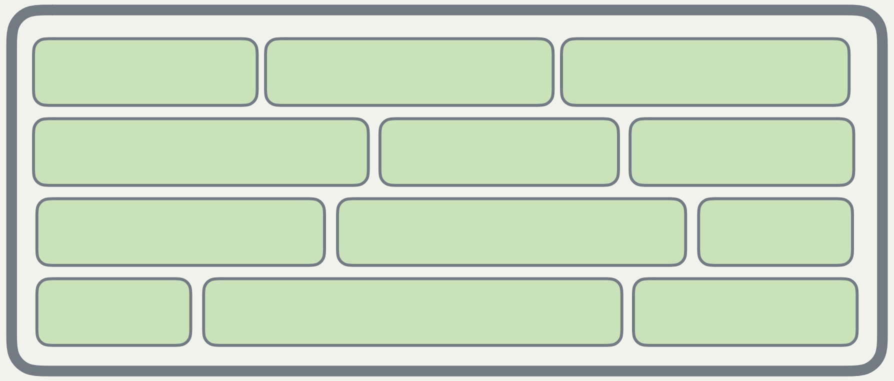
What we talk about when we talk about **On Disk Storage**

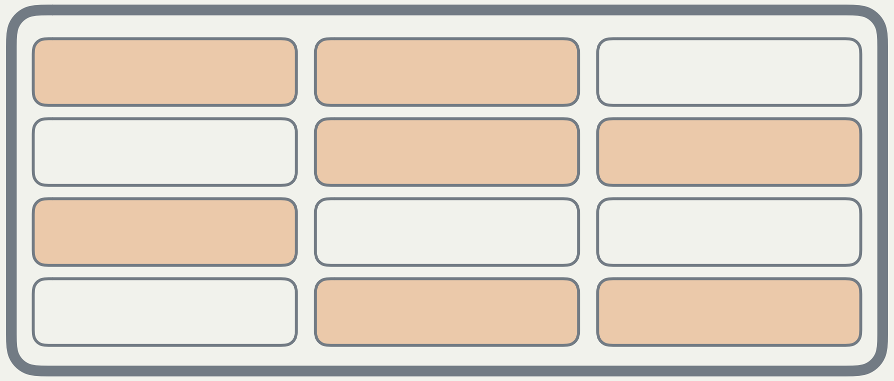
Why this talk?

Access Patterns

Sequential



Random



On Disk Data Structures

Mutable

- Space allocation
- In-place updates
- Fragmentation

Immutable

- Sequential writes
- Multiple read sources
- Needs merge

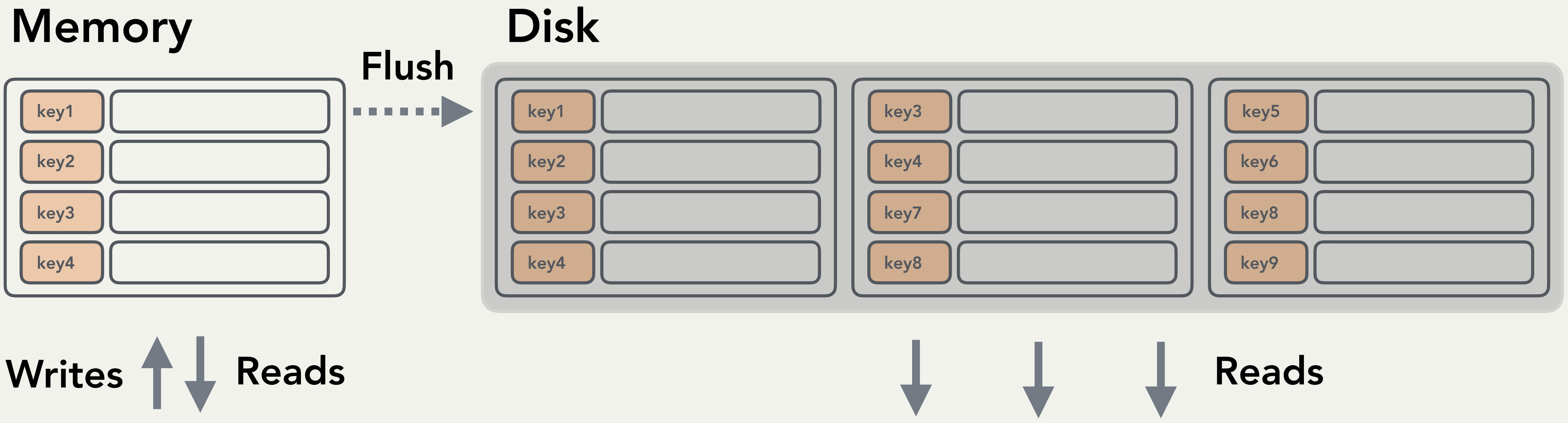
The Log-Structured Merge-Tree (LSM-Tree)

Patrick O'Neil[1], Edward Cheng[2]
Dieter Gawlick[3], Elizabeth O'Neil[1]
To be published: Acta Informatica

