

Efficient Processing of Range Queries in Main Memory

DISSERTATION

zur Erlangung des akademischen Grades

Dr. rer. nat.

im Fach Informatik

eingereicht an der

Mathematisch-Naturwissenschaftlichen Fakultät
der Humboldt-Universität zu Berlin

von

M.Sc. Stefan Sprenger

Präsidentin der Humboldt-Universität zu Berlin:

Prof. Dr.-Ing. Dr. Sabine Kunst

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät:

Prof. Dr. Elmar Kulke

Gutachter:

1. Prof. Dr.-Ing. Ulf Leser
2. Prof. Dr. Odej Kao
3. Prof. Dr.-Ing. Kai-Uwe Sattler

Tag der mündlichen Prüfung: 15. Februar 2019

Abstract

Over the last two decades, the hardware landscape has fundamentally changed, shaping the architecture of modern database systems. First, main memory becomes a popular choice for primary data storage, since current server machines can feature up to several terabytes of main memory. Second, modern CPUs get tremendously parallel providing both data- and task-level parallelism. They support SIMD instructions, which can simultaneously process multiple data elements. Moreover, stand-alone CPUs host an ever increasing number of cores, which can be considered as independent processing units. Recent many-core CPUs install up to 72 cores on one chip.

Database systems employ index structures as means to accelerate search queries. Over the last years, the research community has proposed many different in-memory approaches that optimize cache misses instead of disk I/O, as opposed to disk-based systems, and make use of the grown parallel capabilities of modern CPUs. However, these techniques mainly focus on single-key lookups, but neglect equally important range queries. Range queries are an ubiquitous operator in data management commonly used in numerous domains, such as genomic analysis, sensor networks, or online analytical processing.

The main goal of this dissertation is thus to improve the capabilities of main-memory database systems with regard to executing range queries. To this end, we first propose a cache-optimized, updateable main-memory index structure, the cache-sensitive skip list, which targets the execution of range queries on single database columns. Second, we study the performance of multidimensional range queries on modern hardware, where data are stored in main memory and processors support SIMD instructions and multi-threading. We re-evaluate a previous rule of thumb suggesting that, on disk-based systems, scans outperform index structures for selectivities of approximately 15-20% or more. To increase the practical relevance of our analysis, we also contribute a novel benchmark consisting of several realistic multidimensional range queries applied to real-world genomic data. Third, based on the outcomes of our experimental analysis, we devise a novel, fast and space-efficient, main-memory based index structure, the BB-Tree, which supports multidimensional range and point queries and provides a parallel search operator that leverages the multi-threading capabilities of modern CPUs.

Zusammenfassung

Innerhalb der letzten zwanzig Jahre haben sich die Hardwarekomponenten von Serversystemen stark weiterentwickelt, wodurch die Architektur von modernen Datenbanksystemen entscheidend geprägt wurde. Zum einen wird der Hauptspeicher als primärer Speicherort für Datenbanksysteme verwendet. Aktuelle Serversysteme sind mit bis zu einigen Terabytes an Hauptspeicher ausgestattet, was durch steigende Kapazitäten und fallende Preise ermöglicht wird. Zum anderen sind aktuelle Prozessoren höchst parallel. Sie unterstützen datenparallele SIMD-Instruktionen, mit denen mehrere Werte mit einer Instruktion verarbeitet werden können, und installieren eine immer größere Anzahl an Prozessorkernen auf einen Chip.

Datenbanksysteme verwenden Indexstrukturen, um Suchanfragen zu beschleunigen. Im Laufe der letzten Jahre haben Forscher verschiedene Ansätze zur Indexierung von Datenbanktabellen im Hauptspeicher entworfen. Hauptspeicherindexstrukturen versuchen möglichst häufig Daten zu verwenden, die bereits im Zwischenspeicher der CPU vorrätig sind, anstatt, wie bei traditionellen Datenbanksystemen, die Zugriffe auf den externen Speicher zu optimieren. Die meisten vorgeschlagenen Indexstrukturen für den Hauptspeicher beschränken sich jedoch auf Punktabfragen und vernachlässigen die ebenso wichtigen Bereichsabfragen, die in zahlreichen Anwendungen, wie in der Analyse von Genomdaten, Sensornetzwerken, oder analytischen Datenbanksystemen, zum Einsatz kommen.

Diese Dissertation verfolgt als Hauptziel die Fähigkeiten von modernen Hauptspeicherdatenbanksystemen im Ausführen von Bereichsabfragen zu verbessern. Dazu schlagen wir zunächst die Cache-Sensitive Skip List, eine neue aktualisierbare Hauptspeicherindexstruktur, vor, die für die Zwischenspeicher moderner Prozessoren optimiert ist und das Ausführen von Bereichsabfragen auf einzelnen Datenbankspalten ermöglicht. Im zweiten Abschnitt analysieren wir die Performanz von multidimensionalen Bereichsabfragen auf modernen Serverarchitekturen, bei denen Daten im Hauptspeicher hinterlegt sind und Prozessoren über SIMD-Instruktionen und Multithreading verfügen. Um die Relevanz unserer Experimente für praktische Anwendungen zu erhöhen, schlagen wir zudem einen realistischen Benchmark für multidimensionale Bereichsabfragen vor, der auf echten Genomdaten ausgeführt wird. Im letzten Abschnitt der Dissertation präsentieren wir den BB-Tree als neue, hochperformante und speichereffiziente Hauptspeicherindexstruktur. Der BB-Tree ermöglicht das Ausführen von multidimensionalen Bereichs- und Punktabfragen und verfügt über einen parallelen Suchoperator, der mehrere Threads verwenden kann, um die Performanz von Suchanfragen zu erhöhen.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Problem Definition and Contributions	3
1.3	Outline	5
1.4	Prior Publications	6
2	Fundamentals	7
2.1	Terminology	7
2.2	Multidimensional Access Methods	8
2.2.1	Sequential Scan	9
2.2.2	Point Access Methods (PAM)	11
2.2.3	Spatial Access Methods (SAM)	17
2.3	Index Structures on Modern Hardware	23
2.3.1	Modern Memory Hierarchy	23
2.3.2	Single Instruction Multiple Data (SIMD)	25
2.3.3	Multi-Core CPUs and Simultaneous Multithreading (SMT)	26
2.4	Genomic Multidimensional Range Query Benchmark (GMRQB)	28
2.4.1	Range Queries on Genomic Variant Data	29
2.4.2	Real-World Data Set	30
2.4.3	Realistic Range Query Templates	33
3	CSSL: Processing One-Dimensional Range Queries in Main Memory	35
3.1	Related Work	36
3.2	Conventional Skip Lists	37
3.3	Cache-Sensitive Skip Lists (CSSL)	40
3.3.1	Memory Layout	40
3.3.2	Search Algorithms and Updates	42
3.4	Evaluation	47
3.4.1	Experimental Setup	47
3.4.2	Experimental Data and Workloads	48
3.4.3	Range Queries	49
3.4.4	Lookups	52
3.4.5	Mixed Workloads	53
3.4.6	Space Consumption	53
3.5	Discussion	54
3.6	Summary	55

4	An Analysis of Multidimensional Range Queries on Modern Hardware	57
4.1	Partitioning for Parallelization	58
4.2	Vectorizing Range Queries	61
4.3	Conservative Adaptation of Multidimensional Index Structures	63
4.4	Evaluation	66
4.4.1	Experimental Setup	67
4.4.2	Experimental Data and Workloads	68
4.4.3	Impact of Multithreading and Vectorization	69
4.4.4	Synthetic Data	71
4.4.5	Sensor Data from Hi-Tech Manufacturing Equipment	75
4.4.6	Genomic Variant Data	75
4.4.7	Scalability	76
4.4.8	Space Consumption	77
4.4.9	Other Evaluation Platform	78
4.5	Discussion	79
4.6	Summary	82
5	BB-Trees: Processing Multidimensional Range Queries in Main Memory	83
5.1	Data Organization	85
5.2	Bubble Buckets	90
5.3	Building and Reorganizing BB-Trees	92
5.4	Low-Cardinality Dimensions	94
5.5	Search Algorithms	94
5.6	Parallel BB-Trees	96
5.7	Evaluation	98
5.7.1	Experimental Setup	98
5.7.2	Experimental Data and Workloads	99
5.7.3	Impact of Bubble Bucket Capacities	101
5.7.4	Point Queries	101
5.7.5	Range Queries	102
5.7.6	Impact of Dimensionality	106
5.7.7	Low-Cardinality Dimensions	107
5.7.8	Insertions and Deletions	108
5.7.9	Mixed Read/Write Workloads	109
5.7.10	Scalability	111
5.7.11	Space Consumption	113
5.8	Discussion	114
5.9	Summary	115
6	Summary and Outlook	117
6.1	Summary	117
6.2	Outlook	118
	Appendix A: Genomic Multidimensional Range Query Benchmark	123

1 Introduction

1.1 Motivation

Since over forty years, a plethora of different applications and domains uses database systems [Codd 1970] to store and query very large amounts of data. For historical reasons, they were originally designed for server machines equipped with single-core CPUs and large disks holding all data. Main memories were solely used for buffering, because they were too small to store complete data sets.

Over the last two decades, technological improvements enabled powerful changes in the hardware landscape, shaping the architecture of modern database systems [Faerber et al. 2017; Manegold et al. 2000; Boncz et al. 1999; Saecker et al. 2012]:

- Main memory becomes the number one choice for data storage. Today, single machines can feature up to dozens of terabytes of main memory. Concurrently, cost are sharply dropping: In 2017, the price for one megabyte of main memory fell below \$0.01¹. As a consequence, many databases can be completely kept in main memory. Even data processing tasks previously distributed across large clusters to accelerate analysis of on-disk data, can now be processed in the main memory of one machine [Rowstron et al. 2012].
- Modern CPUs implement the *Single Instruction Multiple Data* (SIMD) execution model [Flynn 1972], which allows to process multiple data elements with one instruction, enabling data-level parallelism. They provide dedicated instruction sets, often available as intrinsic functions, to execute vector operations on extra-wide registers.
- CPUs become tremendously parallel featuring a consistently increasing number of cores, which resemble independent processing units with distinct instruction pipelines and caches. As of today, many-core CPUs host more than seventy cores on one chip². Moreover, simultaneous multithreading (SMT) increases the degree of parallelism by deploying multiple virtual threads (typically two or four) onto one core. When combining SIMD with multi-core parallelism and SMT, modern CPUs achieve degrees of parallelism competitive with highly-parallel GPUs [Sprenger et al. 2018c].

As a consequence, over the last years, completely new competitors targeting deployments on modern hardware entered the database market, e. g., MemSQL [Chen et al. 2016], Redis [Carlson 2013], or VoltDB [Stonebraker et al. 2013], but also established database vendors adapted to the

¹Memory Prices 1957 to 2017, <https://jcmit.net/memoryprice.htm>, Last access: August 29, 2018.

²Intel®Xeon Phi™x200 Product Family Product Specifications, <https://ark.intel.com/products/series/92650/Intel-Xeon-Phi-x200-Product-Family>, Last access: August 29, 2018.

1 Introduction

recent hardware trends, especially an in-memory data storage, e.g., Oracle TimesTen [Lahiri et al. 2013], Microsoft SQL Server through Hekaton [Diaconu et al. 2013], or IBM DB2 with BLU acceleration [Raman et al. 2013].

Users often query databases with workloads boiling down to either point or range queries [Hellerstein et al. 1995]. While point queries retrieve a single tuple from a database table, typically to confirm that a certain value exists, range queries specify selection predicates on one or several columns and return subsets, which usually span multiple tuples. Range queries are an essential part of many database workloads and widely used in numerous applications, such as:

- **Genomics:** Precision medicine analyzes the mutational landscape of entire populations, subpopulations or disease cohorts to adjust the treatment of individual patients based on known associations between the occurrences of certain genomic variants and the impact of certain drugs [Lievre et al. 2006]. Genomic variants are described by a large number of features (typically between fifteen and twenty), e.g., genomic location, phenotype, or metadata of the person where the variant has been found.

Researchers load genomic variant data into database systems to search for variants of interest with range predicates [Hakenberg et al. 2016]. For instance, a query may ask for all variants found in the coding region of a certain gene of patients of a certain age group; genes are defined by ranges of genomic locations. Another example application are visualization tools, like genome browsers [Thorvaldsdóttir et al. 2013], which allow an interactive navigation of complete (human) genomes. Users can visually scroll over individual mutation profiles organized by their genomic location. As genomes are very large, genome browsers apply range queries to narrow down the vast amount of data and retrieve the subset relevant for the current location window.

- **Internet of Things:** Internet of things (IoT) resembles networks of interconnected devices and vehicles equipped with sensors (or actuators) exchanging data with each other [Gubbi et al. 2013]. As of today, IoT is employed in a wide range of different domains in everyday life, such as: (1) Continuous glucose monitoring devices automatically suspend insulin pumps to prevent hypoglycemia when detecting severe drops in blood glucose [Choudhary et al. 2011]. (2) Wireless power metering plugs (or *smart* home devices) provide insights into the energy consumption of (private) households [Jahn et al. 2010]. (3) *Smart* factories install large networks of sensors to monitor the health of components, facilitating automatic detection of malfunctions [Ji et al. 2013]. Range queries play an important role in the analysis of data collected by sensing devices [X. Li et al. 2003; Wang et al. 2014; Wang et al. 2016]. For instance, search queries may ask for all observations within certain ranges regarding features, like temperature, or light level.
- **Online Analytical Processing:** Online analytical processing (OLAP) targets a fast execution of read-heavy, analytical workloads on historical data [Chaudhuri et al. 1997]. Example applications include (a) periodic reporting for sales, marketing or management departments, (b) forecasting in supply chain management to meet future demands of customers, or (c) budgeting. Typically, OLAP queries are executed on multidimensional arrays, relational database systems or hybrid storages. OLAP workloads often involve

(multidimensional) range queries [Ho et al. 1997; Liang et al. 2000]. For instance, reporting may ask for the sum of all sales within a certain time range, a certain price range and a certain range of customers.

Range queries can be answered by sequential scans over complete data or dedicated index structures. The benefits of scans are a sequential data access pattern, no additional space requirements, and no build-up or maintenance cost. On the other side, index structures can prune the data space, but require random accesses and are rather expensive to maintain. Scans are typically useful for search queries retrieving most parts of a data set (low selectivity), whereas index structures are superior for search queries selecting few tuples (high selectivity). When generating execution plans, database systems employ access path selection to choose between full-table scans and index probing based on the estimated selectivity of the individual query [Selinger et al. 1979]

Over the last years, the research community has proposed many main-memory index structures that leverage the features of modern hardware [Hao Zhang et al. 2015]. They optimize cache misses instead of disk I/O³ and make use of the grown parallel capabilities of modern CPUs, e.g., ART [Leis et al. 2013], an adaptive radix tree, FAST [C. Kim et al. 2010], a read-only search tree aligned to the cache hierarchy, or the Bw-tree [Levandowski et al. 2013], a latch-free approach targeting concurrent query processing. However, previous works mainly focused on achieving fast single-key lookups, but neglected range queries. Additionally, most of them are restricted to one-dimensional data and cannot be applied to multidimensional domains.

Consequently, current main-memory database systems primarily rely on full-table scans to evaluate range queries [Das et al. 2015], although scans are inadequate for highly selective workloads, because they lack pruning capabilities. Thus, there is a strong need for cache-optimized index structures that provide an efficient range query operator tailored to the properties of modern hardware and that are able to handle multidimensional workloads.

1.2 Problem Definition and Contributions

As main goal, this thesis aims to improve the capabilities of main-memory database systems [Plattner et al. 2011] with regard to executing (multidimensional) range queries. Many different in-memory index structures have been proposed over the last years, yet none of them supports range queries efficiently, because they require many random accesses to evaluate range queries. In main memory, sequential access patterns are as important for search performance as in disk-based systems. Reading data from consecutive memory locations utilizes prefetched cache lines, since modern CPUs employ *one block lookahead* [Smith 1982] prefetching, and effectively reduces CPU cache and *translation lookaside buffer* (TLB) misses [Manegold et al. 2000].

We start with studying the processing of range queries on single database columns and subsequently delve into the more complex multidimensional domain, where range queries consist of multiple predicates specified on some, many or all dimensions of the data space. As main results,

³For in-memory database systems, Ailamaki et al. [Ailamaki et al. 1999] showed that fifty percent of workload execution times are spent in stalls caused by cache misses.

1 Introduction

this thesis contributes novel, cache-optimized index structures to accelerate (multidimensional) range queries in main-memory settings.

In particular, we make the following contributions.

Processing One-Dimensional Range Queries in Main Memory

Many index structures store data in a sorted order, which allows straightforward implementations of range queries: First search for the smallest matching element, and then process all consecutive elements until the first mismatch occurs. With decreasing query selectivity and increasing data set cardinality, range queries spend most of their execution time on the second step, i. e., collecting matching data. Unfortunately, existing main-memory index structures mainly optimize single-key lookups, the initial action of a range query, but neglect the second, more time-consuming step. Typically, processing matching elements requires navigating tree structures by chasing pointers via random accesses, which produces many CPU cache and TLB misses, requiring data to be loaded from main memory into on-die caches, and ultimately leads to a poor range query performance.

We devise and implement a novel main-memory index structure that employs a CPU-friendly data layout enabling the range query operator to traverse over matching elements with an almost-sequential data access pattern. We exploit SIMD instructions to further speed-up range queries through data-level parallelism. In a comprehensive evaluation, we compare our approach to multiple state-of-the-art main-memory index structures using different query workloads over synthetic and real-world data sets.

Analysis of Multidimensional Range Queries on Modern Hardware

More than twenty years ago, Weber et al. [Weber et al. 1998] showed how the efficiency of multidimensional access methods depends on the selectivity of queries. Their research was conducted in the context of disk-based systems and was driven by the observation that random accesses, as operated by index structures, are approximately five times slower than sequential reads. They reported a selectivity threshold of approximately 15-20% after which sequential scans outperform multidimensional index structures. Since then, this rule of thumb has been used to select access paths when evaluating multidimensional range queries, completely ignoring the advancements in hardware.

In an experimental analysis, we study the strengths and weaknesses of multidimensional access methods when deployed on modern hardware, and evaluate whether the old selectivity threshold of 15-20% still holds or should be updated. We first propose multiple techniques to adapt existing multidimensional index structures to modern hardware. Here, we focus on main-memory storage, SIMD instructions, and multi-core CPUs. We apply the adaptation techniques to the R*-tree, the kd-tree, the VA-file, and a sequential scan. Furthermore, we design a realistic multidimensional range query benchmark executed on real-world data. In an extensive evaluation, we investigate the approaches when executing synthetic and realistic multidimensional range query workloads over synthetic and real-world data sets. We study the impact of various data and workload characteristics, e. g., dimensionality, cardinality, and selectivity.

Processing Multidimensional Range Queries in Main Memory

Most existing multidimensional index structures were originally designed for disk-based systems and do not take the memory hierarchy of modern server machines into account. As a consequence, they show a poor cache efficiency when being stored in main memory. Moreover, their search algorithms were built for single-core processors, neglecting the enormous parallel capabilities of modern CPUs. Thus, most existing multidimensional index structures do not take advantage of the features of modern hardware.

