402, 2011.

[23] G. Graefe, F. Halim, S. Idreos, H. Kuno, and S. Manegold. Concurrency control for adaptive indexing. *PVLDB*, 5(7):656–667, 2012.

[24] G. Graefe, F. Halim, S. Idreos, H. A. Kuno, S. Manegold, and B. Seeger. Transactional support for adaptive indexing. *VLDBJ*, 23(2):303–328, 2014.

[25] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, 2010.

[26] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 5(6):502–513, 2012.

[27] J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton. Architecture of a Database System. *Found. Trends Databases*, 1(2):141–259, 2007.

[28] S. Héman, M. Zukowski, and N. J. Nes. Positional update handling in column stores. In *SIGMOD*, 2010.

[29] J. Hughes. Revolutions in Storage. In *IEEE Conference on Massive Data Storage*, Long Beach, CA, may 2013.

[30] IBM. Storage Class Memory: Towards a disruptively low-cost solid-state non-volatile memory. *IBM Almaden Research Center*, 2013.

[31] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *CIDR*, 2007.

[32] S. Idreos, M. L. Kersten, and S. Manegold. Updating a cracked database. In *SIGMOD*, 2007.

[33] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, 2009.

[34] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 4(9):586–597, 2011.

[35] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental Organization for Data Recording and Warehousing. In *VLDB*, 1997.

[36] B. C. Kuszmaul. A Comparison of Fractal Trees to Log-Structured Merge (LSM) Trees. *Tokutek White Paper*, 2014.

[37] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*, 2013.

[38] Y. Li, B. He, J. Yang, Q. Luo, K. Yi, and R. J. Yang. Tree Indexing on Solid State Drives. *PVLDB*, 3(1-2):1195–1206, 2010.

[39] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *SOSP*, 2011.

[40] W. Litwin and D. B. Lomet. The Bounded Disorder Access Method. In *ICDE*, 1986.

[41] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, 2012.

[42] G. Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*, 1998.

[43] P. E. O'Neil. The SB-Tree: An Index-Sequential Structure for High-Performance Sequential Access. *Acta Informatica*, 29(3):241–265, 1992.

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

[45] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *CACM*, 33(6):668–676, 1990.

[46] R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. *VLDBJ*, 12(2):89–101, 2003.

[47] J. Rao and K. A. Ross. Making B+- trees cache conscious in main memory. In *SIGMOD*, 2000.

[48] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The Uncracked Pieces in Database Cracking. *PVLDB*, 7(2):97–108, 2013.

[49] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[50] L. Sidirourgos and M. L. Kersten. Column Imprints: A Secondary Index Structure. In *SIGMOD*, 2013.

[51] K. Wu, et al. FastBit: interactively searching massive data. *Journal of Physics: Conference Series*, 180(1):012053, 2009.