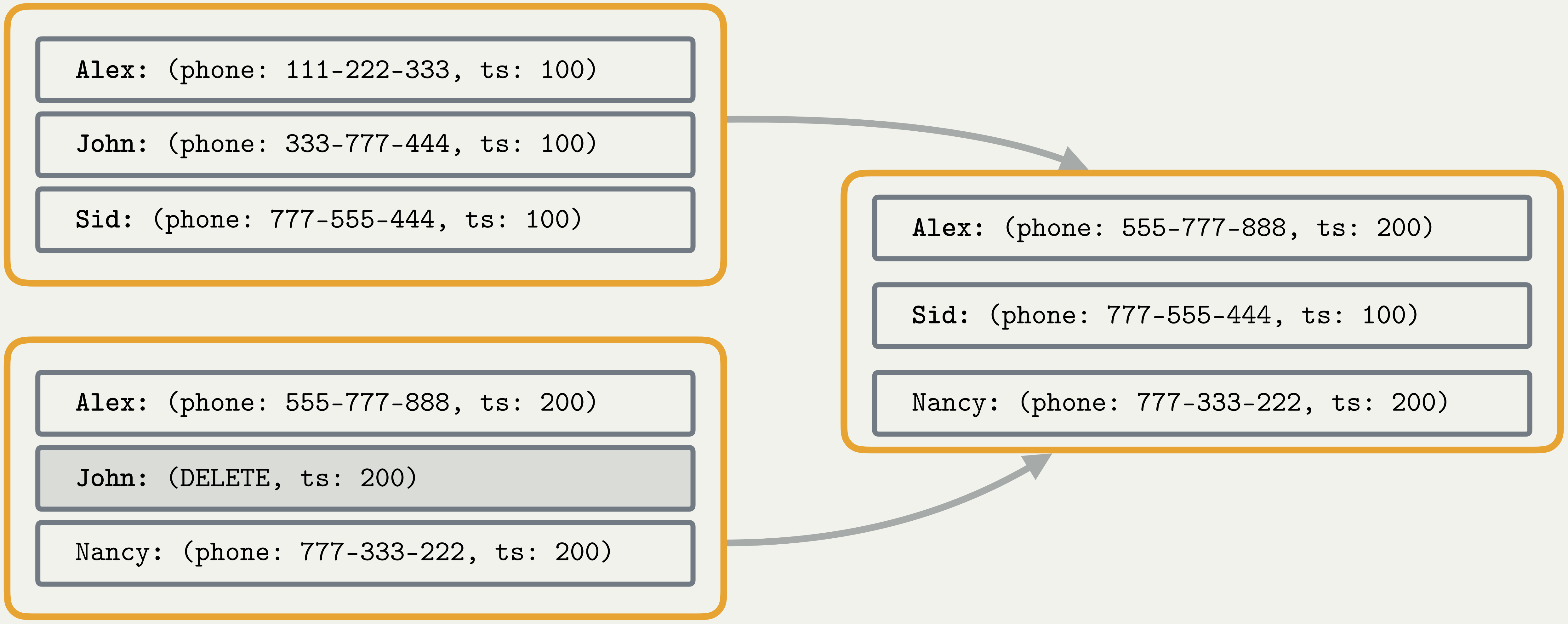
ABSTRACT. High-performance transaction system applications typically insert rows in a History table to provide an activity trace; at the same time the transaction system generates log records for purposes of system recovery. Both types of generated information can benefit from efficient indexing. An example in a well-known setting is the TPC-A benchmark application, modified to support efficient queries on the History for account activity for specific accounts. This requires an index by account-id on the fast-growing History table. Unfortunately, standard disk-based index structures such as the B-tree will effectively double the I/O cost of the transaction to maintain an index such as this in real time, increasing the total system cost up to fifty percent. Clearly a method for maintaining a real-time index at low cost is desirable. The Log-Structured Merge-tree (LSM-tree) is a disk-based data structure designed to provide low-cost indexing for a file experiencing a high rate of record inserts (and deletes) over an extended period. The LSM-tree uses an algorithm that defers and batches index changes, cascading the changes from a memory-based component through one or more disk components in an efficient manner reminiscent of merge sort. During this process all index values are continuously accessible to retrievals (aside from very short locking periods), either through the memory component or one of the disk components. The algorithm has greatly reduced disk arm movements compared to a traditional access methods such as B-trees, and will improve cost-performance in domains where disk arm costs for inserts with traditional access methods overwhelm storage media costs. The LSM-tree approach also generalizes to operations other than insert and delete. However, indexed finds requiring immediate response will lose I/O efficiency in some cases, so the LSM-tree is most useful in applications where index inserts are more common than finds that retrieve the entries. This seems to be a common property for History tables and log files, for example. The conclusions of Section 6 compare the hybrid use of memory and disk components in the LSM-tree access method with the commonly understood advantage of the hybrid method to buffer disk pages in memory.

1. Introduction

As long-lived transactions in activity flow management systems become commercially available ([10], [11], [12], [20], [24], [27]), there will be increased need to provide indexed access to transactional log records. Traditionally, transactional logging has focused on aborts and recovery, and has required the system to refer back to a relatively short-term history in normal


Memory
Disk
Flush
key1
key2
key3
key4
key1
key2
key3
key4
key3
key4
key7
key8
key5
key6
key8
key9
Writes
Reads
Reads

Merge Process



Alex: (phone: 111-222-333, ts: 100)
John: (phone: 333-777-444, ts: 100)
Sid: (phone: 777-555-444, ts: 100)
Alex: (phone: 555-777-888, ts: 200)
John: (DELETE, ts: 200)
Nancy: (phone: 777-333-222, ts: 200)
Alex: (phone: 555-777-888, ts: 200)
Sid: (phone: 777-555-444, ts: 100)
Nancy: (phone: 777-333-222, ts: 200)