We propose and implement a novel general-purpose index structure that can efficiently process multidimensional range and point query workloads on data stored in main memory. It employs a cache-optimized data layout that enables an outstanding search efficiency on modern CPUs. We also provide a parallel implementation of the range query operator, which utilizes multiple threads. In contrast to many existing main-memory index structures, our approach does not sacrifice write for read performance but offers efficient update operations. Using different workloads, e.g., range and point queries, mixed workloads, and realistic workloads, over different data sets, e.g., uniformly distributed, clustered, and real world, we evaluate the performance of our approach and compare it to various other state-of-the-art multidimensional index structures.

1.3 Outline

The remaining sections address the problems stated above.

Chapter 2 defines concepts and notations relevant for this thesis. It gives an overview over multidimensional index structures and their various subclasses. It also studies techniques commonly used to adapt index structures to the features of modern hardware. Finally, it presents a novel multidimensional range query benchmark, which we designed to facilitate realistic experiments. The benchmark is executed on real-world data from the bioinformatics domain.

Chapter 3 presents a novel cache-optimized index structure to execute one-dimensional range queries on single database columns. It is based on skip lists, but employs a fundamentally different memory layout that is tailored to the characteristics of modern CPUs. Furthermore, it applies SIMD instructions to accelerate range queries. We compare it with several state-of-the-art main-memory index structures using synthetic and real-world data sets, and show that it achieves the best range query performance across all evaluated data sets and workloads.

Chapter 4 studies the performance of multidimensional range queries on modern hardware. We carefully adapt three popular multidimensional index structures and two scan flavors to main-memory storage, SIMD instructions and multithreading. We investigate their respective strengths and weaknesses using different synthetic and realistic partial- and complete-match range query workloads on different synthetic and real-world data sets. We show that sequential scans can leverage the features of modern hardware better than multidimensional index structures, which increases their relevance for main-memory database systems.

Chapter 5 proposes a novel, fast and space-efficient multidimensional index structure to execute point and range query workloads in main memory. It employs a cache-optimized data layout that takes cache lines into account, enables sequential data access, and can dynamically ingest updates. Furthermore, it can utilize multithreading to speed-up search queries.

1 Introduction

We demonstrate that our approach outperforms other state-of-the-art multidimensional index structures for complete- and partial-match range queries. It even beats a sequential scan up to a query selectivity of 20%.

Chapter 6 summarizes the results of this dissertation and provides an outlook to future work.

1.4 Prior Publications

Some chapters of this thesis are based on previous peer-reviewed publications.

Chapter 2 introduces the *Genomic Multidimensional Range Query Benchmark* (GMRQB), which was previously published in [Sprenger et al. 2018b]. The roles of the authors were as follows: Sprenger designed GMRQB and wrote the manuscript, which was revised by Schaefer and Leser.

Chapter 3 presents the *Cache-Sensitive Skip List* (CSSL), which was previously published in [Sprenger et al. 2016]. The authors' roles can be assigned as follows: Sprenger designed, implemented and evaluated CSSL and wrote the manuscript. Leser and Zeuch provided helpful feedback on the concepts behind CSSL. Leser revised the manuscript.

Chapter 4 conducts an experimental analysis of the performance of multidimensional range queries on modern hardware, which was previously published in [Sprenger et al. 2018b]. The roles of the authors were as follows: Sprenger designed and implemented most parts of the analysis. Schaefer implemented a main-memory variant of the VA-file and provided helpful feedback on the design of the analysis and drafts of the manuscript. Leser provided extensive feedback on the design of the analysis and revised the manuscript.

Chapter 5 introduces a novel main-memory based multidimensional index structure, the BB-Tree, which is submitted for publication [Sprenger et al. 2018a]. The authors' roles can be assigned as follows: Sprenger designed, implemented and evaluated the BB-Tree and wrote the manuscript. Schaefer and Leser provided helpful feedback on the concepts behind the BB-Tree and drafts of the manuscript and reworked the manuscript.

2 Fundamentals

This chapter starts with introducing terminology and notations relevant for the remaining thesis (see Section 2.1). In Section 2.2, we present popular means to processing multidimensional range query workloads. Section 2.3 studies techniques for adapting index structures to the properties of modern hardware; we focus on commonly available hardware features, such as main-memory storage, *Single Instruction Multiple Data* (SIMD), and multithreading. Finally, Section 2.4 presents the *Genomic Multidimensional Range Query Benchmark* (GMRQB), a set of eight realistic multidimensional range query templates to be executed on real-world genomic data. We designed GMRQB to increase the practical relevance of our experimental results.

2.1 Terminology

This thesis studies the execution of range queries on data kept in main memory. To this end, we propose novel cache-optimized index structures that leverage the properties of modern hardware. We target one-dimensional and multidimensional range query workloads. While one-dimensional range queries are processed on sets of keys resembling single columns of a database, *multidimensional range queries* (MDRQ) are applied to sets of tuples, which resemble multiple or all columns of a database. In this thesis, we focus on numerical data. When dealing with non-numerical data, e.g., strings, we convert them into a numerical representation before indexing. In the following chapters, we use the terms tuple, feature vector, and data object synonymously. Similarly, we use the terms attribute, feature, and dimension synonymously.

Definition 1 (Data Set). A data set $D = \{t_0, t_i, \dots, t_{n-1}\}$ is a collection of n tuples that share the same attributes.

We consider only duplicate-free, homogeneous data sets, where all tuples feature the same numerical attributes. Often, all attributes of a tuple belong to the same domain, for instance $[0, 1]$. Consequently, the number of tuple attributes is derived from the dimensionality of the data set. For instance, we consider a data set that holds tuples, each featuring three attributes, to be three-dimensional. The cardinality of an attribute resembles the number of contained distinct values across the entire data set. An attribute with a low (high) cardinality has few (many) distinct values.

The index structures that we devise in this thesis do not require data sets to provide tuples in a certain sort order. Especially for multidimensional data, that is not always even possible.

Definition 2 (Range Query). A range query $RQ = \{rp_0, rp_j, \dots, rp_{m-1}\}$ for an m -dimensional data set D consists of m range predicates. Each range predicate $rp_j = [lb_j, ub_j]$ is associated with the respective attribute of D and is specified within the according domain. A tuple $t \in D$ matches RQ , iff all range predicates evaluate to true, i. e., $\forall j : lb_j \leq t[j] \leq ub_j$.

2 Fundamentals

First, depending on the data set D , range queries can be one-dimensional or multidimensional. Second, MDRQ can be further divided into complete-match and partial-match range queries [Robinson 1981]: Complete-match range queries specify predicates for all attributes of a data set, whereas partial-match range queries specify predicates for only a subset. We model partial-match range queries as complete-match range queries that use the predicate $[-\infty, +\infty]$ for all attributes that are not restricted.

Typically, range queries return either a list of all matching tuples or a list of the unique identifiers (often called *TIDs*) of the matching tuples.

Definition 3 (Selectivity). *The selectivity of a range query RQ against a data set D is defined as the percentage of tuples from D that match RQ .*

We consider queries that select only a small portion of a data set to have a high selectivity. In contrast, we consider queries that select a large portion of a data set to have a low selectivity [Selinger et al. 1979].

Definition 4 (Lookup). *A lookup (or point query) against a data set D asks if a certain tuple is part of D . Alternatively, lookups can be considered as range queries that exclusively consist of range predicates having identical lower and upper boundaries.*

Since we consider only duplicate-free data sets, lookups always match one single tuple. Hence, they have a selectivity of $1/n$, where n equals the cardinality of D . Although lookups can be implemented as range queries, most index structures provide a separate lookup operator that exploits the unique characteristics of lookups, e.g., searching for an exact match instead of comparing with *greater than* and *less than*.

Definition 5 (Access Method). *(Data) access methods provide means to evaluate search queries against data sets. They access a data set D and return the subset of D that matches the query.*

Access methods can be one-dimensional or multidimensional. One-dimensional access methods provide means to evaluate search queries on single database columns. Multidimensional access methods provide means to process search queries on some, many or all columns of a database. In database systems, common access methods are sequential full-table scans and index structures. While full-table scans compare each tuple of D with the given search query and therefore need to access the complete data, index structures can apply pruning techniques that reduce the amount of data to be investigated. The pruning capabilities of index structures come at the price of a non-sequential access pattern. Hence, full-table scans are beneficial for low selectivities, where most parts of the data set must be accessed and cannot be pruned anyway, and index structures are superior for high selectivities.

2.2 Multidimensional Access Methods

Over the last decades, many different access methods have been proposed to execute search operations, e.g., point queries, range queries, or similarity search, on multidimensional data

sets. According to [Gaede et al. 1998], multidimensional access methods can be divided into *point access methods* (PAM), which store and search point objects, and *spatial access methods* (SAM), which can also handle spatially-extended objects, e.g., polygons, or polyhedrons.

In this thesis, we strongly focus on range queries but also evaluate point queries, as these can be considered as range queries specifying identical lower and upper boundaries. We do neither study nor evaluate similarity search, which is mainly applied to high-dimensional data featuring hundreds to thousands of dimensions [C. Böhm et al. 2001]. We thus do not consider techniques targeting similarity search workloads, like MVP-trees [Bozkaya et al. 1999], VP-trees [Yianilos 1993], or M-trees [Ciacchia et al. 1997].

The following sections discuss access methods that can execute range queries on multidimensional point data. We start with introducing the sequential scan as a baseline approach, and then continue with studying dedicated *multidimensional index structures* (MDIS), both PAM (see Section 2.2.2) and SAM (see Section 2.2.3).

2.2.1 Sequential Scan

A *sequential scan* (or full-table scan) always processes the entire data set to evaluate a query. As opposed to index structures, scans do not require any additional data structure, which leads to an ideal memory utilization but prohibits pruning of non-relevant tuples. When keeping data in a flat array of length n and using four-byte floating-point values to implement m -dimensional data objects, a sequential scan requires $n * (4 * m)$ bytes of memory for storage. For instance, when handling a data set of one Million ten-dimensional objects, it requires approximately 38 *megabytes* (MB) of space.

Algorithm 1 presents the scan-based evaluation of MDRQ. It receives five input parameters: (1) *data_set* holds the data that the range query is applied to, (2) n resembles the cardinality of *data_set*, (3) m resembles the dimensionality of *data_set* and also equals the number of search predicates, (4) *lower* defines the lower boundary of the range query, and (5) *upper* defines the upper boundary. Using a *for* loop, the algorithm iterates over the data set held in a two-dimensional array. For each tuple, *data_set*[i], it compares all m attributes with the according predicates of the range query (see Lines 5-10). The algorithm allows *early breaks* from the inner loop: As soon as an attribute of a tuple does not match the respective range predicate, the search prunes all further steps and proceeds with the next tuple (see Line 8). If all predicates evaluate to true, i.e., the tuple matches the query, the unique identifier of the tuple, here implicitly defined by the array index i , is added to the result set (see Lines 11-13). Tuple identifiers may also be stored in a separate array allowing individual values. Finally, the results are returned (see Line 15).

Scans support insert operations very efficiently, because data do not need to be kept in any particular order. Hence, inserts can be implemented by appending new tuples to the array holding the data set. In turn, deletes can be implemented by replacing the to-be-deleted tuple with the tuple stored at the tail of the data set, which effectively shortens the data set by one position and avoids producing free space within the array.

The implementations of sequential scans are minimalist requiring only few lines of code. They offer various starting points for optimizations. First, if statistics about selectivities of single attributes were available, we could improve the early break capabilities by comparing

Algorithm 1 Scan-based evaluation of multidimensional range queries.

data_set: The data set that the range query is applied to.
n: The cardinality of the data set.
m: The dimensionality of the data set.
lower: The lower boundary of the range query.
upper: The upper boundary of the range query.

```

1: function SCANRANGEQUERY(data_set, n, m, lower, upper)
2:   results  $\leftarrow \emptyset$ 
3:   for i  $\leftarrow 0$  to n - 1 do
4:     match  $\leftarrow$  true
5:     for j  $\leftarrow 0$  to m - 1 do
6:       if lower[j] > data_set[i][j] OR upper[j] < data_set[i][j] then
7:         match  $\leftarrow$  false
8:         break
9:       end if
10:    end for
11:    if match then
12:      results  $\leftarrow$  results  $\cup \{i\}$ 
13:    end if
14:  end for
15:  return results
16: end function

```

attributes in the order of the estimated selectivities, i. e., attributes queried with high selectivities are compared first. Second, we could use different implementations for partial-match and for complete-match queries. This would allow to skip comparisons for dimensions not restricted by a partial-match query. Third, researchers have proposed advanced scan variants optimized for main-memory storage and modern *central processing unit* (CPU) architectures [Broneske et al. 2014; Broneske et al. 2017a; Y. Li et al. 2013], e. g., scans utilizing vectorized instructions [J. Zhou et al. 2002; Willhalm et al. 2009; Willhalm et al. 2013; Polychroniou et al. 2015], scans on compressed data [Y. Li et al. 2014; Lang et al. 2016; Holloway et al. 2007], or parallel scans [Qiao et al. 2008]. In addition to the basic scan variant shown in Algorithm 1, we study vectorized and multithreaded scan implementations in this thesis.

Assuming that data are stored in a dense array without any intermittent free space, scans solely access consecutive memory locations, which is not only beneficial when processing data stored on disk but also when applying range queries to data held in main memory. Sequential scans effectively take advantage of prefetched cache lines and minimize the number of data transfers by utilizing complete cache lines.

2.2.2 Point Access Methods (PAM)

Point access methods (PAM) store and search multidimensional point data by partitioning the data space into multiple regions. Seeger and Kriegel [Seeger et al. 1990] classify PAM according to three properties of the obtained regions: (a) regions are pairwise disjoint or not, (b) regions are rectangular or not, and (c) the union of all regions covers the complete data space or not. For instance, the *Universal B-tree* (UB-tree) does not partition data into rectangular regions, but indexes the z-order of multidimensional points in a B-tree [Bayer 1997]. Most PAM, e.g., the kd-tree [Bentley 1975], the K-D-B-tree [Robinson 1981], or the quadtree [Finkel et al. 1974], divide data into disjoint, rectangular regions that cover the complete data space.

kd-tree

Kd-trees [Bentley 1975] hierarchically organize m -dimensional data points in a binary search tree and support different multidimensional search operations, e.g., point queries, range queries, or similarity search [Sproull 1991]. Each node of a kd-tree holds exactly one data object. Inner nodes are used to recursively partition the data space. Each inner node employs one of the dimensions of the data space as delimiter ($delimiter \in \mathbb{N}; 0 \leq delimiter < m$): The left child tree holds all points that have a smaller or equal value in the delimiter dimension, and the right child tree contains the rest. Hence, inner nodes resemble $(m-1)$ -dimensional axis-aligned hyperplanes. By default, delimiter dimensions are chosen in a round-robin fashion, i.e., successive tree levels employ successive dimensions as delimiter. Today, many major database systems use kd-trees to index spatial data, e.g., PostgreSQL¹, or SAP HANA².

Figure 2.1 illustrates a kd-tree (shown on the right) that indexes six points from a two-dimensional data space (shown on the left). The first tree level splits the data space on the first dimension, age, and the second tree level on the second dimension, body weight. In general, the shape of a kd-tree is determined by the order of the insertions. In this example, the data objects could have been inserted in the following order: (31,120) (50,80) (20,100) (25,75) (40,90) (57,65).

When implementing m -dimensional data objects with arrays of four-byte floating-point values, storing the delimiter information in eight-bit integers, and running on a 64-bit architecture featuring eight-byte pointers, a kd-tree requires $4 * m + 1 + 2 * 8$ bytes to store one node and $n * (4 * m + 1 + 2 * 8)$ bytes for a complete data set of n tuples. For instance, a kd-tree over one Million ten-dimensional points requires approximately 54.36MB of memory space, which, compared to the raw size of the data set, equals an index overhead of 16.21MB.

When executing MDRQ, the search algorithm recursively traverses the kd-tree from the root to the leaf nodes. At each visited node, two actions are conducted. First, the stored data object is added to the result set if it is contained in the query range. Second, the delimiter dimension of the stored data object is compared with the corresponding dimension of the query object to

¹PostgreSQL: Documentation: 10: 11.2. Index Types, <https://www.postgresql.org/docs/current/static/indexes-types.html>, Last access: August 29, 2018.

²KNN | SAP HANA Platform | SAP Help Portal, <https://help.sap.com/viewer/2cfbc5cf2bc14f028cfbe2a2bba60a50/1.0.12/en-US/f2440c6b3daa41dd8cc9fc5b64805a68.html>, Last access: August 29, 2018.

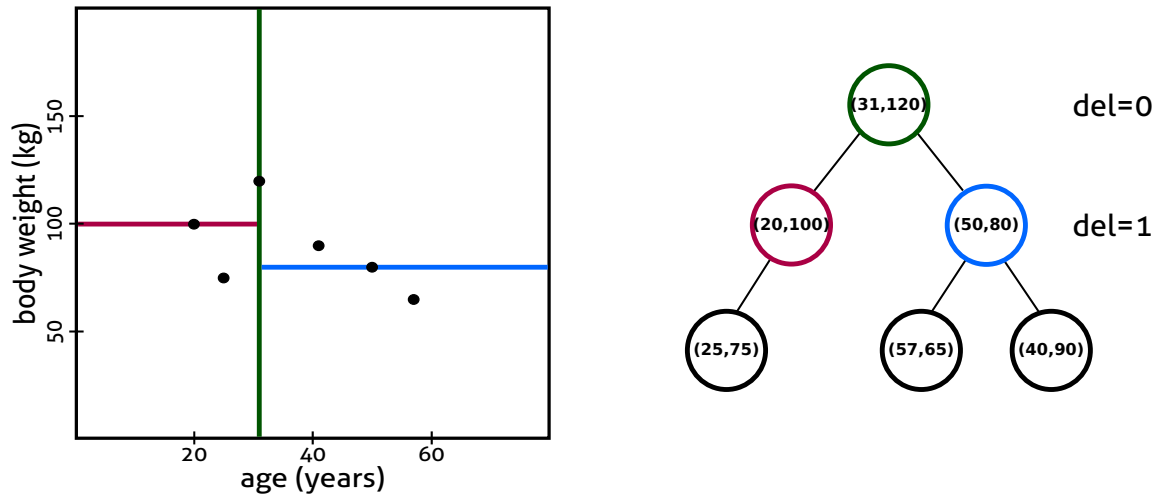


Figure 2.1: A kd-tree (right) indexing six points from a two-dimensional data set (left).

determine which subtrees need to be taken to continue searching. Note that, unlike for point queries, range queries usually have to visit multiple parallel subtrees of a kd-tree.

Traditionally, kd-trees can only be balanced³ by manually rebuilding the complete index structure. However, there exist a number of variants that can keep a kd-tree always balanced, often at the cost of more complicated update operations, e.g., the Bkd-tree [Procopiuc et al. 2003], the divided kd-tree [Kreveld et al. 1991], or the holey brick-tree [Lomet et al. 1990].

When inserting new data, the inner nodes are navigated to find a leaf. This leaf node will become the parent node of the new node. No rebalancing operations, as in a B-tree, are needed. When deleting data from a kd-tree, the inner nodes are navigated to find the node, X , that holds the to-be-deleted object. If X is a leaf node, it can be removed. If X is an inner node, two techniques can be used to perform the deletion. First, we could recreate the according part of the kd-tree, while skipping X . Second, we could replace X with a replacement node, R , that is chosen from the children of X . We could either choose the node from the right sub tree of X having the minimal value in the delimiter dimension of X , or we could choose the node from the left sub tree of X having the maximal value in the delimiter dimension of X . We would have to delete R recursively.