Merge Arithmetics

$\forall \; Y > X$

Updates

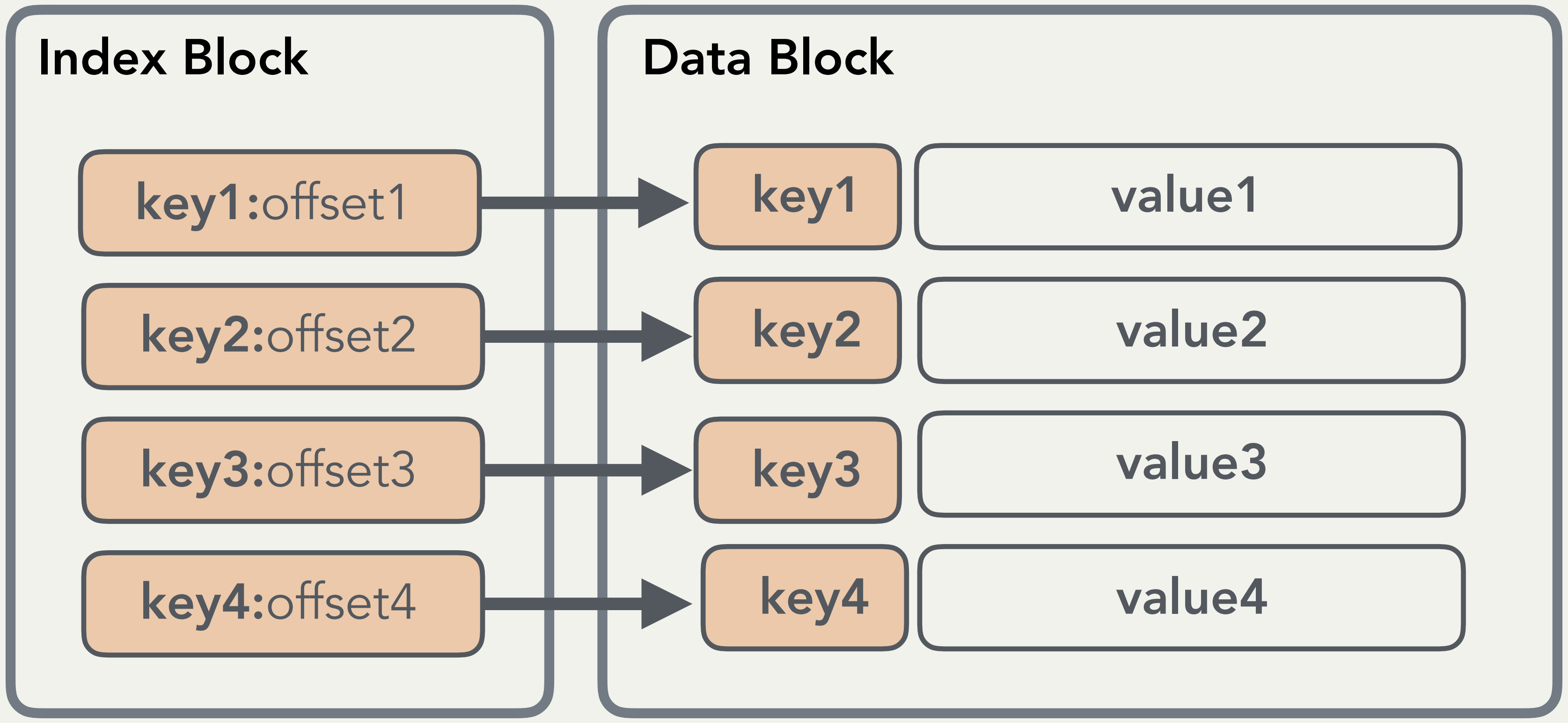
$\boxed{\texttt{Key: Value1; Timestamp: } Y} $

$$\boxed{\texttt{Key: Value1; Timestamp: } X} + \boxed{\texttt{Key: Value1; Timestamp: } Y} = \boxed{\texttt{Key: Value1; Timestamp: } Y}$$

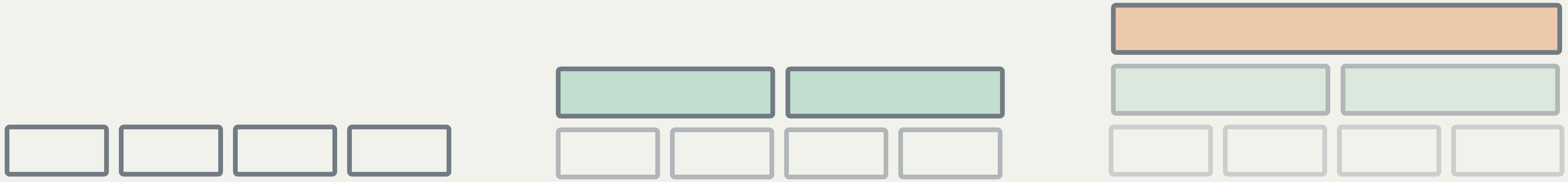
Deletes

$$\boxed{\texttt{Key: Value1; Timestamp: } X} + \boxed{\texttt{Key: -(Value1); Timestamp: } Y} = \varnothing$$

Sorted String Tables



Compaction



Summary

- Immutable
- Write-optimised
- Read multiplexing
- High maintenance costs
- Well-suited for concurrent ops
- Simple to implement

🐝🌲

The Ubiquitous B-Tree

DOUGLAS COMER

Computer Science Department, Purdue University, West Lafayette, Indiana 47907

B-trees have become, de facto, a standard for file organization. File indexes of users, dedicated database systems, and general-purpose access methods have all been proposed and implemented using B-trees This paper reviews B-trees and shows why they have been so successful It discusses the major variations of the B-tree, especially the B$^+$-tree, contrasting the relative merits and costs of each implementation. It illustrates a general purpose access method which uses a B-tree.

Keywords and Phrases: B-tree, B*-tree, B$^+$-tree, file organization, index

CR Categories: 3.73 3.74 4.33 4 34

INTRODUCTION

The secondary storage facilities available on large computer systems allow users to store, update, and recall data from large collections of information called files. A computer must retrieve an item and place it in main memory before it can be processed. In order to make good use of the

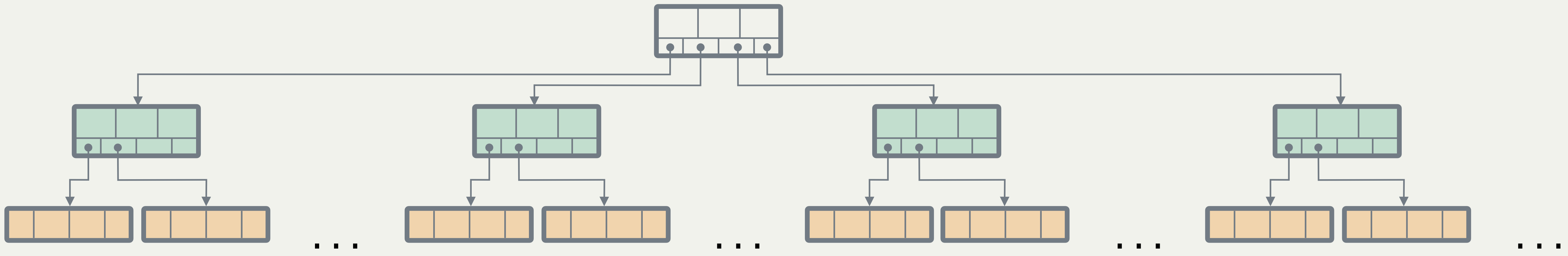
might be labeled with the employees' last names. A sequential request requires the searcher to examine the entire file, one folder at a time. On the other hand, a random request implies that the searcher, guided by the labels on the drawers and folders, need only extract one folder.

Associated with a large, randomly accessed file in a computer system is an *index*

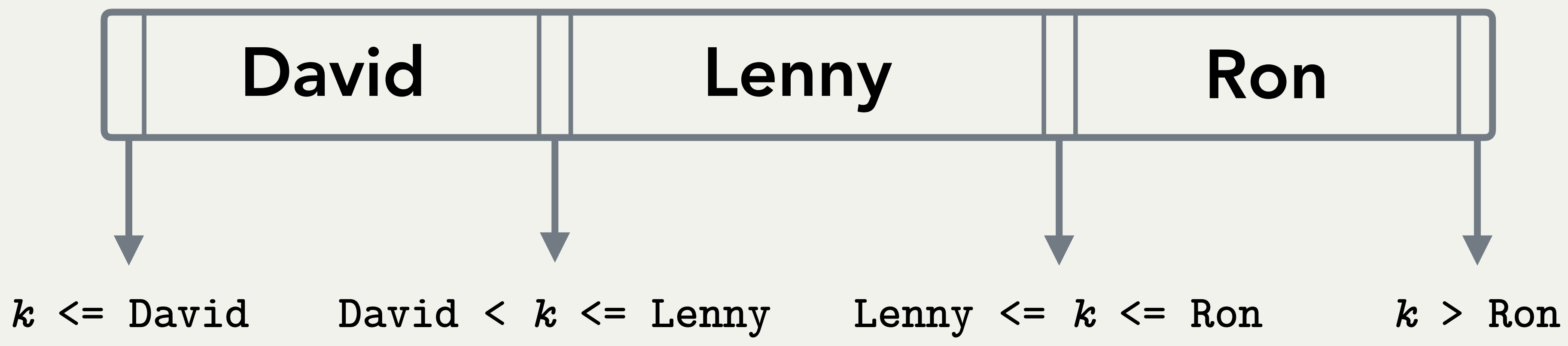
B-Trees

- Ordered data structure
- Often used for indexing
- Usually implemented as mutable DS
- Self Balancing