Since the initial proposal by Bentley more than forty years ago, researchers have proposed different kd-tree variants tailored to different needs. Adaptive kd-trees [Bentley et al. 1979] make the partitioning of kd-trees more sensitive to the indexed data and relax the constraints of kd-trees with regards to delimiter dimensions and splitting hyperplanes. Adaptive kd-trees store all data objects in leaf nodes and choose delimiter dimensions such that the tree is balanced, which can improve the worst case complexity of search operations but hinders dynamic updates. The

³Balanced search trees keep all leaves at the same depth.

K-D-B-tree [Robinson 1981], the Bkd-tree [Procopiuc et al. 2003], and the holy brick-tree [Lomet et al. 1990] adapt the concepts of kd-trees to external memory. Zhou et al. [K. Zhou et al. 2008] exploit graphic processors to build up kd-trees in parallel. Similarly, Choi et al. [Choi et al. 2010] present parallel construction algorithms for multi-core CPUs. The extended kd-tree [Chang et al. 1981] and the skD-tree [Khamayseh et al. 2007] are variants of the kd-tree that can handle spatially-extended objects.

Traditional kd-trees are main-memory index structures by design, because they employ small nodes holding only a single data object, and because they do not optimize for disk I/O. In this thesis, we study the kd-tree and compare it to our own approach.

K-D-B-tree

K-D-B-trees [Robinson 1981] integrate the concepts of kd-trees and B^+ -trees [Comer 1979]. K-D-B-trees organize multidimensional point objects in a search tree, employing a partitioning technique similar to kd-trees. However, they use inner nodes only for partitioning and store data solely in leaves. As K-D-B-trees optimize for disk I/O, they tailor node sizes to disk block sizes, allowing inner nodes to hold multiple delimiters and leaf nodes to keep multiple data objects. The search engine Apache Lucene⁴ uses a variant of the K-D-B-tree⁵, the Bkd-tree [Procopiuc et al. 2003], to index spatial data.

In the optimal case, when all nodes are completely occupied, K-D-B-trees require n/B disk blocks of size B to store a data set of n tuples. In contrast, in the worst case, K-D-B-trees require $n/(B * (f/B))$ disk blocks, assuming that leaves are filled with at least f entries. Note that, in contrast to B^+ -trees, K-D-B-tree typically do not guarantee a minimum node fill degree.

When executing search queries, K-D-B-trees navigate the inner nodes to determine those leaf nodes that can possibly hold data objects satisfying the query. These leaf nodes are scanned to find the true query results.

K-D-B-trees keep their search tree always balanced. They require rebalancing operations, such as node splits, to cope with overflowing and underflowing nodes. In K-D-B-trees, rebalancing operations can involve multiple paths between different leaf nodes and the root, recursively splitting several nodes.

Similar to kd-trees, insertions are implemented by locating the leaf node that is responsible for a new data object. If the leaf node has free space, the new data object is appended. Otherwise, the node needs to be split. If no minimal requirements on node occupancy are specified ($f = 0$), deletions can be implemented by removing the to-be-deleted object from the according leaf node. Otherwise, multiple nodes have to be merged when underflows occur.

Researchers have presented several variants of the K-D-B-tree. The Bkd-tree [Procopiuc et al. 2003] is based on a static implementation of the K-D-B-tree. It achieves high space efficiency at the cost of more complicated updates, which require to rebuild the entire index structure. However, Bkd-trees strongly reduce the cost of such rebuilds by employing multiple instances

⁴Apache Lucene - Welcome to Apache Lucene, <https://lucene.apache.org>, Last access: August 29, 2018.

⁵Uses of Class org.apache.lucene.index.PointValues (Lucene 7.2.1 API), https://lucene.apache.org/core/7_2_1/core/org/apache/lucene/index/class-use/PointValues.html, Last access: August 29, 2018.

2 Fundamentals

of the static K-D-B-tree, where each instance indexes a different portion of the data. Consequently, updates are implemented by periodically rebuilding a subset of these instances (instead of rebuilding all instances). The hB-tree [Lomet et al. 1990] improves the insert algorithm of the K-D-B-tree, which, in the worst case, has to consider multiple paths between different leaf nodes and the root node when splitting nodes. In contrast, the hB-tree can restrict node splits to a single path in the search tree.

K-D-B-trees target external memory as storage layer and are less beneficial when deployed in main memory. Therefore, we do not consider K-D-B-trees in this thesis.

Quadtree

Quadtrees [Finkel et al. 1974] are very similar to kd-trees, but let inner nodes split the data space in all dimensions, whereas kd-trees split in only one dimension. Originally, quadtrees were proposed for two-dimensional data sets that are handled in four-ary search trees, recursively splitting the data space into four subregions, but they can also be applied to higher dimensionalities by increasing the fan out. Nodes can hold up to four entries, each representing a subregion. If a subregion contains only one object, the node stores the data object in the corresponding entry, without the need for a further tree level. Otherwise, it holds a pointer to a child node at the next lower tree level, which is responsible for all objects of the subregion. Quadtrees are included in PostgreSQL⁶.

The space requirements of quadtrees depend on the node occupancies. If most nodes are completely occupied, quadtrees require similar amounts of space as kd-trees. However, if many nodes contain only one data object, although they reserve space for four entries, quadtrees waste a lot of memory space.

When executing MDRQ, quadtrees navigate the search tree to prune those regions that do not contain any matching objects. In particular, at each node, the search algorithm compares all four entries, i. e., subregions, with the query object. If an entry is a data object that satisfies the range boundaries, it is added to the result set. Otherwise, it is ignored. If an entry is a region object that intersects the range query object, the search algorithm visits the child node. Otherwise, it is ignored.

Similar to kd-trees, the structure of quadtrees strongly depends on the insertion order. Quadtrees do not guarantee to be balanced, but allow data objects to be stored at any tree level. As unbalanced quadtrees can lead to a poor space utilization, it may be useful to periodically perform rebuilds.

Insertions are implemented by locating the node that is responsible for the region of the data space that the new point belongs to. If the node has free space, the new point can be inserted and the insertion procedure terminates. Otherwise the node needs to be split to allocate new space. When deleting an object from a quadtree, we first locate the node that holds the object and delete it there. If the node becomes empty, we can remove it including the reference from the parent node. Note that a deletion of a single object may cause the removal of multiple nodes.

⁶PostgreSQL: Documentation: 10: 11.2. Index Types, <https://www.postgresql.org/docs/current/static/indexes-types.html>, Last access: August 29, 2018.

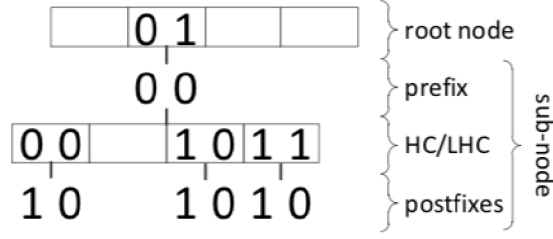


Figure 2.2: A PH-tree indexing the bitstrings of three two-dimensional points: $(1, 8)$, $(3, 8)$, $(3, 10)$. The bitstrings are: $(0001, 1000)$, $(0011, 1000)$, $(0011, 1010)$. The figure is taken from [Zäschke et al. 2014].

When increasing the fan out of inner nodes, quadtrees can handle more than two dimensions. For instance, in an octree [Meagher 1982], inner nodes divide the data space into eight partitions, which allows to handle three-dimensional point objects.

Quadrees use a simple memory layout that can be easily adapted to main memory. However, in this thesis, we consider kd-trees instead of quadrees, because they show better support for higher dimensionalities. Quadrees typically become very inefficient for more than three dimensions, because it is likely that many nodes remain sparsely filled, especially when data are dynamically inserted.

PH-tree

The *PATRICIA-hypercube-tree* (PH-tree) [Zäschke et al. 2014] is a recent MDIS that merges the concepts of quadrees [Finkel et al. 1974] with prefix sharing [Morrison 1968] and bitstream serialization [Germann et al. 2009] to achieve high space efficiency when stored in main memory. Technically, PH-trees are radix trees that index the bitstrings of multidimensional data objects. For the one-dimensional domain, where objects consist of single values, they hence resemble binary PATRICIA-tries [Morrison 1968]. For the multidimensional domain, inner nodes of PH-trees split the data space in all m dimensions, similar to quadrees. Thus, nodes hold up to 2^m entries and the fan out of inner nodes equals 2^m . PH-trees use an hypercube-based addressing scheme when navigating inner nodes to efficiently find child nodes of interest.

As opposed to most previously discussed access methods, the structure of PH-trees does not depend on the order of insertions, a property inherited from PATRICIA-tries. Furthermore, the depth of the tree does not depend on the number of indexed objects, but on the length of the indexed bitstrings. For instance, when using four-byte floating-point values to implement data objects, the resulting PH-tree has a maximum depth of 32 and contains 2^{32m} leaf nodes at most. Figure 2.2, which is taken from [Zäschke et al. 2014], illustrates an exemplary PH-tree indexing three two-dimensional points.

Similar to PATRICIA-tries, PH-trees can apply compression techniques to increase their space efficiency. When multiple bitstrings have identical prefixes, PH-trees merge redundant information and store them only once. In the example from Figure 2.2, all points share the prefix 00 in

2 Fundamentals

the first dimension and the prefix 10 in the second, which allows the PH-tree to compress the root node.

Executing a multidimensional range query over a PH-tree consists of two steps: (1) Use a point query⁷ to search for the lower boundary of the range query. If the query succeeds, the found entry resembles the smallest object matching the range query. Otherwise, we end up at a node that is used as starting point for the second step. (2) All successive nodes need to be visited to find the remaining objects satisfying the range boundaries. As for most access methods, the complexity of this step correlates negatively with the selectivity of the query.

By default, PH-trees are unbalanced search trees. However, imbalances are limited, because the depth of a PH-tree depends on the size of the data type used to implement data objects.

Update operations are implemented similarly to PATRICIA-tries and need to touch at most two nodes, because PH-trees do not require rebalancing.

PH-trees belong to the tiny group of MDIS that are optimized to be stored in main memory. However, they cannot be considered as general-purpose MDIS and are often outperformed by traditional MDIS approaches [Wang et al. 2016], like R-trees, despite these approaches were originally designed for disk storage. PH-trees excel for low-dimensional data sets, but show a vastly decreasing space efficiency with an increasing dimensionality. In this thesis, we consider the PH-tree for data sets that can be completely indexed in main memory, but we omit it for experiments where the space requirements of the PH-tree exceed the memory capacities of our evaluation machine.

VA-file

Vector approximation-files (VA-files) [Weber et al. 1998] combine the concepts of grid files [Nievergelt et al. 1984] with partitioned hashing methods [Ullman 1988]. VA-files can be considered as enhanced scans, because they can apply pruning techniques while employing a sequential access pattern. According to hash functions chosen at initialization time, VA-files divide an m -dimensional data space into 2^b cells of equal size. Each cell is identified by a distinct encoding of b bits that is used to approximate the contained data objects. Thus, all objects belonging to the same cell share the same approximation value. VA-files manage data objects in an array, which we call *data_list*. On top of that, they employ a list of all approximation values, which we call *approx_list*.

As every approximation of a cell requires b bits, VA-files incur an indexing overhead of $2^b * b$. For instance, when using ten bits, the indexing overhead equals 1 KB, and when using 20 bits, the indexing overhead equals 2.5MB. Therefore, the space efficiency of VA-files strongly depends on how many bits are needed for approximation. In turn, the approximation length (b) should be chosen depending on the size of the data set. When using the same cell capacity, large data sets require more bits for approximation than small data sets. For instance, we would need ten bits to partition one Million objects into cells of size 1,000, whereas four Thousand objects would require only two bits.

Executing MDRQ on VA-files requires multiple steps: (1) The lower and upper boundaries of

⁷The complexity of point queries depends on the number of investigated nodes, w , and the dimensionality of the data space, m : $O(w * m)$.

the range query object are approximated using the encoding values of the cells of the data space they belong to. (2) The search algorithm scans *approx_list* to determine the cells that intersect the range query object. (3) The search algorithm scans *data_list* to find the true results.

By default, VA-files are bulk loaded with all data objects. Bulk loading is necessary, because VA-files require knowledge about the distribution of the data to evenly distribute the data objects among the cells. New objects can be inserted by first obtaining their approximation and subsequently inserting them into *data_list*. Existing data can be deleted by first obtaining their approximation and subsequently removing them from *data_list*. When inserting new objects that follow a data distribution grossly different from the one initially used to determine the hash functions, the partitioning becomes unsuitable. Thus, VA-files do not support updates very efficiently.

Weber et al. [Weber et al. 2000] proposed a parallel variant of the VA-file that exploits networks of workstations to accelerate search queries.

For historical reasons, as most MDIS, VA-files were originally designed for systems with small main memories and large disks. They keep approximations in main memory, but manage data objects on disk. However, on modern server machines, main memory capacities are much larger, allowing to store both approximations and data in memory. Moreover, the sequential access pattern of VA-files may lead to a high cache efficiency on modern CPUs, making it a good choice for in-memory indexing. In our experiments, we consider the VA-file.

Space-filling curves

Space-filling curves present a radically different approach to multidimensional indexing than the methods discussed in the previous sections [Sagan 2012]. A space-filling curve covers the complete m -dimensional data space and maps it into the one-dimensional domain. They can be used to transform multidimensional point objects into one-dimensional values, which can be indexed with traditional one-dimensional index structures, like B-trees. Prominent examples are the UB-tree [Bayer 1997] and z-ordering [Orenstein et al. 1984]. In most cases, space-filling curves are a viable alternative for point queries, but lack efficiency for MDRQ, which are difficult to transform into the one-dimensional domain [Gaede et al. 1998]. Especially partial-match range queries are challenging, because most space-filling curves treat all dimensions identical. Due to our strong focus on range queries, we do not consider techniques based on space-filling curves in this thesis.

2.2.3 Spatial Access Methods (SAM)

Spatial access methods (SAM) store and search spatially-extended objects, like rectangles. They typically use minimum bounding rectangles (MBR) to partition a data space into multiple regions, which can be indexed with search trees. SAM are especially useful for geographical

2 Fundamentals

database systems, e.g., PostGIS⁸, Oracle Spatial⁹, SpatiaLite¹⁰, or SpaceBase¹¹. Although SAM can be used to handle multidimensional points [Kanth et al. 2002], they are typically less efficient than PAM for such data.

As shown by Seeger and Kriegel [Seeger et al. 1988], SAM can be considered as PAM that apply one of the following three techniques to add support for spatially-extended objects: (1) *Overlapping regions* allows indexing objects that are enclosed by multiple index regions, which is in contrast to PAM dividing data objects into disjoint partitions. As a consequence, insert algorithms must decide between multiple index buckets for storing an object. (2) *Clipping* prohibits overlapping index regions, but divides a spatially-extended object into multiple simpler objects, which are stored in one or multiple index buckets. (3) *Transformation* maps spatially-extended objects into point objects of higher dimensionality that can be managed with PAM.

The following sections present selected SAM, namely the R-tree [Guttman 1984], the R*-tree [Beckmann et al. 1990] and the R+-tree [Sellis et al. 1987]. We also sketch SAM approaches further away from our own research, e.g., the X-tree [Berchtold et al. 1996], or the M-tree [Ciaccia et al. 1997]. For a broader overview, we refer the interested reader to excellent surveys, like [Gaede et al. 1998] or [C. Böhm et al. 2001].

R-tree

R-trees [Guttman 1984] manage spatially-extended objects in a balanced search tree. Inner nodes recursively partition the data space by storing *minimum bounding rectangles* (MBR), which define the minimal enclosure of all objects managed in the corresponding subtree. The granularity of MBR increases with the depth of the tree. Multiple MBR of the same inner node can possibly overlap, because R-trees apply the *overlapping regions* technique. However, to maximize the pruning capabilities of the search algorithms, it is crucial that overlaps between different MBR belonging to the same tree level are as small as possible. Otherwise, many different paths (or sub trees) would need to be considered when evaluating a search query. Like for most search trees, the structure of R-trees depends on the order of the insertions. R-trees share multiple commonalities with B⁺-trees [Comer 1979]: (1) Data objects are exclusively managed in leaf nodes, while inner nodes are only used for pruning. (2) The search tree is height balanced, i.e., all leaf nodes are found at the same tree level. (3) The search tree is designed to be stored on disk and therefore optimizes data transfers between external memory and main memory. The sizes of the tree nodes are chosen depending on the sizes of the disk blocks: One inner or leaf node fills an entire disk block. Today, R-trees are included in many

⁸PostGIS - Spatial and Geographic Objects for PostgreSQL, <https://postgis.net/>, Last access: August 29, 2018.

⁹Oracle Spatial and Graph, <http://www.oracle.com/technetwork/database/options/spatialandgraph/overview/index.html>, Last access: August 29, 2018.

¹⁰SpatiaLite: SpatiaLite, <https://www.gaia-gis.it/fossil/libspatialite/index>, Last access: August 29, 2018.

¹¹Parallel Universe, <http://www.paralleluniverse.co/spacebase/>, Last access: August 29, 2018.

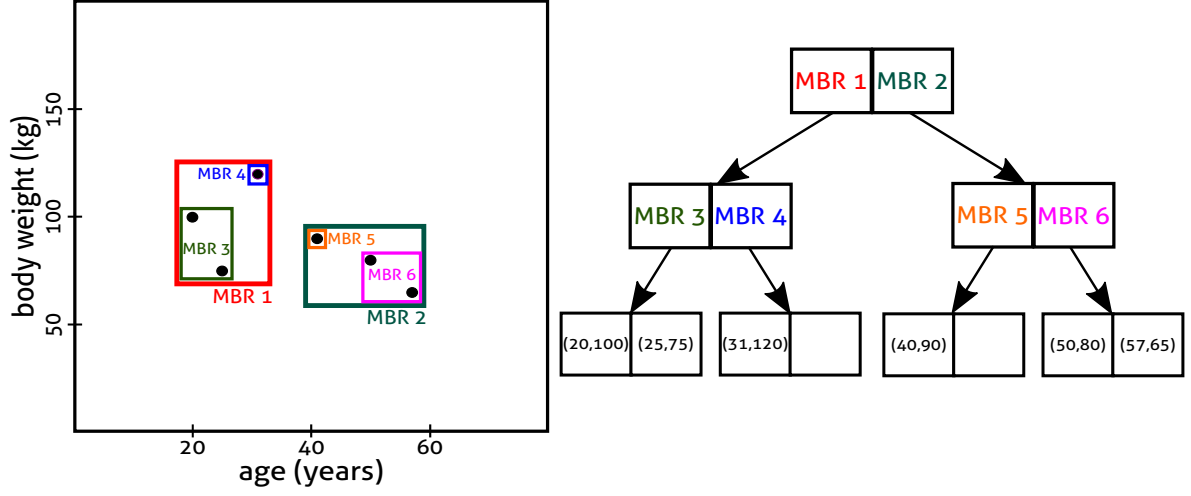


Figure 2.3: An R-tree (right) managing six points from a two-dimensional data set (left).

major database systems, e. g., MySQL¹², PostgreSQL¹³, and Oracle¹⁴.

Figure 2.3 illustrates an R-tree indexing the same six two-dimensional points as the kd-tree shown in Figure 2.1. In this example, inner nodes can hold up to two MBR and leaf nodes can store up to two data objects.