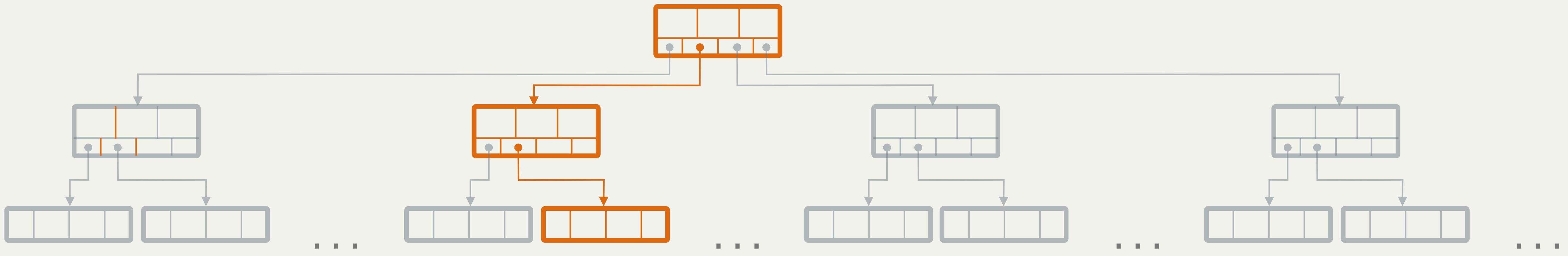
Anatomy of B-Tree



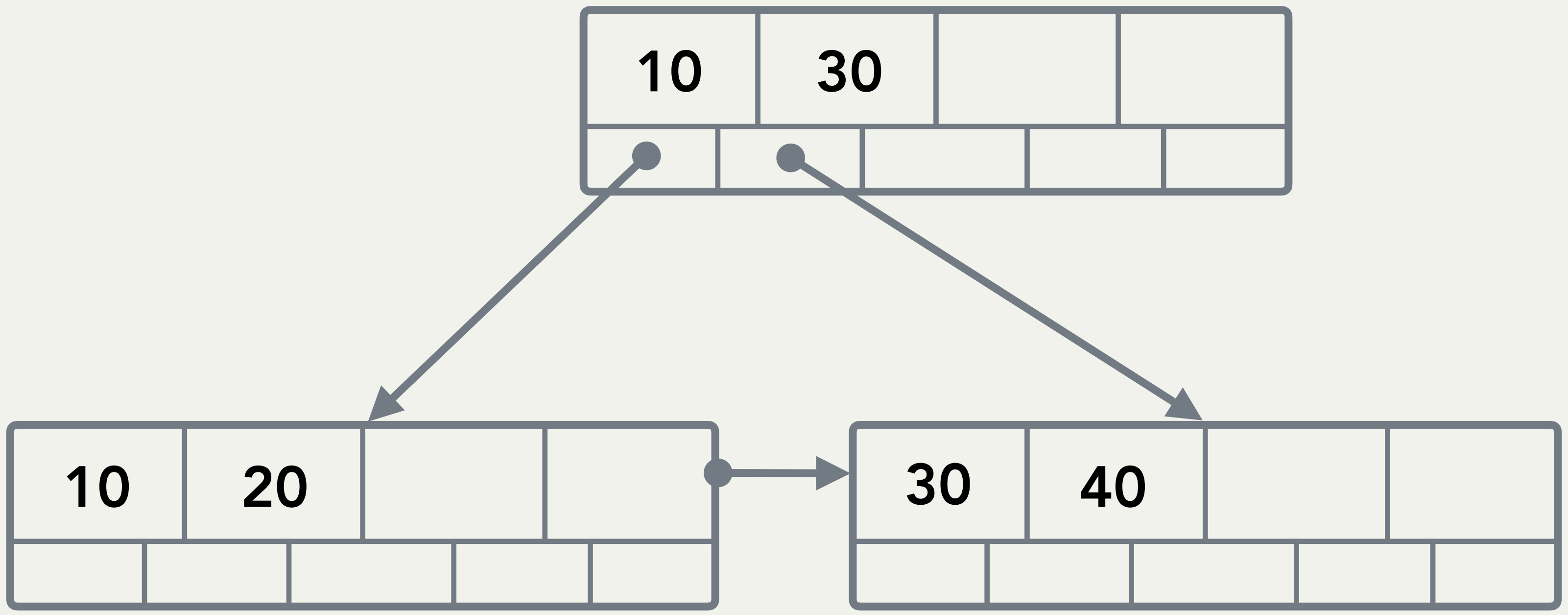
Invariants



Search

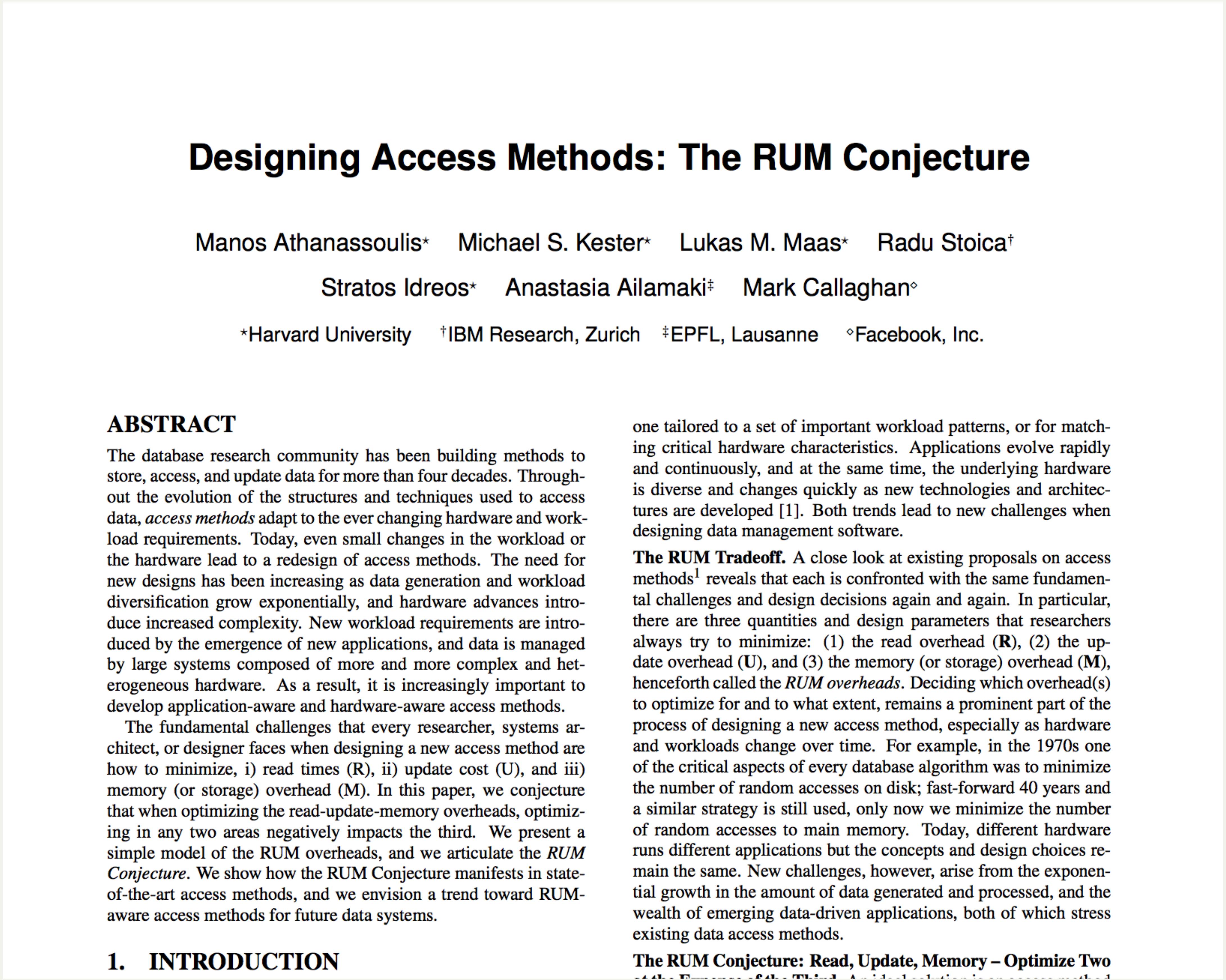


Splits and Merges



Summary

- Mutable
- Read-optimised
- Requires splits/merges/rebalancing
- Block storage optimised
- Overhead for in-place updates



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.