In the following analysis of the space requirements of R-trees, we assume that inner nodes and leaf nodes have the same capacity: Inner nodes hold up to k MBR and leaf nodes store up to k data objects. MBR are defined by two m -dimensional vectors, one resembling the lower left and another one resembling the upper right corner. Furthermore, we assume that the implementation uses arrays of four-byte floating-point values for MBR and data objects. Hence, one MBR requires $2 * (m * 4)$ bytes of space and one data object needs $m * 4$ bytes of space. An optimally-filled R-tree of height h has $\sum_{i=0}^{h-1} k^i$ inner nodes and n/k leaf nodes. Overall, it requires $\sum_{i=0}^{h-1} k^i * (k * 2 * m * 4) + (n/k) * (k * m * 4)$ bytes of space to manage n m -dimensional data objects. For instance, when using a node capacity of $k = 50$ and managing one Million ten-dimensional points in an R-tree of height 3, employing an R-tree induces an index overhead of $47.88\text{MB} - 38.15\text{MB} = 9.73\text{MB}$ compared to the raw data set.

R-trees execute MDRQ by starting at the root node and hierarchically traversing the tree down to the leaves. At each inner node, the algorithm intersects the query object with the stored MBR to determine those subtrees that may include data objects matching the query; all

¹²MySQL :: MySQL 5.7 Reference Manual :: 11.5.9 Creating Spatial Indexes, <https://dev.mysql.com/doc/refman/5.7/en/creating-spatial-indexes.html>, Last access: August 29, 2018.

¹³PostgreSQL: Documentation: 10: 37.14. Interfacing Extensions To Indexes, <https://www.postgresql.org/docs/current/static/xindex.html>, Last access: August 29, 2018.

¹⁴Spatial Concepts, https://docs.oracle.com/html/A88805_01/sdo_intr.htm, Last access: August 29, 2018.

2 Fundamentals

other subtrees are pruned. Whenever a leaf node is reached, all data objects contained in the MDRQ search object are added to the result set.

R-trees are balanced search trees that keep all leaves at the same depth. When inserting or deleting data, leaf nodes can overflow or underflow. Overflowing leaf nodes are split into two new nodes and underflowing leaf nodes are merged with neighboring nodes. In general, rebalance operations may involve multiple paths between different leaf nodes and the root node. R-trees provide two heuristics for splitting overflowing nodes, which both aim at reducing the coverage area of the new nodes:

- The *linear split* algorithm first determines those two objects, which have the largest distance, and assigns each object to one of the new nodes. Next, it distributes each of the remaining objects to the node that requires the least area increase.
- The *quadratic split* algorithm first determines those two objects that would maximize the MBR area when inserted into the same node. Each of these objects is added to one of the two new nodes. Then, the remaining objects are assigned to the new nodes such that the increase of the respective MBR area is minimized.

Over the last decades, various variants of the R-tree have been proposed. The R*-tree [Beckmann et al. 1990] and the R+-tree [Sellis et al. 1987] are two popular R-tree variants aiming at reducing MBR overlap. We discuss them in detail in the following sections. The P-tree [Jagadish 1990] uses polyhedral search regions and polyhedral bounding rectangles instead of rectangular regions and MBR. Hilbert R-trees [Kamel et al. 1994] order data objects by their Hilbert curve to improve the partitioning and respectively the coverage of MBR. X-trees [Berchtold et al. 1996] are based on R-trees but target data of very high dimensionality, which are, for instance, found in multimedia databases. Leutenegger et al. [Leutenegger et al. 1997] study different R-tree packing variants, e. g., the Hilbert R-tree, and introduce a new approach, the sort-tile-recursive algorithm. The CR-tree [K. Kim et al. 2001] compresses MBR to enable inner nodes to handle more entries, which effectively increases the tree fan out. Priority R-trees [Arge et al. 2004] use novel bulk-loading techniques to build asymptotically optimal R-trees. However, dynamic updates deteriorate the query efficiency of these R-trees. Further bulk-insert approaches have been presented [Arge et al. 2004; Qi et al. 2018].

Although R-trees were originally designed to be stored on disk, we can adapt them to main memory. Instead of tailoring the sizes of the tree nodes to the sizes of disk pages, we could choose node capacities based on the sizes of CPU cache lines, which improves cache line utilization and reduces the number of data accesses.

R*-tree

The *R*-tree* [Beckmann et al. 1990] is a variant of the R-tree that aims to reduce the overlap of MBR and improves the robustness towards different insertion orders. The R*-tree uses the same search and deletion algorithms as the original R-tree, but employs a fundamentally different technique for handling overflowing nodes. Although R*-trees typically drastically reduce MBR overlap, they do not guarantee to prevent it and therefore belong to the group of SAM employing

overlapping regions. SQLite¹⁵ uses R*-trees to index and search spatial data.

When a node overflows, the R*-tree first aggressively reinserts parts of the entries (typically around 30%), a technique called *forced reinsert*, which may postpone the necessary node split. Forced reinserts move entries between neighboring nodes, effectively reducing overlap. Eventually, if the forced reinsertion also causes an overflow, the according node is split.

R*-trees use none of the original split techniques, but propose a novel, more efficient but also more complex approach to split overflowing nodes. While the linear and quadratic split heuristics of the R-tree primarily aim to minimize the coverage (or area) of the new MBR, the split technique of the R*-tree follows multiple further goals associated with certain desirable effects: (1) A *minimal overlap* between MBR at the same tree level (or granularity) reduces the number of sub trees that need to be considered when evaluating search queries. (2) A *minimal margin* of MBR leads to a more quadratic shape of the MBR that is especially preferable for quadratic query rectangles. (3) An *optimal space utilization* reduces the tree height and effectively minimizes the query cost. In particular, the R*-tree chooses the dimension of the data space as split axis for the objects of the overflowing node that minimizes the perimeters of the new MBR.

The techniques of R*-trees come at the cost of an higher insert complexity [Hoel et al. 1992], but effectively reduce the coverage and overlap of the MBR typically resulting in a better search performance. Theoretically, R*-trees and regular R-trees have identical space requirements. In practice, enabled by forced reinserts, R*-trees often achieve a better space utilization than R-trees [Beckmann et al. 1990].

In our experiments with MDRQ, we consider the R*-tree as representative of SAM, because it achieves a very high search and space efficiency. We adapt it to an in-memory storage by tailoring the sizes of the nodes to the sizes of the cache lines.

R+-tree

The *R+-tree* [Sellis et al. 1987] is another popular R-tree variant. Similar to R*-trees, R+-trees aim to improve the search efficiency of R-trees by avoiding MBR overlap. Contrary to R*-trees, which strongly reduce overlap, R+-trees completely prevent overlap between MBR at the same tree level. R+-trees divide data objects into disjoint partitions and therefore belong to the group of SAM employing *clipping*.

When inserting data into an R+-tree, the insertion algorithm determines the number of MBR (or leaf nodes) that the new object intersects with. If the new data object intersects with only a single MBR, it can be directly inserted into the corresponding leaf node. Otherwise, the data object is split into multiple non-overlapping *sub-rectangles*, which are inserted into different leaves.

Depending on the concrete data set, R+-trees have higher space requirements than R-trees, because they may need to store one data object in multiple leaf nodes.

The search algorithms of R+-trees and R-trees behave very similar. R+-trees decompose a search object into disjoint sub-regions, which are evaluated on the search tree. When processing MDRQ, R+-trees may need to consider less sub trees than R-trees, because MBR do not overlap.

¹⁵The SQLite R*Tree Module, <https://sqlite.org/rtree.html>, Last access: August 29, 2018.

2 Fundamentals

In the case of point queries, R+-trees must consider only one single search path, whereas R-trees, in the case of MBR overlap, may need to follow multiple search paths.

R+-trees handle overflowing nodes similarly to regular R-trees. Though, splits of inner nodes do not only affect upper tree levels, but may also propagate to lower tree levels. Downward propagation of node splits is necessary, because R+-trees require that objects at lower tree levels are completely covered (or enclosed) by their parent objects at upper tree levels. Thus, when such a parent object gets split, the split must be recursively pushed down to the lower tree levels.

R+-trees can be adapted to a main-memory storage using the same technique that we proposed for R-trees, i.e., by tailoring the sizes of the tree nodes to the sizes of the cache lines. As the benefits of R+-trees, which mainly apply to point queries, come at the cost of increased space requirements, we do not consider them in this thesis.

Other Spatial Access Methods

In the following, we sketch other SAM approaches further away from our own research: (1) approaches for high-dimensional data, and (2) techniques targeting metric spaces.

In this thesis, we focus on multidimensional point data of low and moderate dimensionality (between two and 100 dimensions). Motivated by multimedia database systems storing feature vectors derived from objects, like images or videos, there exist different techniques to handle data of very high dimensionality (up to thousands of dimensions) [C. Böhm et al. 2001]. *X-trees* [Berchtold et al. 1996] are similar to R*-trees but introduce *super nodes*, which are larger than regular tree nodes and effectively reduce MBR overlap in the case of high-dimensional data spaces. The *Sphere/Rectangle-tree* (SR-tree) [Katayama et al. 1997] is another SAM addressing high-dimensional data that integrates the concepts of R*-trees and *similarity search-trees* (SS-trees) [White et al. 1996], which employ bounding spheres instead of bounding rectangles. SR-trees specify regions by intersecting bounding spheres and bounding rectangles. Compared to SS-trees, SR-trees typically achieve smaller regions (in terms of coverage) and also reduce the overlap between different regions. Compared to R*-trees, SR-trees are more efficient for high-dimensional data spaces. The *Telescope Vector-tree* (TV-tree) [Lin et al. 1994] considers only a subset of the dimensions of a data space for indexing and can therefore efficiently handle high-dimensional data while requiring few space for the index. Though, various experiments have shown that TV-trees are outperformed by other approaches, like X-trees [Berchtold et al. 1996].

Sometimes it is not feasible to transform data objects into features vectors, which makes SAM, such as R-trees, R*-trees, or R+-trees, useless. However, it may still be possible to derive distances between different objects. For such cases, various MDIS have been proposed that index the distances between the objects instead of the data objects themselves. The *vantage point-tree* (VP-tree) [Uhlmann 1991; Yianilos 1993] is technically a binary search tree that recursively partitions data objects by choosing one of the objects as pivot element and dividing the remaining objects into two partitions depending on their distance to the pivot element. The *generalized hyperplane-tree* (GH-tree) [Uhlmann 1991] is another SAM similar to the VP-tree. It is also a binary search tree, but uses two pivot elements at each inner node, instead of only one. Consequently, the left sub tree contains all objects that are closer to the left pivot element,

and the right sub tree holds all objects that are closer to the right pivot element. The *geometric near-neighbor access tree* (GNAT) extends the concepts of GH-trees by using more than two pivot elements at each inner node, which improves the search efficiency but also increases the build cost. Another popular approach to indexing metric spaces is the *paged metric tree* (M-tree) [Ciaccia et al. 1997], which is very similar to the VP-tree but explicitly targets settings, where the complete index is stored in external memory.

2.3 Index Structures on Modern Hardware

Most index structures for database systems were originally designed for machines with single-core CPUs, small main memory capacities and comparatively large disks. They provide only single-threaded implementations and store the indexed data in the external memory. Main memory is solely used for buffering disk accesses. For instance, VA-files manage their approximations in main memory, but store data objects on disk.

This section studies how to adapt index structures to the features of current hardware architectures. We focus on (a) using main memory for exclusive data storage, (b) accelerating search operators with SIMD instructions, and (c) exploiting the multithreading capabilities of modern CPUs. Note that the term *modern hardware* includes further recent hardware trends, such as high-speed networks [Rödiger et al. 2015], or highly-parallel co-processors, e.g., *graphics processing units* (GPU) [Bakkum et al. 2010], or *field-programmable gate arrays* (FPGA) [Teubner et al. 2013]. However, we restrict our research to hardware features available in server setups commonly used for general-purpose computing.

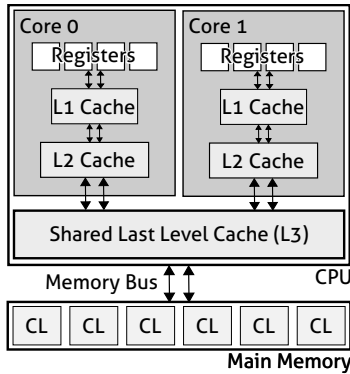
2.3.1 Modern Memory Hierarchy

Today, main memories [Drepper 2007; Bryant et al. 2003] are large enough to entirely hold most databases, turning it into the number one choice for data storage [Plattner et al. 2011]. Concurrently, the computing capabilities of processors have rapidly advanced, outpacing the improvements in memory latency. Due to the increasing parallelism of modern CPUs, the bus between the CPU caches and the main memory becomes a bottleneck [Manegold et al. 2000] and must be taken into account when optimizing data accesses.

CPUs execute instructions, e.g., arithmetic operations, on data held in the registers installed on the CPU. Thus, before the actual operation can be applied, the required data need to be loaded from main memory. The basic unit for transfers between the processor and the main memory are cache lines, which, today, typically consist of 64 bytes. The memory bus induces a high access latency. To overcome this obstacle, modern CPUs feature a hierarchy of different on-die caches. These cache hierarchies usually consist of three levels, that typically hold the most-recently accessed cache lines [Drepper 2007], although other replacement policies are possible.

Higher levels of the cache hierarchy contain subsets of data stored in lower levels. Hence, the *level 3* (L3) cache resembles a subset of the main memory, the *level 2* (L2) cache resembles a subset of the L3 cache, etc. When requesting a cache line, the processor first probes the highest cache level (L1) and, if the cache does not contain the cache line, hierarchically moves

2 Fundamentals



(a) CPU architecture.

Memory Layer	Capacity	Latency	Bandwidth (for random reads)
L1 Cache	32 KB	4-5 cycles	0.5 cycles per access
L2 Cache	256 KB	12 cycles	2 cycles per cache line
L3 Cache	8 MB	40 cycles	5 cycles per cache line
Main Memory	terabytes	42 cycles + 51 ns	5.9ns per cache line

(b) Performance numbers.

Figure 2.4: Memory hierarchy of Intel Skylake CPUs (Source: <https://www.7-cpu.com/cpu/Skylake.html>, Last access: August 29, 2018).

down to the lower levels. Accessing a cache line that is not featured in the *last level cache* (LLC), which typically equals the L3 cache on current processors, requires a main memory access to load the requested cache line via the memory bus into the CPU caches. Compared to cache misses occurring at upper levels, LLC misses are the most expensive in terms of latency (or performance). As a consequence, main-memory index structures aim to work as much as possible on data held in on-die CPU caches. Figure 2.4a illustrates the memory hierarchy [Jacob et al. 2010] of Intel’s Skylake microarchitecture, and Figure 2.4b provides according performance numbers¹⁶.

Modern operating systems execute processes within a virtual address space for multiple reasons, e. g., to separate individual processes from each other, and to unnoticably swap memory pages to disk when needed. For each process, an operating system manages the mappings between virtual memory addresses and physical memory pages in a *page table* that is stored in main memory. When programs access data using a virtual address, for instance when navigating a search tree, the *memory management unit* (MMU) scans the page table to find the physical location. Modern CPUs accelerate address translation by integrating a *translation lookaside buffer* (TLB). The TLB resembles a cache, storing the most-recent translations in a fast, low-latency memory integrated on the die. Before accessing the page table, the MMU probes the TLB and checks whether the requested virtual memory address has recently been translated. If the address is found, a TLB hit occurs and the MMU can return the desired translation. If the address is not found, a TLB miss occurs and the MMU must scan the page table stored in the comparatively slow main memory.

When employing index structures in main memory, various aspects must be considered to achieve high performance and fully leverage the capabilities of modern processors: (1) Navigation of the index structure should access data held in higher, faster levels of the memory hierarchy as often as possible. Especially, costly transfers between the main memory and the LLC need to be

¹⁶Source: <https://www.7-cpu.com/cpu/Skylake.html>, Last access: August 29, 2018.

avoided. (2) As data are always transferred at the granularity of cache lines, search algorithms must utilize as much of a cache line’s content as possible to minimize data transfers. (3) The evaluation of search queries should produce as few TLB misses as possible.

The most important technique to achieve high cache efficiency is employing data layouts that enable search algorithms to sequentially traverse over the indexed data. When accessing a cache line, modern processors also prefetch the successive cache line, a technique called *one block lookahead* prefetching [Smith 1982]. A sequential access pattern makes use of the prefetched cache line and therefore effectively prevents LLC and TLB misses. To this end, pointer-based implementations of search trees can be linearized to reduce random accesses [Schlegel et al. 2009]. Furthermore, the data layout of index structures should be aligned to the sizes of cache lines. For instance, search trees can adjust the capacity of their nodes to the sizes of the cache lines [Rao et al. 1999], which maximizes cache line utilization and reduces the number of data transfers.

In this thesis, we often compare different index structures with each other in terms of their cache efficiency. For instance, we may investigate how many LLC misses are produced per executed range query. To this end, modern CPUs provide hardware performance counters¹⁷, which are dedicated registers installed on the processor that sample the occurrences of performance-related events, like cache misses, cache accesses, or branch mispredictions. We use the *Performance Application Programming Interface* (PAPI) [Mucci et al. 1999], which provides an uniform interface to different CPU families, to obtain access to the hardware performance counters of the evaluation machine.

2.3.2 Single Instruction Multiple Data (SIMD)

Single Instruction Multiple Data (SIMD) is an execution model, where one instruction is simultaneously applied to multiple data elements [Flynn 1972]. Modern processors implement SIMD through vectorized instructions that are processed on dedicated, extra-wide registers. The *degree of parallelism* (DOP) of SIMD instructions depends on how many data elements fit into one SIMD register. For instance, a 32-byte SIMD register can hold eight four-byte values. In general, when processing data for which k values fit into one register, SIMD offers a theoretical speed-up of k . However, such optimal performance gains are rarely achieved in practice, because various other factors, like the memory bandwidth and the concrete vector instruction to perform, play an important role [Polychroniou et al. 2015].

Developers can implement SIMD parallelism either by using intrinsic functions¹⁸ provided by hardware vendors or by relying on automatic vectorization offered by modern compilers. Intrinsic functions require low-level hardware knowledge, often result in complex code and are specific to the underlying instruction set, impeding code deployments on arbitrary platforms. Although compiler-based automatic vectorization has substantially improved over the last years, manually-tuned intrinsics code is still more efficient and remains the number one choice when

¹⁷Intel®Performance Counter Monitor - A better way to measure CPU utilization, <https://www.intel.com/software/pcm>, Last access: August 29, 2018.

¹⁸Compilers, like GCC, provide highly-optimized implementations for low-level tasks, e.g., vector processing, or floating-point operations, in the form of built-in intrinsic functions.

2 Fundamentals

desiring high performance [Sprenger et al. 2018c; Pohl et al. 2016]. Thus, the remaining work uses SIMD intrinsics.

SIMD instructions have multiple requirements that must be met to obtain speed-ups:

- In general, SIMD parallelism prefers simple memory layouts that keep data elements, which should be simultaneously processed, in consecutive memory locations. Otherwise multiple load or store operations are needed to fill or read an entire SIMD register. Although recent SIMD instruction sets feature gather and scatter operations [Polychroniou et al. 2015], which can read from or write to non-consecutive memory locations, these are limited to a small number of use cases and usually show inferior performance than when working on consecutively-stored data, especially if the to-be-processed data elements are part of different cache lines [Hofmann et al. 2014], as each cache line needs to be loaded individually.
- SIMD instructions are restricted to simple control flows with few, if any, conditional branches. Batch-style algorithms, like scans, benefit the most from vectorization.
- SIMD instructions are limited to simple data types. For instance, AVX intrinsics support only integers, floats and doubles¹⁹.

Depending on the memory layout and the search algorithms, SIMD instructions can accelerate the search operators of main-memory index structures. For instance, Zeuch et al. [Zeuch et al. 2014] proposed in-memory B-tree variants that use vectorized instructions to navigate the inner tree nodes.