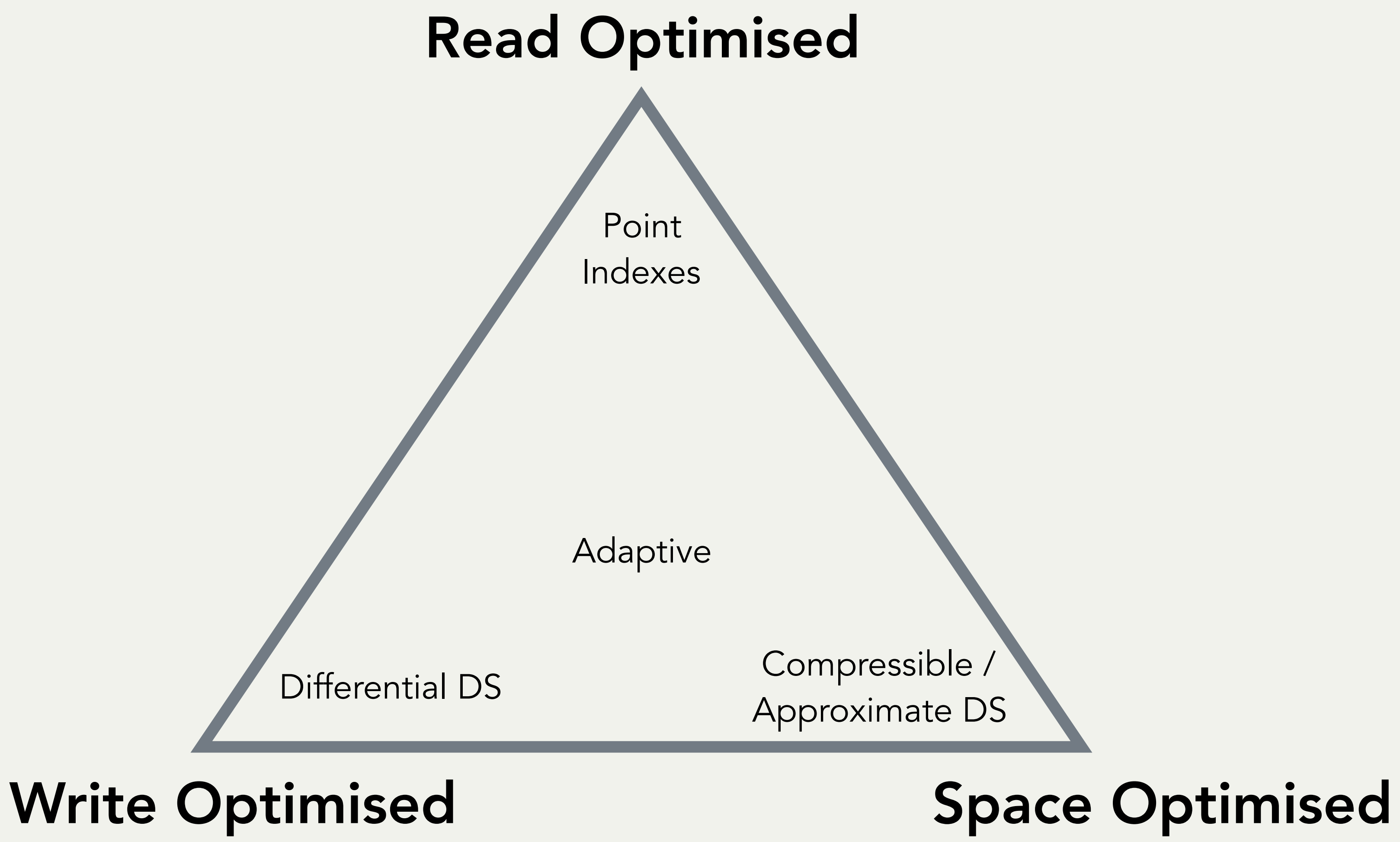
The fundamental challenges that every researcher, systems architect, or designer faces when designing a new access method are how to minimize, i) read times (R), ii) update cost (U), and iii) memory (or storage) overhead (M). In this paper, we conjecture that when optimizing the read-update-memory overheads, 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.