2.3.3 Multi-Core CPUs and Simultaneous Multithreading (SMT)

Until the mid 2000s, most processors featured only one single processing unit. Manufacturers improved the computing power by placing a growing number of transistors on the die and by increasing the clock rate of the CPU. However, at a certain point, the advancements in clock rates slowed down due to physical limitations, which forced processor vendors to consider alternative means to improve the performance. To this end, modern CPUs employ parallelization: They install multiple almost-independent²⁰ processing units, named *cores*, on the same chip. Cores can concurrently execute processes and possess individual instruction pipelines and caches.

Figure 2.5 illustrates the evolution of server processors in terms of cores per CPU and clock rate²¹, using the Intel Xeon family as example. Note the beginning of stagnating clock rates in 2005 and the concurrent introduction of multi-core CPUs as means to further improve the performance. Another interesting point in time is marked by the availability of the Intel Xeon Phi many-core CPU, which targets traditional application areas of highly-parallel GPUs, as stand-alone processor in 2016.

¹⁹Intel Intrinsics Guide, <https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=AVX>, Last access: August 29, 2018.

²⁰Most multi-core CPUs feature a large LLC shared by all cores (see Figure 2.4a).

²¹Source: https://en.wikipedia.org/wiki/List_of_Intel_Xeon_microprocessors and https://en.wikipedia.org/wiki/Xeon_Phi, Last access: August 29, 2018.

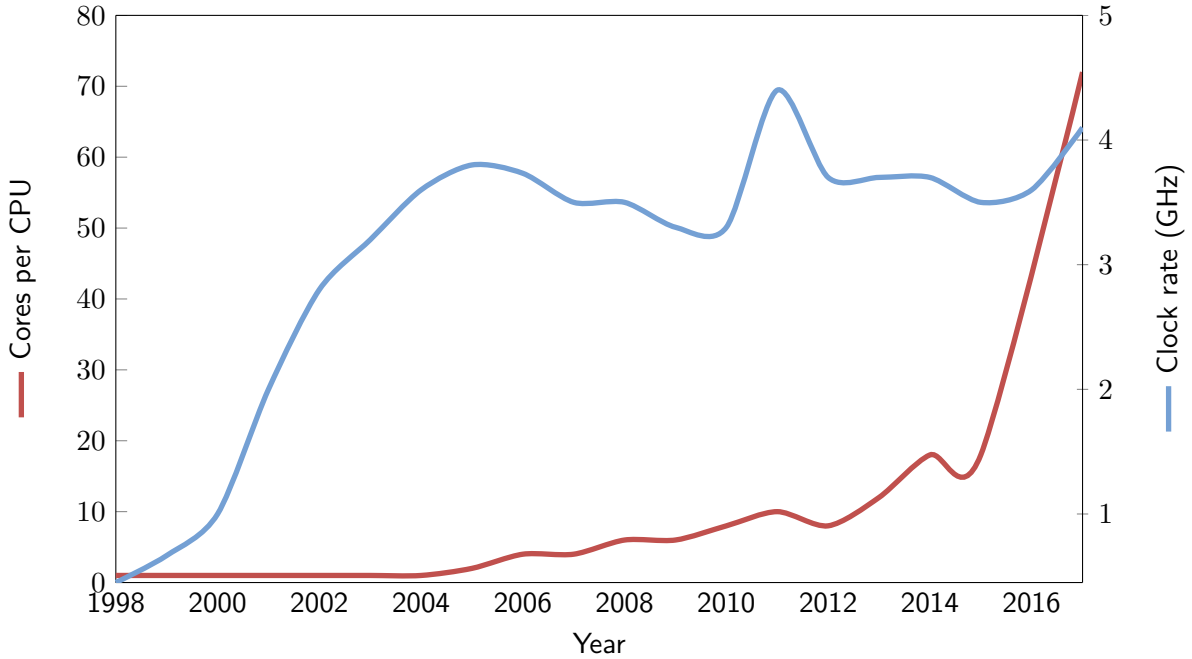


Figure 2.5: Evolution of Intel Xeon server CPUs (Source: https://en.wikipedia.org/wiki/List_of_Intel_Xeon_microprocessors and https://en.wikipedia.org/wiki/Xeon_Phi, Last accesses: February 25, 2019).

In addition to multi-core processing, most current processors support *simultaneous multi-threading* (SMT) that enables to handle multiple independent processes in the same pipeline. Of course, *physical CPU cores* can still execute only one instruction at the same time. However, processors are able to dynamically switch between different contexts, which is useful when processes stall, like in the case of unavailable resources. For instance, while a blocking process waits for data to be loaded into the CPU caches, another process can be executed, which prevents wasting the computing power of the CPU. SMT is mainly beneficial for frequently-blocking, *memory-bound* processes. Recent Intel CPUs feature a proprietary implementation of SMT, named hyperthreading. Typically, they allow to simultaneously run two *hyperthreads*, also called *virtual CPU cores*, on one physical core.

In multi-socket machines, multiple (multi-core) CPUs share the same memory bus when accessing the main memory. While multi-channel memory may reduce the pressure on the memory bus, they cannot hold up with the increasing parallel capabilities of modern CPUs. *Non-uniform memory access* (NUMA) provides means to cope with such highly-parallel settings. It divides all cores of a machine into NUMA groups, of which each group is assigned to a separate, *local* region of the memory. In NUMA, processes can access data stored in the local memory region very fast, whereas accesses to non-local regions, associated with other NUMA groups, are slower.

While increasing clock rates of single-core CPUs enabled performance boosts without requiring changes to the software, nowadays programs must explicitly control the processing units of multi-core CPUs to fully utilize the available computing capabilities. Multi-core CPUs require

2 Fundamentals

algorithms to divide (large) problems into independent subproblems that can be processed by distinct threads in parallel, while requiring as little synchronization as possible. Although many algorithms, like scans, can be easily parallelized, obtaining the perfect DOP is not always possible and remains a challenge.

Database systems can accelerate search operations with multithreading using two approaches: inter- and intra-query parallelism. Inter-query parallelism increases the throughput of large batches of queries by executing multiple queries in parallel. Each query is processed by a separate thread. Hence, it is best suited for online transaction processing, where queries are simple and can typically be answered within a very short amount of time. In contrast, intra-query parallelism improves the performance of one particular query by processing it with multiple threads. It is mainly beneficial for complex, long-running queries, like online analytical processing.

In the past, database systems have favored inter-query parallelism [Graefe 1990; Graefe 1994], because server machines featured only few threads reducing the benefits of parallelizing individual queries. Furthermore, inter-query parallelism allows to re-use single-threaded implementations.

When taking the growing parallel capabilities of modern CPUs into account, which enable settings with hundreds to thousands of hardware threads on one machine²², intra-query parallelism becomes increasingly important to avoid wasting computing resources. Furthermore, with the rise of big data analytics [Labrinidis et al. 2012; Hao Zhang et al. 2015], many database workloads are long running and read heavy, and would strongly benefit from intra-query parallelism to reduce their execution times. As a consequence, modern index structures must explicitly leverage multi-core CPUs and provide parallel search algorithms that can use multiple threads to navigate the index.

2.4 Genomic Multidimensional Range Query Benchmark (GMRQB)

The strength of any empirical evaluation of index structures critically depends on the representativeness of the data and the queries used. The performance of MDIS may strongly vary for different data distributions, e.g., uniform, unimodal or multimodal clustered, clustered in subspaces, etc., and different query workloads, e.g., hot-spot regions, partial- or complete-match queries, low or high selectivities, etc. In the past, MDIS were mainly evaluated with synthetic data or synthetic workloads, such as in [K. Kim et al. 2001], [Pagel et al. 1993], or [Wang et al. 2016], with the obvious advantage of being able to influence many parameters of the data and the workloads. Although some previous evaluations have used real-world data sets, to the best of our knowledge, none of them has considered any real-world query workloads. Therefore, in addition to experiments with synthetic data and workloads, we strive to also evaluate all considered methods on real-world multidimensional data and workloads.

To this end, we create a novel multidimensional range query benchmark derived from the analysis of genomic data. As motivated in the Introduction, genomic analysis heavily relies on MDRQ to interactively study the variant profiles of complete disease cohorts and entire populations. It represents a very interesting use case from a data management perspective,

²²Current many-core CPUs can feature more than seventy cores. When adding four-way hyperthreading and using four- or eight-socket settings, more than thousand threads per machine are possible.

2.4 Genomic Multidimensional Range Query Benchmark (GMRQB)

because continuous improvements in sequencing technology lead to an ever-growing amount of genomic data available for research and analysis.

This section presents the *Genomic Multidimensional Range Query Benchmark* (GMRQB), which consists of eight parameterized, partial-match and complete-match, realistic range queries applied to a large set of real-world genomic variants derived from the 1000 Genomes Project [The 1000 Genomes Project Consortium 2015]. Data points are of moderate dimensionality (19 dimensions) and dimensions feature very different numbers of distinct values. The used data set is publicly available²³, which allows reproduction of our evaluation results and facilitates further research on range queries.

2.4.1 Range Queries on Genomic Variant Data

A human genome consists of approximately three Billion base pairs (the DNA) structured in 23 chromosomes. When sequencing a human, i.e., experimentally determining its genome, these three Billion base pairs are typically compared to a human reference genome, which models a hypothetical *normal* human genome [International Human Genome Sequencing Consortium 2004]. Deviations from this reference are typically called genomic variants [The 1000 Genomes Project Consortium 2012], or mutations if they affect the human in some negative sense, e.g., when a certain variant is associated with an increased risk for developing a certain disease. On average, every human genome contains approximately four to five Million variants [The 1000 Genomes Project Consortium 2015]. Genomic variants are not distributed at random over the genome, but certain genomic regions are more prone to contain variants than others. Within the large field of genomic analysis, multiple application areas make use of MDRQ. In the following, we describe two of them: precision medicine and genome browsers.

- The premise of *precision medicine*²⁴ is to correlate an individual’s variant profile to his or her susceptibility to diseases and treatments [Lievre et al. 2006]. An important part of research in precision medicine is concerned with collecting large numbers of genomes together with medical information about the individuals to integrate these different data in databases. Using these databases, researchers routinely perform statistical analysis of genomic variant profiles regarding commonalities and differences between individuals with respect to health-related issues. For instance, researchers search for sets of variants sharing certain characteristics, e.g., belonging to the same genomic region, being present in the same class of diseases or a similar group of patients, being present in patients reacting in the same way to medication, etc. These searches eventually boil down to MDRQ.
- Another application area of range queries in the context of genomic analysis are *genome browsers*. Resembling the concepts of web browsers, genome browsers can be used to interactively scroll over the variant profiles of complete genomes, region by region. Figure 2.6 shows a screenshot taken from a popular genome browser, the *integrative genomics viewer* (IGV) [Thorvaldsdóttir et al. 2013]. In this screenshot, IGV visualizes all variants of selected genomes found on chromosome 22 between positions 30,927,764 and 30,936,369.

²³Data | 1000 Genomes, <http://www.internationalgenome.org/data>, Last access: August 29, 2018.

²⁴See, for instance, <https://allofus.nih.gov>. Last access: August 29, 2018.

2 Fundamentals

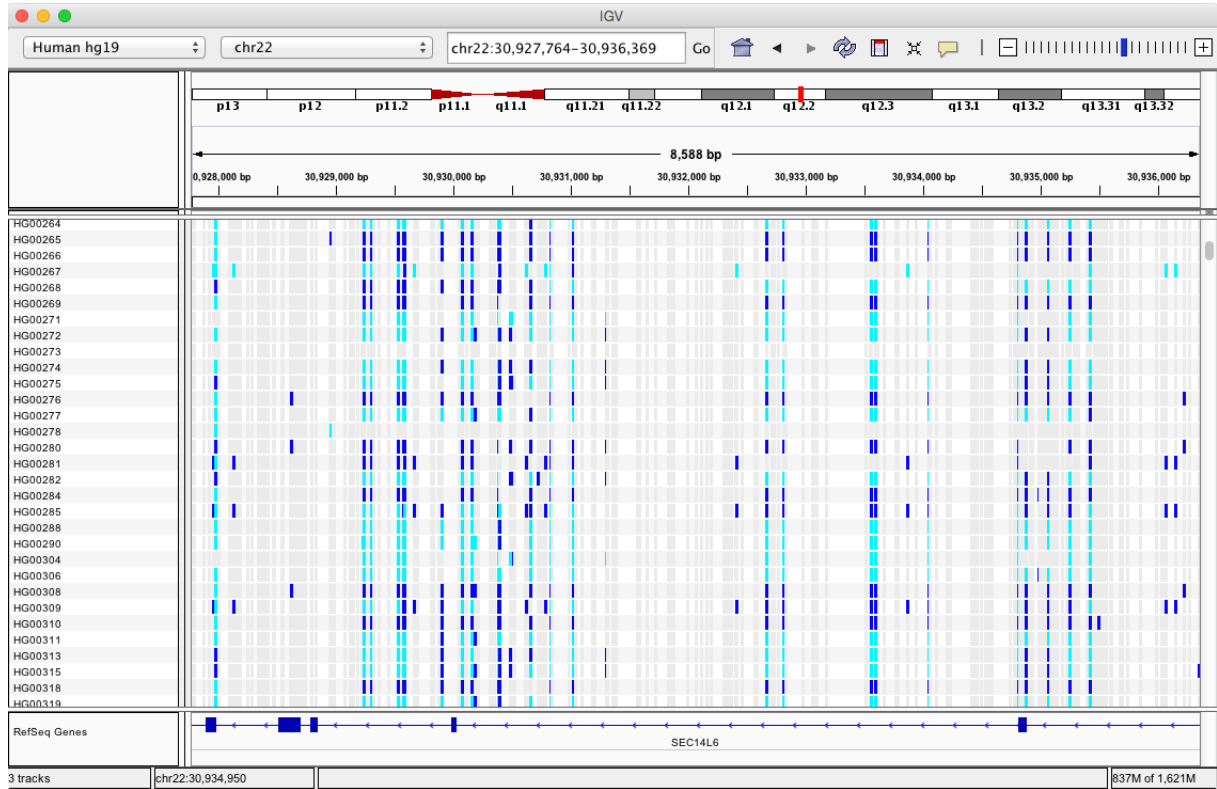


Figure 2.6: A screenshot taken from a genome browser, the integrative genomics viewer [Thorvaldsdóttir et al. 2013].

Different shades of blue highlight genomic regions containing variants. Technically, genome browsers use range queries to narrow down large sets of variants to the subsets relevant for the visualized genomic region [H. Li 2011]. Genome browsers pose especially strong requirements on the latency of range queries, because they must provide an interactive interface to the application user.

2.4.2 Real-World Data Set

The 1000 Genomes Project has sequenced the entire genomes of 2,504 human individuals, also called *samples*, from across the world to facilitate research in precision medicine and related areas [The 1000 Genomes Project Consortium 2015]. The resulting data set is publicly available and contains 84.4 Million variants grouped by individual and by genomic location. In addition to sequencing data, the 1000 Genomes Project also provides meta data of the individuals, e. g., gender, and population.

For the GMRQ Benchmark, we integrate the genomic variant data with the meta data of the samples. We obtain a data set²⁵ featuring 19 dimensions:

²⁵Our benchmark data set can be considered as a cross product between both data sources.

2.4 Genomic Multidimensional Range Query Benchmark (GMRQB)

- *Chromosome* holds the chromosome that the variant was found at, e. g., 22.
- *Location* provides the according position within the chromosome, e. g., 100,000.
- *Quality* resembles a quality score of the sequencing read, e. g., 1.0, providing information about the accuracy of the used sequencing platform.
- *Depth* denotes the number of unique sequencing reads²⁶, e. g., 30.
- *Reference_genome* holds the reference genome that was compared with the sample genome to determine the genomic variant, a procedure called variant calling [Hwang et al. 2015].
- *Variant_id* provides the unique identifier of the variant.
- *Allele_freq* describes the relative frequency of the variant within the population, e. g., 1%.
- Similarly, *allele_count* provides the total number of occurrences of the variant within the population, e. g., 100.
- *Ref_base* holds the nucleobase of the reference genome at the position of the variant, e. g., A.
- *Alt_base* provides the variant found in the sample genome, e. g., C. Besides *single-nucleotide polymorphisms* (SNP), the data set also includes *copy-number variations* (CNV) and *genomic insertions and deletions* (INDEL). Therefore, this dimension contains more distinct values than the number of possible nucleobases (four).
- *Ancestral_allele* provides allele data from organisms very close to that of humans. The 1000 Genomes Project integrates *Ensembl Compara*²⁷, which provides the alleles from different primates.
- *Variant_type* denotes the type of the variant, e. g., SNP, CNV, or INDEL.
- *Sample_id* provides the unique identifier of the sample. When combining *variant_id* and *sample_id*, we can obtain an unique tuple identifier.
- *Gender* provides the gender of the individual. It holds two different values (male and female) and is therefore solely queried with point queries (or range queries with identical lower and upper boundaries).
- *Population* holds the population of the individual, e. g., Finland.
- *Genotype* provides genotype information, i. e., the two alleles carried by the sample genome.
- The 1000 Genomes Project has sequenced the genomes of multiple individuals that belong to the same family. For such samples, *family_id* provides an unique identifier of the family the individual belongs to.

²⁶Next-generation sequencing technologies read each DNA base multiple times.

²⁷Comparative Genomics, <http://www.ensembl.org/info/genome/compara/index.html>, Last access: August 29, 2018.

2 Fundamentals

Attribute	Domain	Distinct Values
chromosome	[22,22]	1
location	[16050100,18791500]	20,550
quality	[100,100]	1
depth	[301,91949]	19,538
reference_genome	[398393985191641088,398393985191641088]	1
variant_id	[1225,99992304]	63,883
allele_freq	[0.0002,0.9998]	3,176
ref_base	[25855900690415616,18445699537663164416]	540
alt_base	[66030698359685120,18446499982128185344]	423
ancestral_allele	[1476799971876405248,8591840045650935808]	3
ancestral_count	[1,5008]	3,177
filter	[4774679821152157696,4774679821152157696]	1
variant_type	[310693982822727680,17548400192863076352]	3
sample_id	[961370,99848200]	2,502
gender	[1,2]	2
family_id	[11118,94755104]	1,867
population	[390559002771062784,17293700523312021504]	26
relationship	[184136999010041856,17976799609858555904]	16
genotype	[107920004023844864,18388100521530490880]	29

Table 2.1: The set of genomic variants used in this thesis consisting of ten Million tuples.

- Finally, *relationship* contains information about the role of the individual within the family, if present, e. g., father, or child.

The 1000 Genomes Project provides variant data as *Variant Call Format* (VCF) files [Danecek et al. 2011]. VCF is a text-based, tab-separated file format designed for the outcomes of variant calling. As most MDIS store tuples as arrays of floats, we need to transform the data before indexing. For numeric values, such a transformation is straightforward. However, attributes originally provided as strings, i. e., *reference_genome*, *ref_base*, *alt_base*, *ancestral_allele*, *filter*, *variant_type*, *population*, *relationship*, and *genotype*, are more challenging. We transform these attributes into floating-point values by hashing. Unfortunately, this transformation makes range queries on these dimensions less meaningful.

Taking into account that our evaluation machine has only 32 *gigabytes* (GB) of main memory and MDIS require some indexing overhead, we limit the data used in the experiments of this thesis to ten Million tuples, which equals 724.79MB when stored with four-byte floats. These tuples were extracted from the sequencing data for chromosome 22. Table 2.1 describes the resulting data set. For each dimension, it provides the domain as a range of real numbers and the number of contained distinct values.

2.4.3 Realistic Range Query Templates

In collaboration with Bioinformaticians²⁸, we designed the *Genomic Multidimensional Range Query Benchmark* (GMRQB), a workload of eight realistic complete- and partial-match range query templates, which are applied to the data set described in Section 2.4.2. In addition to the eight query templates, we also provide a mixed workload consisting of all templates randomly mixed together, which enables experiments with changing workload patterns. These range queries retrieve specific subsets of genomic variants interesting for further analysis. They specify range predicates on some, most, or all dimensions. Depending on the dimension, predicates may either specify single points or ranges of different sizes. For instance, we always use a point query for the dimension gender, yet always apply range predicates to the dimension location. All queries of GMRQB restrict the genomic location, i. e., attributes chromosome and location.

Queries in the workload are templates that have to be instantiated with meaningful values. For the genomic location, we use the RefSeq database²⁹ to align genomic ranges to coding regions. All other variables are filled using randomly-selected values found in the original data. Listing 2.1 shows Query Template 3 from GMRQB written as a SQL statement to ease readability³⁰. Like all query templates, it restricts the search to a certain genomic region defined by a chromosome and a location range. Additionally, it retrieves only variants that were found in an individual of a particular gender.

Listing 2.1: Query Template 3 of GMRQB.

```
1 SELECT * FROM variants
2 WHERE chromosome BETWEEN ? AND ?
3 AND location BETWEEN ? AND ?
4 AND gender = ?;
```

For each query template, Table 2.2 shows the average selectivity and the average number of queried dimensions. Except Query Template 8, all query templates are partial-match queries. For completeness, Appendix A provides all query templates.

The used data set has three disadvantages with regards to the evaluation of MDRQ:

- We had to use hashing to transform attributes originally stored as strings into floating-point values. Unfortunately, such transformations prevent meaningful range queries on these dimensions.
- Some dimensions, e. g., gender, or reference genome, have only very few distinct values. Actually, range queries on these dimensions turn into point queries.
- Although the queries of the benchmark resemble a real-world interactive analysis of genomic variant data, these queries were not extracted from real applications. It would be

²⁸We would like to thank the bioinformaticians from our working group, especially Yvonne Lichtblau, for their valuable feedback on the design of the GMRQ Benchmark.

²⁹RefSeq: NCBI Reference Sequence Database, <https://www.ncbi.nlm.nih.gov/refseq/>, Last access: August 29, 2018.

³⁰Implementations of MDIS typically require multidimensional range queries to be specified as two vectors, where the first (second) vector denotes the lower (upper) boundary of the range query.

2 Fundamentals

GMRQB Query Template	Average Selectivity	Average Number of Queried Dimensions
Query Template 1	10.76% ($\sigma = 7.24\%$)	2 ($\sigma = 0.0$)
Query Template 2	2.19% ($\sigma = 2.27\%$)	5 ($\sigma = 0.0$)
Query Template 3	5.36% ($\sigma = 3.61\%$)	3 ($\sigma = 0.0$)
Query Template 4	0.22% ($\sigma = 0.15\%$)	4 ($\sigma = 0.0$)
Query Template 5	0.20% ($\sigma = 0.15\%$)	5 ($\sigma = 0.0$)
Query Template 6	0.11% ($\sigma = 0.11\%$)	6 ($\sigma = 0.0$)
Query Template 7	0.05% ($\sigma = 0.06\%$)	7 ($\sigma = 0.0$)
Query Template 8	0.00001% ($\sigma = 0.00002\%$)	19 ($\sigma = 0.0$)
Mixed Workload	1.58% ($\sigma = 3.58\%$)	5.81 ($\sigma = 4.11$)

Table 2.2: The query templates of the GMRQB.

very interesting to monitor workloads from researchers in, for instance, precision medicine and add these queries to GMRQB.

3 CSSL: Processing One-Dimensional Range Queries in Main Memory

This chapter addresses one-dimensional main-memory index structures, of which many have been proposed over the last years, e. g., the *adaptive radix tree* (ART) [Leis et al. 2013], the *fast architecture sensitive search tree* (FAST) [C. Kim et al. 2010], or the *cache-sensitive B⁺-tree* (CSB⁺-tree) [Rao et al. 2000]. These are typically based on the concepts of traditional index structures, e. g., B-trees [Bayer et al. 1972], radix trees [Morrison 1968], or hash tables [Garcia-Molina et al. 2000], but adapt them to the needs of main-memory settings. Like disk-based index structures, which optimize data transfers between external and main memory, in-memory index structures aim to work as much as possible on data held in higher, faster levels of the memory hierarchy when evaluating search queries, which boils down to optimizing CPU cache misses. Such optimizations are motivated by analyses of Ailamaki et al. [Ailamaki et al. 1999], which identified LLC misses as one of the major contributors to the runtimes of database workloads on modern hardware.

Existing in-memory index structures mainly focus on achieving high lookup performance, but neglect range queries, despite their numerous applications and use cases (see Introduction). While most hash tables obviously lack pruning capabilities for range queries anyway, because they do not store data in a sorted order, also many in-memory tree variants, such as ART or CSB⁺, show poor search efficiency when executing range queries. Search trees keep data in a sorted order and implement range queries by looking up the smallest matching element and iterating over all consecutive elements until a mismatch occurs. Most in-memory approaches optimize the first step of range queries, typically implemented as a lookup operation, but neglect the second step, which often requires chasing many pointers with random accesses.

The major challenge to an efficient in-memory execution of range queries, especially for queries with a moderate or a low selectivity, are random data accesses that induce cache misses and lead to CPU stalls [Ailamaki et al. 1999]. Taking this observation into account, it is not surprising that, in main memory, sequential full-table scans outperform tree-based index structures for range queries with selectivities of approximately 1% or larger [Das et al. 2015].

In this chapter, we present the *cache-sensitive skip lists* (CSSL) as a novel main-memory index structure based on conventional skip lists [Pugh 1990; Munro et al. 1992]. CSSL employ a specific memory layout to take maximal advantage of the features of modern CPUs, e. g., multi-level cache hierarchies, SIMD instructions, and pipelined execution. They store data such that the range query operator can almost-sequentially traverse over matching elements, which exploits cache line prefetching and strongly reduces CPU cache and TLB misses. Moreover, the used memory layout enables a vectorization of the range query algorithm.

The contributions of this chapter are as follows:

- We propose the cache-sensitive skip list, a main-memory index structure offering efficient execution of range queries.
- We show how to apply SIMD instructions to the range query operator of skip lists.
- We compare CSSL with other main-memory index structures using different workloads on synthetic and real-world one-dimensional data sets.

The remainder of this chapter is organized as follows. In Section 3.1, we present work related to CSSL. Section 3.2 introduces skip lists, the index structure that CSSL are based on. Section 3.3 presents the foundational concepts behind CSSL and describes algorithms for executing lookups and range queries; we also show how to process updates. Section 3.4 compares CSSL with state-of-the-art main-memory index structures and Section 3.5 summarizes this chapter.

Parts of this chapter have been previously published in [Sprenger et al. 2016].

3.1 Related Work

Although concepts like cache-aligned data layouts, index traversal with SIMD instructions, and pointer elimination have been investigated before [C. Kim et al. 2010; Rao et al. 1999; Rao et al. 2000], to the best of our knowledge, we are the first to combine these to accelerate range queries in one-dimensional main-memory index structures.

Skip lists [Pugh 1990] were proposed as a probabilistic alternative to B-trees [Bayer et al. 1972]. They have been applied to multiple areas and have been adapted to different purposes, e.g., lock-free skip lists [Fomitchev et al. 2004], deterministic skip lists [Munro et al. 1992], or concurrent skip lists [Herlihy et al. 2006]. In [Xie et al. 2016], Xie et al. present a parallel skip list-based main-memory index, named PI, that processes batches of queries using multiple threads. Skip lists are not only of interest for researchers, but also part of several modern database management systems. The main-memory database system MemSQL [Chen et al. 2016] uses skip lists to implement secondary indexes¹ and the key-value store Redis [Carlson 2013] employs them to manage *sorted sets*². CSSL are based on deterministic skip lists [Munro et al. 1992], but employ a cache-friendly data layout tailored to modern CPUs and beneficial for the execution of range queries.

There are several other approaches addressing in-memory indexing [M. Böhm et al. 2011; C. Kim et al. 2010; Kissinger et al. 2012; Leis et al. 2013; Rao et al. 1999; Rao et al. 2000], yet few specifically target range queries. *Cache-sensitive search trees* (CSS-trees) [Rao et al. 1999] build a tree-based dictionary on top of a sorted array that is tailored to the properties of the cache hierarchy, e.g., the sizes of the cache lines, and can be searched in logarithmic time. CSS-trees are static by design and need to be completely rebuilt when ingesting updates. Rao and Ross [Rao et al. 2000] introduce the CSB⁺-tree, a cache-conscious B⁺-tree [Comer 1979], which minimizes pointer usage and reduces space consumption. As shown in Section 3.4, CSSL outperform CSB⁺-trees significantly for all considered workloads.

¹The Story Behind MemSQL's Skiplist Indexes - MemSQL Blog, <http://blog.memsql.com/the-story-behind-memsqls-skiplist-indexes/>, Last access: August 29, 2018.

²An introduction to Redis data types and abstractions - Redis, <https://redis.io/topics/data-types-intro>, Last access: August 29, 2018.

Masstree [Mao et al. 2012] is an in-memory database that employs a trie of B^+ -trees as index structure. It supports keys of arbitrary length, which is useful when indexing strings. We do not include Masstree in our evaluation, because its implementation is multithreaded, which prevents a fair comparison. Instead, we consider an in-memory implementation of its base index structure, the B^+ -tree, as competitor.

Zhang et al. [Huanchen Zhang et al. 2016] introduce a hybrid two-stage index that can be built on top of existing index structures, like B-trees or skip lists. Interestingly, they also propose a paged-based skip list implementation as example that is tailored to main memory. However, in contrast to CSSL, it is completely static by design and does not exploit SIMD instructions.

ART [Leis et al. 2013] is a main-memory index structure based on radix trees. It employs adaptive node sizes and makes use of advanced CPU features, like SIMD instructions, to enhance the search performance. While ART achieves high lookup performance currently only superseded by hash tables [Alvarez et al. 2015], its support for range queries is much less efficient since these require traversing over the tree by chasing pointers. As shown in the evaluation, CSSL significantly outperform ART for range queries. We assume that the results of our comparison between CSSL and ART would carry over to other index structures based on prefix trees, such as generalized prefix trees [M. Böhm et al. 2011] or KISS-trees [Kissinger et al. 2012].

Another recent data structure is FAST [C. Kim et al. 2010], a binary search tree tuned to the underlying hardware by taking architecture parameters, like page and cache line sizes, into account. It offers both thread- and data-level parallelism, the latter by using SIMD instructions. Similar to CSSL, FAST does not need to access pointers when traversing the tree structure. However, FAST is optimized for lookup queries only, where it is clearly outperformed by ART [Leis et al. 2013]. Therefore, we do not include it in our evaluation.

Schlegel et al. showed how to linearize k-ary search trees [Schlegel et al. 2009]. We use similar techniques to adapt the fast lanes of CSSL to main memory.

3.2 Conventional Skip Lists

Skip lists are a probabilistic data structure similar to B-trees [Pugh 1990]. Skip lists consist of multiple lanes of keys organized in a hierarchical fashion. At the highest level of granularity, a skip list contains a linked list of all keys in a sorted order. On top of this *data list*, skip lists maintain *fast lanes* at different levels. A fast lane at level i contains $n * p^i$ elements on average, where n is the number of stored keys and $0 < p < 1$ is an input parameter. Originally, skip lists are probabilistic data structures, because elements stored in higher lanes are randomly chosen from those at lower lanes: Every element of fast lane i appears in fast lane $i + 1$ with probability p . This scheme allows for efficient updates and inserts, yet makes the data structure less predictable.

In our work, we use *perfectly balanced skip lists* [Munro et al. 1992], a deterministic variant of skip lists. In perfectly balanced skip lists, the fast lane at level $i + 1$ contains every $1/p$ 'th element of the fast lane at level i . Accordingly, for $p = 0.5$, the fast lane at level $i + 1$ contains every second element of level i , in which case a skip list resembles a balanced binary search tree. Figure 3.1 illustrates a balanced skip list over nine integer keys with two fast lanes for $p = 0.5$.

When executing lookups, the fast lanes are used to narrow down the segment of the data list

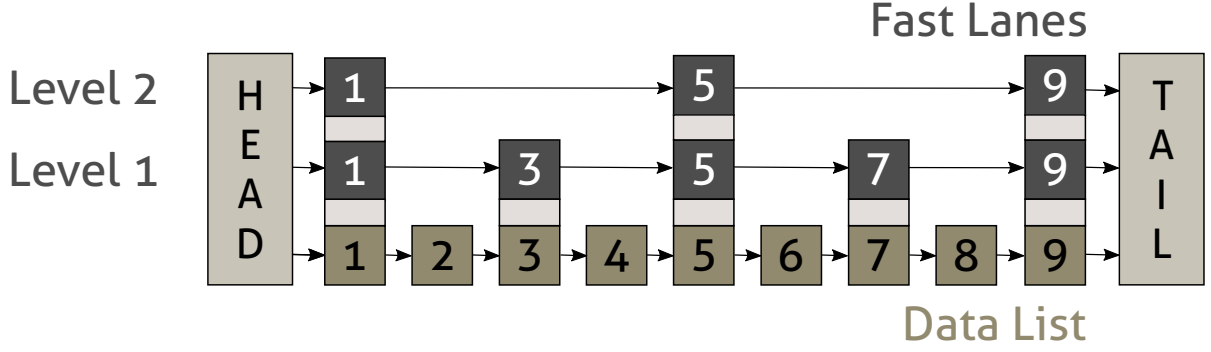


Figure 3.1: A balanced skip list that manages nine keys with two fast lanes. Each fast lane skips over two elements ($p = 1/2$).

that may contain the searched element, which effectively prunes most parts of the data. For instance, a search for key 6 would traverse the skip list shown in Figure 3.1 as follows. First, the search algorithm obtains the first element of the highest fast lane at level 2 by using the head element, which holds pointers to the first elements of each fast lane and the data list. Second, the search traverses the fast lane element by element until an element is either equal to the searched element, in which case the search can terminate, or greater than the searched key, in which case the search continues with the next lower level. Here, the search stops at element 5 and moves down to the next fast lane at level 1. Fourth, steps two and three are recursively repeated until the data list is reached. Fifth, the data list is scanned until the searched element is found or proven to not exist. In this example, the search operator returns after scanning elements 5 and 6 at the data list.

Input parameter p influences the density of the fast lanes. A small p value leads to *sparse fast lanes* skipping over many elements, while a large p value induces *dense fast lanes* skipping over only few elements. A fully-built balanced skip list for n keys contains $\log_{1/p}(n)$ fast lanes and requires $1/p$ comparisons per fast lane. Assuming that p is a constant value, searching for a certain key requires $\log_{1/p}(n) * 1/p = O(\log(n))$ key comparisons in the worst case.

Besides single-key lookups, skip lists also support range queries very efficiently. Since the data list and the fast lanes are kept in a sorted order, implementing a range query requires only two steps: (1) Find the smallest element satisfying the queried range, which is similar to a lookup, and (2) collect all subsequent elements until a mismatch occurs. We have chosen skip lists as basic index structure for multiple reasons:

- Skip lists store data in a sorted order, which facilitates range queries.
- At all levels (or fast lanes) of a skip list, consecutive elements can be accessed in constant time. As described in Section 3.3.2, we exploit this property in our implementation: Range queries are evaluated using fast lanes, which effectively saves key comparisons.
- Skip lists are main-memory index structures by design, because they do not tailor nodes to disk blocks, as for instance B⁺-trees.

Listing 3.1: Fat nodes.

```

1 struct FatNode {
2     KEY_TYPE key;
3     std::string value;
4     FatNode* forward[MAX_HEIGHT];
5 };

```

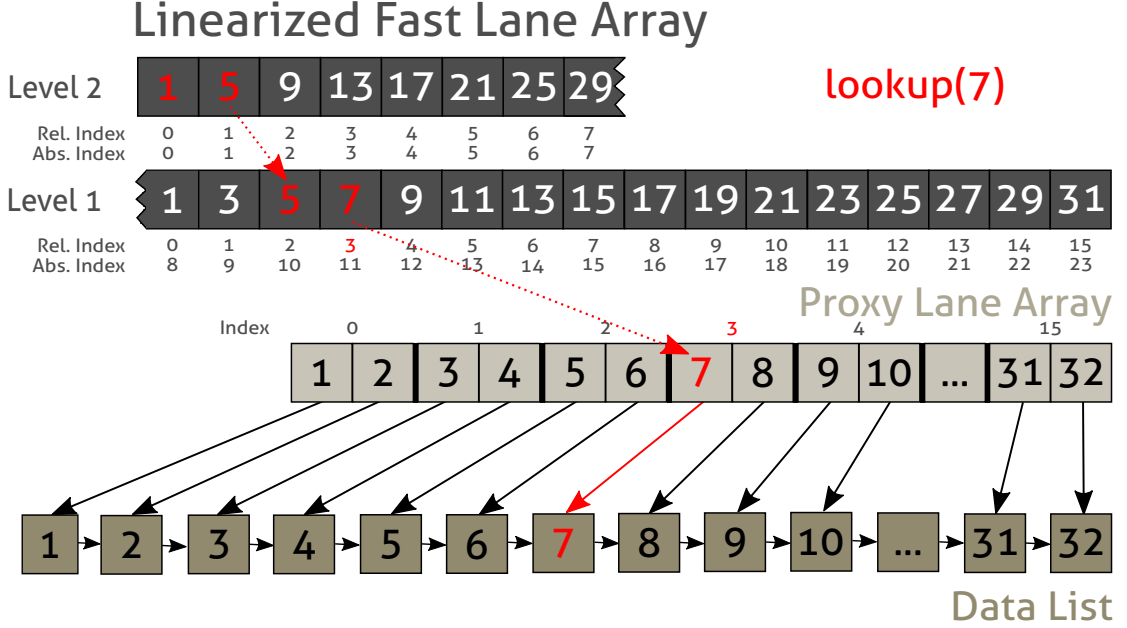
- The implementation of a skip list is more simple in terms of code complexity than that of other index structures. For instance, the developers behind MemSQL report that their skip list implementation requires fifty times less lines of code than a typical B-tree implementation³. As described by Boncz et al. [Boncz et al. 2009], simple implementations potentially allow sequential access patterns and facilitate various adaptations, e. g., optimizations for cache efficiency.
- Although our focus lies on range queries, skip lists provide very efficient update operations, because they do not require to deal with node overflows, as, for instance, a B-tree.

In the original paper [Pugh 1990], skip lists are implemented using *fat nodes*⁴ (see Listing 3.1), also called *towers*. A fat node is a record that contains a key, a value and an array holding pointers to the subsequent element for every fast lane and the data list. The advantage of this approach is that all nodes are uniform, which simplifies the implementation. Furthermore, if a key is found in an upper lane, the search can terminate as all instances of a key are kept in the same record. On the other hand, such an implementation wastes memory space, because it reserves space for $O(n * m)$ pointers (given that m denotes the number of fast lane levels), although most values in higher levels are padded with *NULL*.