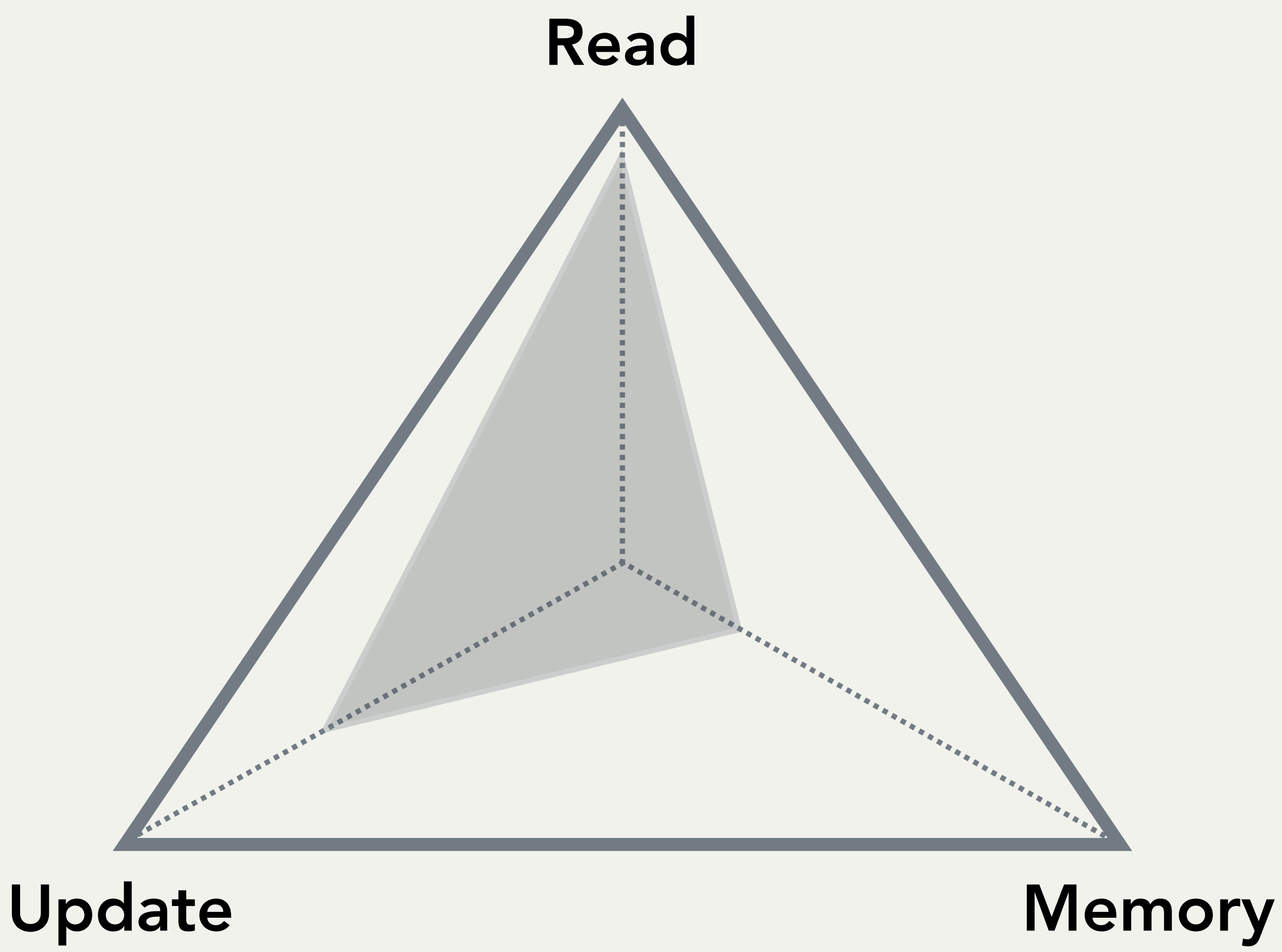
1. INTRODUCTION

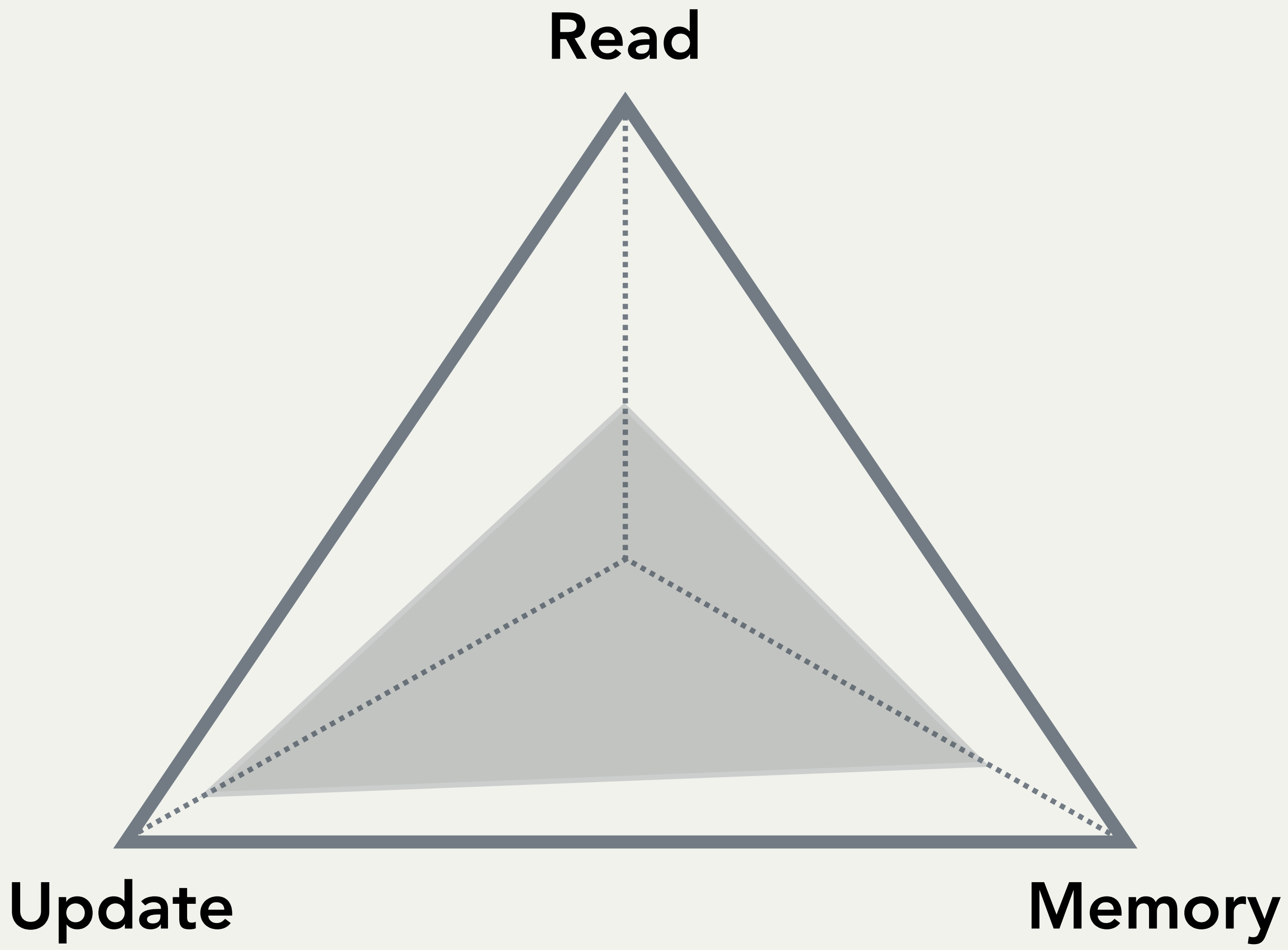
one tailored to a set of important workload patterns, or for matching critical hardware characteristics. Applications evolve rapidly and continuously, and at the same time, the underlying hardware is diverse and changes quickly as new technologies and architectures are developed [1]. Both trends lead 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.

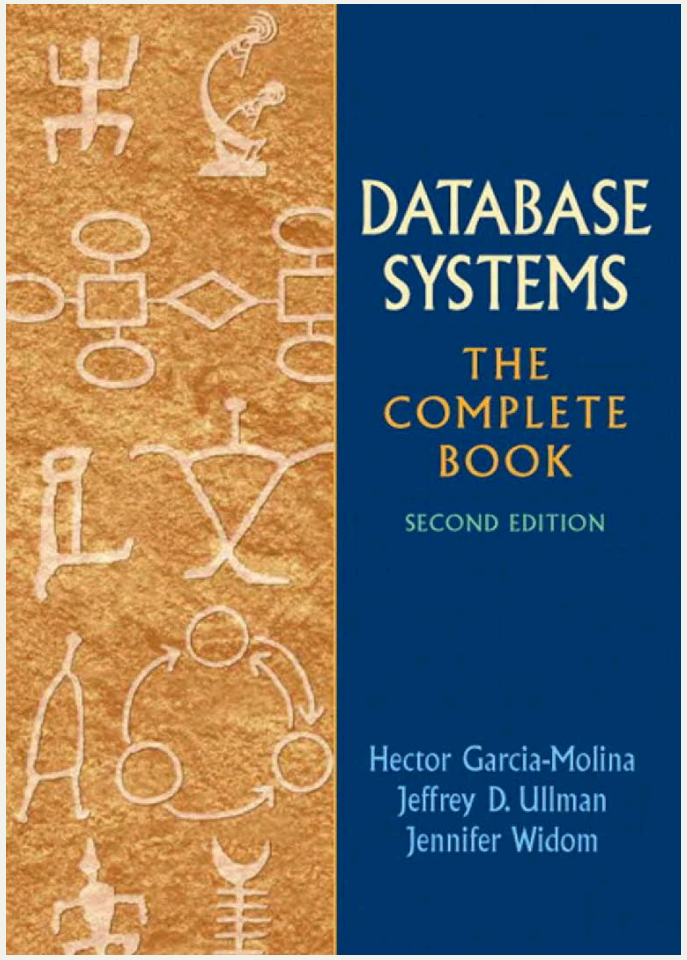
The RUM Conjecture: Read, Update, Memory – Optimize Two

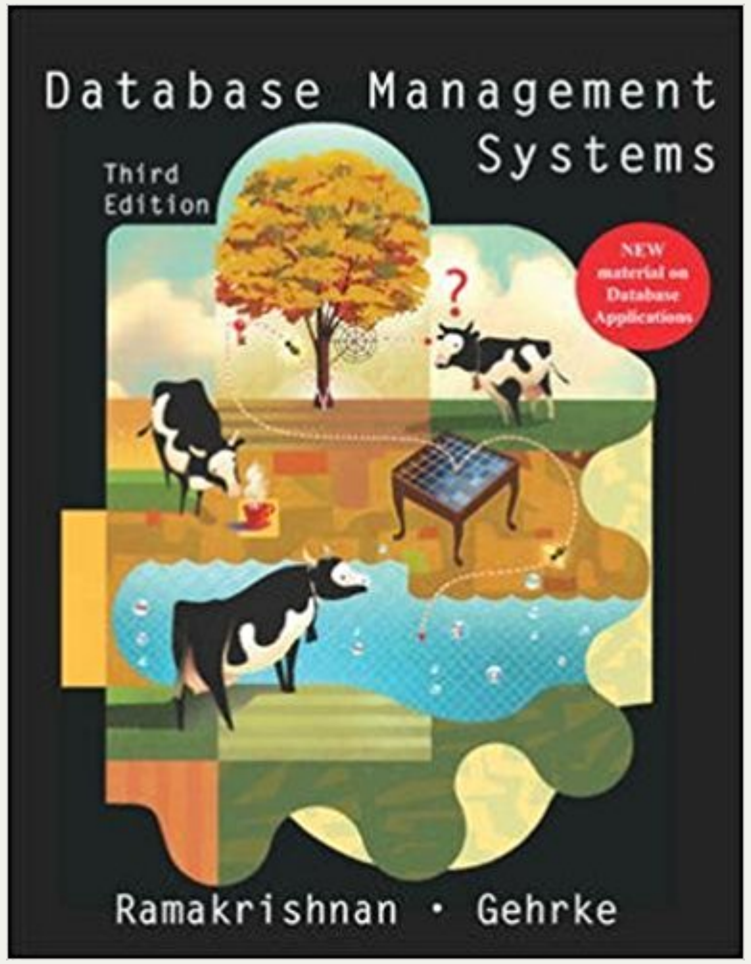

Read Optimised
Point Indexes
Adaptive
Differential DS
Compressible / Approximate DS
Write Optimised
Space Optimised

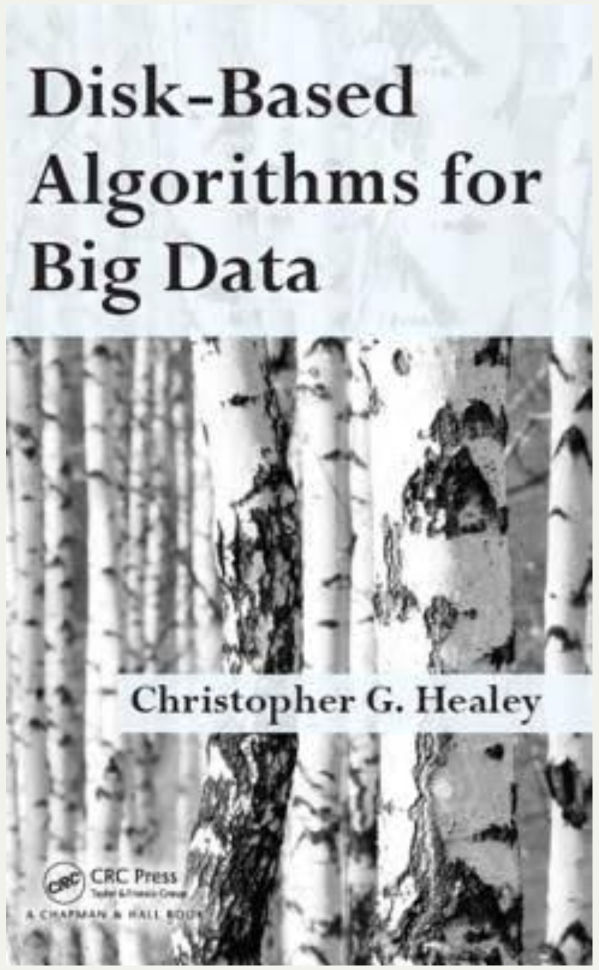

Read
Update
Memory

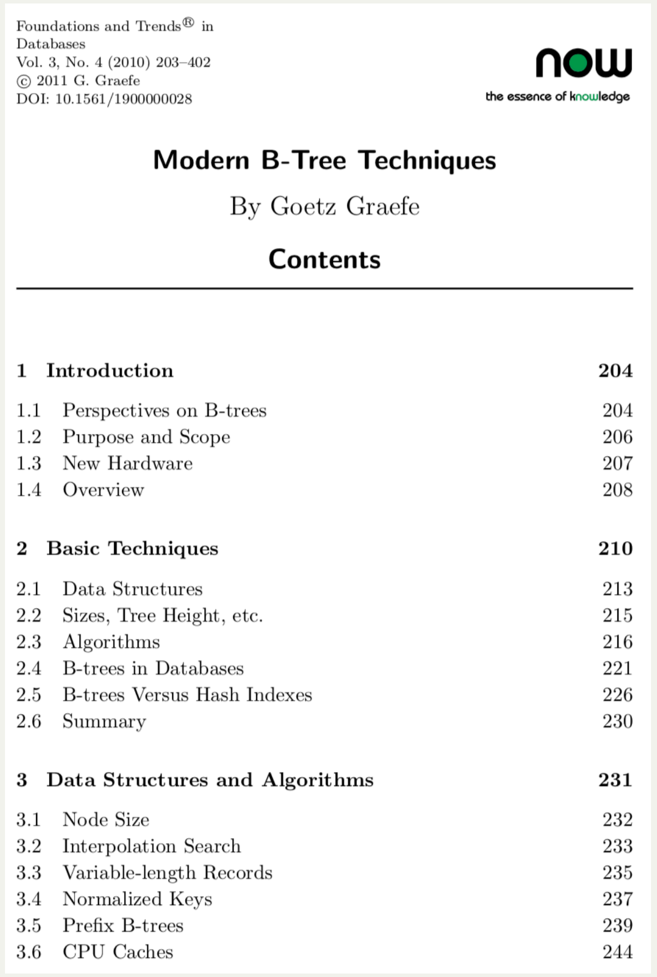

Read
Update
Memory

Wrapping Up




Database Management Systems
Third Edition
NEW material on Database Applications
Ramakrishnan • Gehrke




Foundations and Trends® in
Databases
Vol. 3, No. 4 (2010) 203–402
© 2011 G. Graefe
DOI: 10.1561/1900000028

now
the essence of knowledge

Modern B-Tree Techniques

By Goetz Graefe

Contents

@ifesdjeen