Searching in skip lists using fat nodes involves chasing many pointers. Pointer-heavy memory layouts are suboptimal on modern CPUs, because they require search algorithms to jump between non-contiguous parts of the allocated memory. Even when only searching the data list, cache line utilization is suboptimal due to the fatness of keys. For instance, in a skip list that stores four-byte integer keys and maintains five fast lanes on top of the data list, each node requires $4 \text{ bytes} + 6 * 8 \text{ bytes} = 52 \text{ bytes}$ of memory on a 64-bit CPU architecture (using eight-byte pointers). Given that a cache line typically holds 64 bytes on current CPU architectures, each navigation step fills almost an entire cache line although only a small part of it is used. Usually, a traversal step just needs the key and one pointer to retrieve the subsequent element on a certain fast lane, i.e., $4 \text{ bytes} + 8 \text{ bytes} = 12 \text{ bytes}$, which equals a cache line utilization of $12/64 = 18.75\%$.

³The Story Behind MemSQL's Skiplist Indexes - MemSQL Blog, <http://blog.memsql.com/the-story-behind-memsqls-skiplist-indexes/>, Last access: August 29, 2018.

⁴In practice, similar implementations are used. For instance, see the in-memory database system Redis: <https://github.com/antirez/redis/blob/unstable/src/server.h#L795>; Last access: August 29, 2018.

Figure 3.2: A CSSL managing 32 keys with two fast lanes ($p = 1/2$).

3.3 Cache-Sensitive Skip Lists (CSSL)

We present *cache-sensitive skip lists* (CSSL) as an alternative implementation for (deterministic) balanced skip lists [Munro et al. 1992] that use a radically different memory layout leading to a much higher search and space efficiency when deployed on current CPU architectures. The first and most obvious idea is to linearize the fast lanes in a breadth-first order and manage them as separate entities in a dedicated array, called *linearized fast lane array*. When navigating this array, we can compute the positions of follow-up elements based on the current position and parameter p , making pointers superfluous. Figure 3.2 illustrates a CSSL indexing every integer from 1 to 32 with two fast lanes ($p = 1/2$). The red arrows visualize the traversal path, which the search algorithm would take to find key 7.

3.3.1 Memory Layout

Since CSSL are based on balanced skip lists, given the number of keys (n) and the density of the fast lanes (p), we can exactly predict the structure of the according fast lane hierarchy. CSSL take that into account and allocate a certain amount of memory space for the linearized fast lane array.

The underlying *data list*, which holds all data elements, is implemented as a linked list to support updates. We introduce the *proxy lane* to connect the static fast lane array with the dynamic data list. For each key of the lowest fast lane, the proxy lane maintains a pointer to the corresponding object of the data list. Connections are implicit: The connection to the i -th fast lane element can be found at index $i - 1$ of the proxy lane.

The memory layout of CSSL offers five main advantages when deployed on modern processors:

- **Better Cache Line Utilization:** The navigation of linearized fast lanes utilizes cache lines better than the evaluation of fat nodes. We can always make use of entire cache lines until we jump to a lower fast lane layer, which reduces the amount of data transferred through the cache hierarchy. In the case of four-byte keys, 16 fast lane elements fit into one 64-byte cache line, while a cache line of the same size can hold only one fat node of a conventional skip list (assuming that the skip list contains more than two fast lanes). CSSL have to access $((1/p)/16) * m$ cache lines when traversing over the m fast lanes, whereas regular skip lists need to read up to $1/p * m$ cache lines.
- **Reduced Cache Misses:** When navigating a fast lane, the search algorithm has to compare up to $1/p$ elements with the search key. In CSSL, after the first element of a fast lane has been loaded, the search algorithm accesses only consecutive memory locations, which minimizes the number of CPU cache misses. In contrast, in a regular, pointer-based skip list implementation using fat nodes, every comparison of a fast lane element with the search key requires a random access leading to up to $1/p$ cache misses per fast lane.
- **No Pointer Chasing:** Pointer-heavy data structures, like linked lists, require the evaluation of long chains of pointers for traversal. Every evaluation of a pointer (a) induces a translation of the virtual into the physical memory address using the MMU, possibly producing a TLB miss, and (b) loads data from a random memory location, possessing high risk for CPU cache misses. The search algorithms of CSSL do not need to chase pointers, but can utilize the linearized fast lane array. Only one pointer evaluation is needed, when accessing the data list via the proxy lane, and up to $1/p$ pointers need to be followed when traversing the data list.
- **Improved Space Efficiency:** CSSL require less space than regular skip lists. Let k be the size of a key, r be the size of a pointer and m be the number of fast lanes. As a fat node of a regular skip list reserves space for $m + 1$ pointers (m fast lanes and the data list) in addition to the key, it requires $m * r + r + k$ bytes of memory. Overall, a regular skip list has a space complexity of $n * (m * r + r + k)$. In contrast, CSSL require $\sum_{i=1}^m p^i * n * k$ bytes of memory for the linearized fast lane array. The proxy lane and the data list need $n * (r + k)$ bytes each. In total, that boils down to a space complexity of $2 * n * (r + k) + \sum_{i=1}^m p^i * n * k$.
For instance, when storing one Million four-byte keys with ten fast lanes on a 64-bit architecture (eight-byte pointers), regular skip lists using fat nodes require 87.74MB of memory, whereas CSSL applying linearization need only 26.70MB.
- **SIMD Instructions:** As described in Section 2.3.2, SIMD instructions require data to be stored in consecutive memory locations. CSSL's memory layout fully satisfies this requirement and enables a vectorization of the search algorithm. Given that s is the width of a SIMD register and k is the size of a key, s/k fast lane elements can be compared in

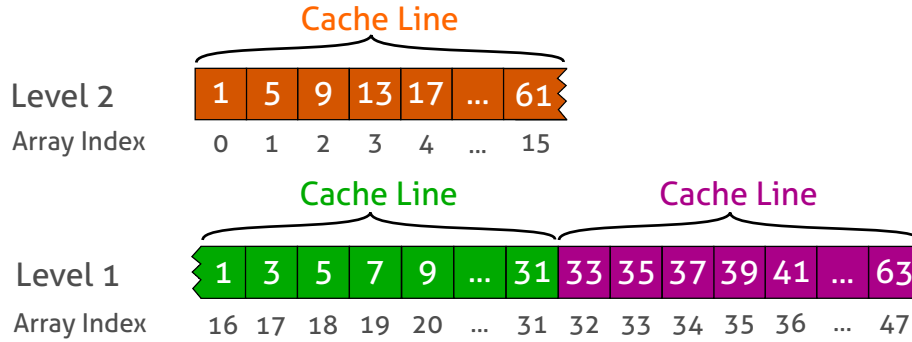


Figure 3.3: The linearized fast lane array of a CSSL indexing all four-byte integers in $\{1, \dots, 64\}$ with two levels ($p = 1/2$). The fast lane array is aligned to 64-byte cache lines.

parallel. Modern CPUs usually feature SIMD registers having a size of 256 bits⁵, thus eight four-byte integers can be processed with one instruction.

However, using SIMD instructions for the navigation of fast lanes would only be useful for very sparse fast lanes ($1/p > s/k$), where many elements have to be compared at each fast lane. As such a setting is rarely used in practice, we refrain from a vectorization of this part of the search algorithm. Instead, as shown in Section 3.3.2, we apply SIMD instructions to the range query operator of CSSL, which exploits the lowest fast lane to iterate over elements matching the search object.

Besides these main concepts, we apply a number of further optimizations to fully exploit modern CPUs:

- As illustrated in Figure 3.3, we always tailor the size of the fast lanes as multiples of the CPU cache line size. This primarily affects the highest fast lane level and further increases the cache line utilization of CSSL.
- In practice we observed that searching the highest fast lane is very expensive in terms of CPU cycles if the number of fast lanes is restricted and the highest fast lane contains a lot more than $1/p$ elements. In the worst case, we have to scan the entire highest fast lane, while searching the remaining fast lanes never requires more than $1/p$ comparisons per lane. Although caching helps, as the highest fast lane is frequently accessed, it still remains challenging. Therefore, we accelerate searching the highest fast lane by using a binary search instead of sticking to a sequential scan. The remaining fast lanes are still scanned.

3.3.2 Search Algorithms and Updates

This section presents algorithms to execute lookups, range queries and updates in CSSL. The shown algorithms work on four-byte integer keys. We start with describing lookups and sub-

⁵Introduction to Intel®Advanced Vector Extensions, <https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions>, Last access: August 29, 2018.

Algorithm 2 Lookup operator of CSSL.

key: The search key.

```

1: function LOOKUP(key)
2:  pos  $\leftarrow$  BinarySearchTopFastLane(fast_lanes, key)
3:  for level  $\leftarrow$  MAX_LEVEL-1 to 1 do
4:    rel_pos  $\leftarrow$  pos - start_pos[level]
5:    while key  $\geq$  fast_lanes[pos + 1] do
6:      pos  $\leftarrow$  pos + 1
7:      rel_pos  $\leftarrow$  rel_pos + 1
8:    end while
9:    if level = 1 then
10:     break
11:    end if
12:    pos  $\leftarrow$  start_pos[level-1] + 1/p * rel_pos
13:  end for
14:  if key = fast_lanes[pos] or SearchViaProxy(pos - start_pos[1], key) then
15:    return key
16:  end if
17:  return INT_MAX
18: end function

```

sequently delve into range queries. Finally, we also show how to process inserts and deletes in CSSL.

Lookups

In database systems, lookups are typically used to verify that a certain key exists. Algorithm 2 shows the pseudocode of our lookup implementation. The linearized fast lane array is stored in the variable *fast_lanes*. For each fast lane level, the array *start_pos* holds the first occupied index within *fast_lanes*; the highest fast lane obviously starts at index 0. When navigating the fast lanes, we use the helper variable *pos* to track the absolute position within the array, and the helper variable *rel_pos* to track the relative position within the current fast lane level.

In particular, CSSL execute lookups as follows. First, the highest fast lane is processed with a binary search (see Line 2). Second, the remaining fast lanes are hierarchically searched to narrow down the segment of the data list that can hold the search key (see Lines 3-13). Here, we use a sequential scan instead of a binary search, because we need to compare only $1/p$ elements (and p is typically between $1/2$ and $1/5$). Third, if the lowest fast lane contains the searched element, which happens on average at every $1/p$ -th search, the search key is immediately returned. Otherwise the associated proxy node is loaded to access the corresponding elements of the data list (see Lines 14-16). If the search key is neither part of the lowest fast lane nor found in the data list, we return *INT_MAX* to indicate that the search key does not exist (see Line 17).

Listing 3.2: Return value of the range query operator.

```

1 typedef struct {
2     _CSSL_Node* first_match, last_match;
3     size_t count;
4 } _CSSL_RangeSearchResult;

```

Range Queries

Range queries retrieve all keys satisfying specified lower and upper range boundaries. In database systems, the results of range queries are often summarized by aggregate functions, e.g., *COUNT*, *SUM*, *MIN*, or *MAX*. For instance, a medical researcher may be interested in the number of genomic variants that have been found in a certain region of the genome, which boils down to counting the results of a range query.

The range query operator of CSSL accepts range boundaries as input parameters and returns pointers to the smallest and largest elements of the data list matching the query, which equals a linked list that can be conveniently processed in downstream analysis. CSSL also return the size of the result set, which resembles the *COUNT* aggregation. The range query operator could be easily extended such that further pre-computed values of other aggregate functions are returned. Listing 3.2 shows the return value.

Since CSSL store data in a sorted order, they implement range queries by looking up the smallest matching key and iterating over all subsequent keys until a key larger than the upper range boundary is found. In particular, the processing of range queries can be broken down into two parts. First, we navigate the linearized fast lanes to retrieve the segment of the data list that holds the smallest key within the query range. The implementation is very similar to a lookup for the lower boundary, except that we are not restricted to exact matches but use a *greater or equal* comparison. Second, once the smallest element has been found, the range query operator jumps back to the lowest fast lane and, starting at the position where the lookup in the previous step stopped, scans all elements using SIMD instructions to find the largest element that satisfies the queried range. CSSL can process eight four-byte keys, e.g., integers, or floating-point values, in parallel using AVX intrinsics.

Listing 3.3 shows how CSSL collect all keys satisfying a range query; we omit the lookup for the first matching key and the computation of the result set size. SIMD intrinsics are highlighted in bold.

We first propagate the *upper_boundary* to all eight lanes of a 256-bit SIMD register (see Line 4). Then, we traverse over the lowest fast lane using SIMD instructions until we come across a non-matching key (see Lines 7-20): We load eight elements, starting at index *rel_pos* of the lowest fast lane, into a SIMD register (see Lines 10-11) and compare them with the search key using *greater than* (see Line 13). If the comparison fails we can immediately abort and prune further comparisons (see Lines 15-17). Once the segment of the data list featuring the largest matching key has been found, we load the according proxy node (see Line 23) and determine the exact results (see Lines 24-30), which are finally returned (see Line 31).

Listing 3.3: Collecting keys matching a range query.

```

1  _CSSL_RangeSearchResult result;
2  result.start = lookup(lower_boundary);
3  // store upper boundary in all lanes of a SIMD register
4  __m256 search = _mm256_castsi256_ps(_mm256_set1_epi32(upper_boundary));
5  // process the lowest fast lane to find the segment containing
6  // the largest key matching the range query object
7  while (rel_pos < level_items[1]) {
8      // load the current fast lane element and its successors into a
9      // SIMD register (rel_pos and pos are filled by the initial lookup)
10     __m256 compare = _mm256_castsi256_ps(_mm256_loadu_si256(
11         (__m256i const *) &fast_lanes[pos]));
12     // compare the current segment of the fast lane with the search key
13     __m256 res      = _mm256_cmp_ps(search, compare, _CMP_GT_OQ);
14     // abort once a non-matching key has been found
15     if (_mm256_movemask_ps(res) < 0xff) {
16         break;
17     }
18     pos += 8;
19     rel_pos += 8;
20 }
21 // load the associated proxy node and determine which key of the
22 // current segment is the largest key satisfying the range boundaries
23 proxy = proxy_nodes[rel_pos];
24 result.end = proxy->pointers[1/p - 1];
25 for (size_t i=1; i < 1/p; i++) {
26     if (upper_boundary < proxy->keys[i]) {
27         result.end = proxy->pointers[i - 1];
28         break;
29     }
30 }
31 return result;

```

Insertions and Deletions

Our implementation initializes a CSSL with a sorted set of keys, which resembles a bulk load. Although we linearize the fast lanes and store them in a static array, leading to less flexibility than a pointer-based implementation, CSSL are able to handle updates. In the following, we describe techniques for inserting new keys, updating existing keys, and deleting keys.

Inserting keys

Directly ingesting new keys into the linearized fast lane array would, especially in the case of large data sets, require lots of shift operations to preserve the sorted order. Therefore, insertions leave the fast lanes untouched but are only applied to the data list, which is implemented as a pointer-based linked list. We create a new node and add it at the appropriate position of the data list. Once a certain number of keys have been inserted, we rebuild the fast lane array. Hence, new keys are eventually reflected in the fast lane hierarchy. Although it may take a while until updates are pushed to the fast lanes, new keys are instantly available for search. As shown in Section 3.3.2, if a key has not been found in the lowest fast lane, the search algorithm moves down to the data list and scans it until the key is found or proven to be non-existing.

Rebuilding the linearized fast lane array consists of four steps: (1) We allocate a new array of size $\sum_{i=1}^m p^i * n$ that can hold m fast lanes. (2) We process the complete data list and add every $1/p$ -th key to the lowest fast lane. Concurrently, we build up the new proxy lane. (3) Recursively, we build the fast lane hierarchy from the *bottom up*, adding every $1/p$ -th element from the fast lane at level i to the fast lane at level $i + 1$. (4) We replace the previous fast lane array and proxy lane with the new ones. Once replaced, we delete the old entities.

Periodic rebuilds of the static fast lane array allow CSSL to adapt to changes in the indexed data. Rebuilds enable fast lanes to reflect the current data list and keep their pruning capabilities despite updates. If we would only insert into the data list but never update the fast lanes, search algorithms would have to do a lot of work at the data list. On the other hand, such rebuilds come at the cost that they are invoked by insert operations, which block until the rebuild is finished⁶. Therefore, the frequency of rebuilds has a strong impact on the insert performance of CSSL. We identify a trade-off between the performance of inserts, which prefer seldom rebuilds, and search queries, which favor frequent rebuilds.

Deleting keys

As opposed to insertions, we cannot delete keys from the data list but leave the fast lanes untouched, because this could lead to invalid search results. Hence, before deleting a key from the data list, we need to eliminate it from the fast lane array.

We cannot remove an element from a fast lane and shift all consecutive elements by one position in the left direction, because this would be very expensive, especially when removing from the beginning of a large fast lane. Instead, we replace the deleted fast lane element with a copy of its successor, which can be conducted with one instruction. Alternatively, we could also replace the removed entry with a special character indicating that the element has been

⁶In a multithreaded setting, we could handle rebuilds in the background with a worker thread.

deleted. However, that would introduce additional complexity to the search algorithm, while our approach does not require any changes. Duplicate fast lane entries will be eventually removed once the array gets rebuilt.

In the last step of a deletion, we remove the key from the data list by changing the *forward* pointer of the preceding node to point to the successor of the to-be-removed node. Finally, the node can be deleted.

Our approach to deleting elements from fast lanes may waste a lot of memory space, because duplicates are not removed until the next rebuild happens. In the meantime, the space is not used although it is allocated. In the worst case, when a CSSL gets deleted element by element, the linearized fast lane array remains unchanged in terms of size, although the data list is empty. To cope with such scenarios, we introduce an internal counter that tracks the number of insertions and deletions that have occurred since the last rebuild. Once the counter exceeds a certain threshold, a rebuild is triggered. The counter is reset after every rebuild.

Updating keys

Updates are implemented as an insertion followed by a deletion.

3.4 Evaluation

We compare CSSL with ART, CSB⁺-trees, and a binary search (BS) on a static array. The binary search is the only competitor that is read only by design. We also include an open-source, main-memory adapted variant of the B⁺-tree [Comer 1979] as baseline approach, though we note that B⁺-trees were originally designed to be stored on disk.

We evaluate CSSL with two configurations to study the effects of dense and sparse fast lanes: CSSL₂ with $p = 1/2$ and CSSL₅ with $p = 1/5$. In our implementation, we expose the number of desired fast lanes as input parameter, which we set to a value close to the optimum that we empirically determine. In the experiments, it is set to nine.

The next section describes the experimental setup. Subsequently, we investigate the performance of range queries (see Section 3.4.3), the performance of lookup operations (see Section 3.4.4), the performance of mixed range/lookup workloads (see Section 3.4.5), and the space consumption (see Section 3.4.6) of the competitors.

3.4.1 Experimental Setup

Methodology

All experiments are single-threaded and measure the search throughput of the competitors, i.e., how many queries can be processed per second. Experiments are run three times; we present the arithmetic mean.

Software

All competitors were implemented in C/C++ and compiled with GCC 4.8.4 using optimization `-O3`. For ART and the CSB⁺-tree, we used implementations provided by the authors. For the

B⁺-tree, we used a popular in-memory implementation⁷. For BS, we use our own implementation that stores all data in a static array in a sorted order. Here, we implement range queries by first looking up the smallest matching key with a binary search and then scanning over all subsequent keys until a key larger than the upper boundary of the range query has been found. We use PAPI⁸ to collect hardware performance counters and valgrind⁹ to measure space consumption.

Hardware

All experiments were run on a server machine equipped with a Intel Xeon E5-2620 CPU (2 GHz clock rate, 15MB L3 cache, 256-bit SIMD registers). The system features 32 GB of main memory and uses Linux as operating system.

3.4.2 Experimental Data and Workloads

Synthetic Data

For synthetic data, we generate n four-byte integer keys with a dense and a sparse distribution. For the dense distribution, every key in $[1, n]$ is indexed; for the sparse distribution, n random keys from $[1, 2^{31})$ are indexed. We generate range queries by selecting a random key from the set of indexed elements as lower boundary and add the range size, e.g., 0.1% of n , 1% of n , or 10% of n , to obtain the upper boundary. For a dense distribution, this results in ranges covering $|upper_boundary - lower_boundary|$ elements. For a sparse distribution, the queries are created using the same technique, yet return less elements, which usually leads to higher search throughput.

Real-World Data

We also obtained real-world data from the bioinformatics domain to study the performance of the competitors when being confronted with a non-synthetic key distribution. Like the Genomic Multidimensional Range Query Benchmark (see Section 2.4), we used genomic variant data from the 1000 Genomes Project [The 1000 Genomes Project Consortium 2015] as data source. As opposed to GMRQB, which targets multidimensional data and workloads, we here indexed only one attribute, the genomic position of the variants. As some competitors show a poor space efficiency quickly exceeding the available main memory, we do not index the complete genome but only consider the first two chromosomes, which sums up to 13,571,394 genomic positions (keys) in total.

For the experiments on real-world data, we generate ranges using the same technique as for synthetic data.

⁷bpt: B⁺ Tree Implementation, <http://www.amittai.com/prose/bpt.c>, Last access: August 29, 2018.

⁸PAPI, <http://icl.cs.utk.edu/papi>, Last access: August 29, 2018.

⁹Valgrind Home, <http://valgrind.org>, Last access: August 29, 2018.

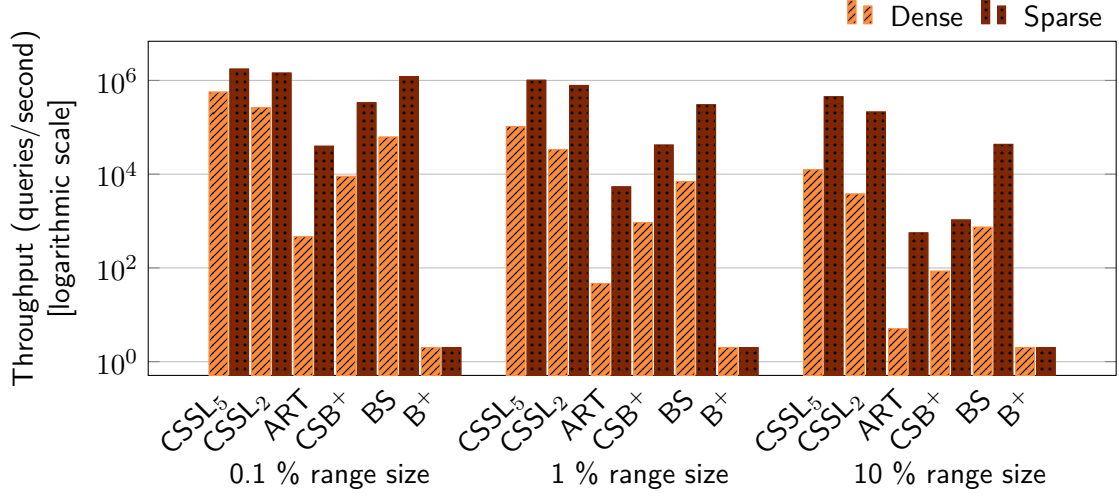


Figure 3.4: Range query throughput of the competitors on 16 Million four-byte synthetic integer keys depending on the range size.

3.4.3 Range Queries

We investigate the performance of the competitors when processing range queries with varying range sizes on synthetic and real-world data.

16 Million Synthetic Keys

Figure 3.4 shows the throughput of the competitors when executing range queries of varying sizes on 16 Million synthetic keys, which equals 61.04MB of raw data. Both CSSL configurations outperform all other contestants for both key distributions and all evaluated range sizes.

In contrast to ART, CSB⁺, and B⁺, CSSL do not need to follow pointers when iterating over keys matching a range query, but sequentially process data held in an array structure. Although BS can make use of a sequential access pattern to evaluate range queries, like CSSL, BS has to scan all keys within the query range, while CSSL exploit the lowest fast lane to reduce the number of keys to compare. Furthermore, CSSL is the only competitor that applies SIMD instructions to the range query operator, which provides an additional speed-up of between two to three times. On average, CSSL₅ is up to 16.8X faster (10.4X for sparse data) than the second best competitor, BS, and outperforms the remaining approaches by up to four orders of magnitude.

CSSL₅ is faster than CSSL₂, because fast lanes skip over five instead of two elements, reducing the number of keys that need to be compared when searching for the range end (see Listing 3.3).

We also investigated various cache-related hardware performance counters (see Table 3.1), which explain the superior range query performance of CSSL: (1) The range query operator of CSSL exploits most prefetched cache lines, which leads to only few cache misses. (2) CSSL achieve high cache line utilization requiring only few cache accesses (accesses equal the sum of hits and misses). (3) Furthermore, CSSL rarely generate branch mispredictions, because they

3 CSSL: Processing One-Dimensional Range Queries in Main Memory

Performance Counter	CSSL ₅	CSSL ₂	ART	CSB ⁺	BS	B ⁺
Dense						
CPU Cycles	202k	661k	501M	27M	3.4M	1,070M
Branch Mispredictions	12	15	813k	46	13	1.4k
LLC Hits	8k	24k	1.3M	49k	21k	1.6k
LLC Misses	21	7.3k	2.7M	243k	7.4k	7.8M
TLB Misses	5	13	1.6M	99	24	381k
Sparse						
CPU Cycles	5k	13k	4.5M	620k	59k	1,095M
Branch Mispredictions	13	16	16k	4.6k	13	832
LLC Hits	139	373	14k	364	325	1.8k
LLC Misses	23	165	28k	5.7k	278	7.4M
TLB Misses	3	5	19k	958	10	369k

Table 3.1: Hardware performance counters per range query (10 % range size) on 16 Million four-byte integer keys.

process mainly consecutive positions of the fast lane array. This benefits the number of CPU cycles needed to execute a range query.

In contrast, the tree-based approaches ART, CSB⁺ and B⁺ produce many LLC misses due to random data accesses necessary for navigating their search trees. As expected, by eliminating pointer accesses and taking the sizes of cache lines into account, the CSB⁺-tree achieves higher cache efficiency than the B⁺-tree.

256 Million Synthetic Keys

In this experiment, we study the competitors when confronted with a larger number of keys. We execute range queries of varying selectivity on 256 Million four-byte synthetic integer keys, which equals a raw data set size of 976.56MB. We only show the performance of CSSL, ART, and BS, because the space needs of both the CSB⁺-tree and the B⁺-tree exceeded the available 32GB of main memory.

As shown in Figure 3.5, also for large data set sizes, both variants of CSSL outperform all remaining competitors. For both key distributions, CSSL are up to 11.5X faster than the second best competitor BS.

Real-World Data

To increase the practical relevance of our experimental results, we also investigate the range query performance of the competitors when being confronted with a non-synthetic key distribution. We use variant data from the 1000 Genomes Project (see Section 3.4.1) and index the genomic positions of all variants found on the first two chromosomes. This results in $n = 13,571,394$ four-byte integer keys. As in the previous experiments, we evaluate range queries of different selectivities (0.1%, 1%, and 10% of n). Figure 3.6 shows the results.

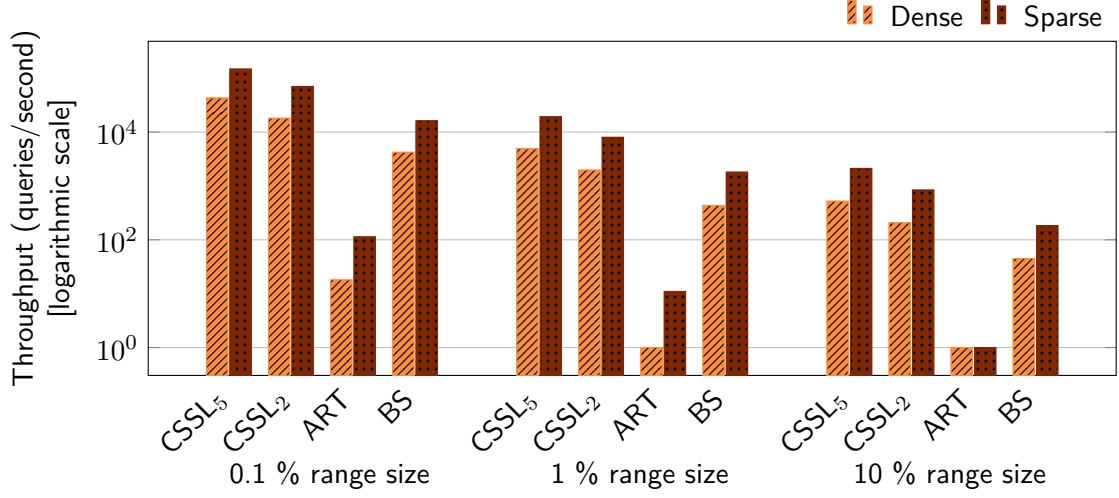


Figure 3.5: Range query throughput of the competitors on 256 Million four-byte synthetic integer keys depending on the range size.

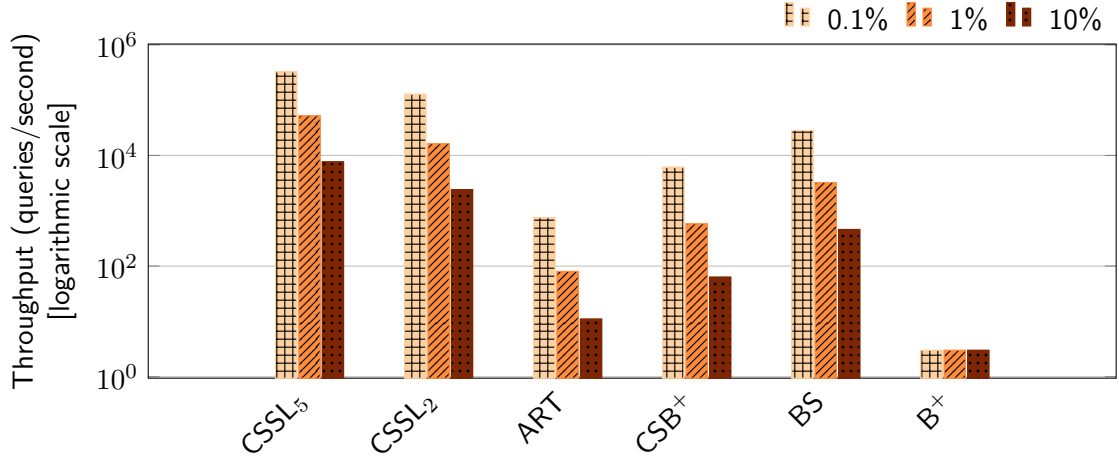


Figure 3.6: Range query throughput on 13,571,394 four-byte integer keys obtained from real-world genomic variant data depending on the range size.

Also for non-synthetic data, CSSL outperform all other competitors when executing range queries. CSSL₅ achieves the highest throughput and is followed by CSSL₂, BS, CSB⁺, ART and B⁺. While CSSL₅ is up to 16.69X faster than BS for low selectivities (10% range size), it is up to 11.52X faster than BS for high selectivities (0.1% range size). The performance gains of CSSL over the other competitors increase with the number of matching keys.

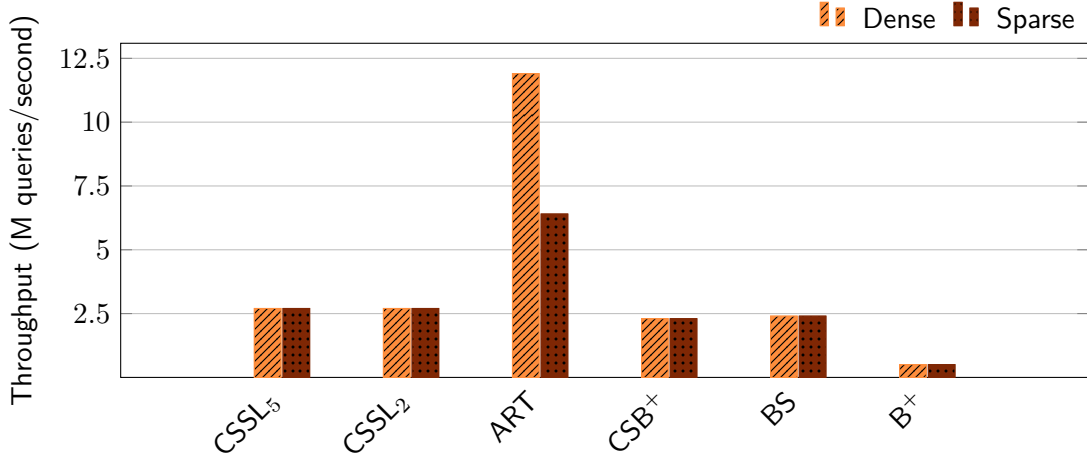


Figure 3.7: Lookup throughput on 16 Million four-byte synthetic integer keys.

3.4.4 Lookups

Lookups are a common operation in database management systems and mainly needed to verify the existence of a certain key. Although CSSL target the efficient execution of range queries, they also provide an implementation of single-key lookups. Figure 3.7 shows the throughput of the competitors when processing lookups on 16 Million four-byte synthetic integer keys. ART achieves the best performance for both key distributions. Additionally, it is the only competitor that can boost its search performance for dense keys, for instance by using lazy expansion [Leis et al. 2013]; the remaining competitors show identical results for both distributions. CSSL achieve the second best throughput and are closely followed by BS and CSB⁺. The B⁺-tree shows the worst performance. In this experiment, ART is 4.4X faster than CSSL for dense keys, and 2.4X faster than CSSL for sparse keys. When evaluating lookups, CSSL₅ requires up to five comparisons per fast lane, whereas CSSL₂ needs only up to two comparisons. On the other hand, as both configurations employ the same number of fast lanes, CSSL₂ requires more work at the highest fast lane level, which holds more entries than that of CSSL₅. CSSL₂ and CSSL₅ show very similar results, because the differences in the fast lane granularity have only negligible impact on the performance of lookups.

Table 3.2 presents cache-related hardware performance counters per lookup operation on 16 Million four-byte integer keys. When executing lookups, ART achieves high cache efficiency: It follows most predicted branches and produces only few LLC misses. ART can evaluate lookups with only few instructions, especially when dealing with dense keys, while the other competitors need more computations resulting in a larger number of CPU cycles spent per lookup. CSSL produce only few LLC and TLB misses, but require more cache accesses than ART. In summary, though being optimized for range queries, CSSL are able to achieve a lookup throughput that outperforms BS, CSB⁺ and B⁺ and is almost as fast as ART, especially in the case of sparse keys.

Performance Counter	CSSL ₅	CSSL ₂	ART	CSB ⁺	BS	B ⁺
Dense						
CPU Cycles	927	956	209	1,068	1,036	5,889
Branch Mispredictions	9	13	0	1	12	12
LLC Hits	11	8	2	3	21	28
LLC Misses	5	8	2	5	9	39
TLB Misses	1	3	2	3	4	20
Sparse						
CPU Cycles	926	951	383	1,054	1,029	5,789
Branch Mispredictions	9	13	0	3	12	12
LLC Hits	11	8	5	3	20	29
LLC Misses	5	8	3	4	10	38
TLB Misses	1	3	4	5	4	20

Table 3.2: Hardware performance counters per lookup on 16 Million four-byte integer keys.

3.4.5 Mixed Workloads

Many real-world applications execute neither only lookups nor only range queries, but apply a mix of both. For instance, a medical researcher may be interested in a certain variant found in a certain patient, which would require a lookup, as well as all variants of complete disease cohorts, which would boil down to a range query.

We investigate the competitors when executing a mixed workload consisting of an equal number of lookups and range queries. In particular, we generate a workload of one Million queries, i.e., 500,000 lookups and 500,000 range queries, and execute it on 16 Million four-byte integer keys. For range queries, we always use a range size of 500,000.

As shown in Figure 3.8, CSSL achieve the best performance for mixed lookup/range query workloads and are followed by BS, CSB⁺, ART and B⁺. Although ART offers the best lookup performance, CSSL are up to two orders of magnitude faster when running a workload that also includes range queries.

Range queries involve tremendously more keys than single-key lookups, take more time to execute and therefore have higher impact on the overall performance of mixed workloads with equal numbers of range queries and lookups. The results of this experiment emphasize the need for a fast range query implementation, even for applications that do not exclusively run range queries.

3.4.6 Space Consumption

Figure 3.9 visualizes the space requirements of the competitors when storing 16 Million four-byte integer keys. As observed in Section 3.4.4, ART is better suited for managing dense than sparse data. For dense key distributions, ART requires the least amount of space and is followed by BS and CSSL. The tree-based approaches B⁺ and CSB⁺ show the worst memory consumption. For sparse key distributions, BS achieves the highest space efficiency and is followed by CSSL₅

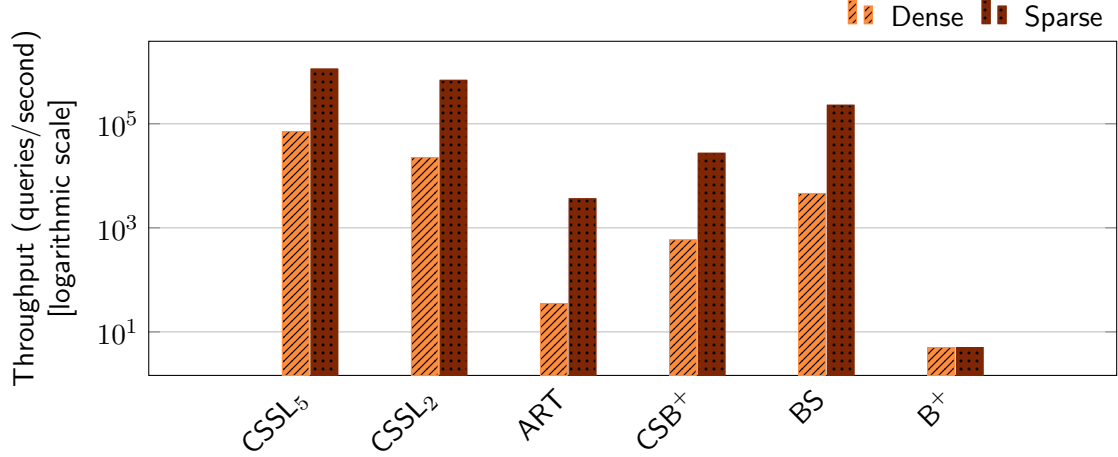


Figure 3.8: Throughput of a mixed workload consisting of 500,000 lookups and 500,000 range queries when applied to 16 Million four-byte synthetic integer keys.

and ART. Again, B⁺ and CSB⁺ show the worst results. For 16 Million keys, CSSL₂ requires approximately 1.8 times more memory than CSSL₅, because the fast lanes hold more entries.

Note that ART’s space efficiency would probably grow for keys larger than four bytes. Then, ART is able to employ further optimization techniques, e.g., path compression, which are not beneficial for *small* keys [Leis et al. 2013].

3.5 Discussion

The main idea of CSSL is the linearization of the fast lanes, which allows to pack all fast lane entities into a dense array. The presented memory layout enables the search algorithms of skip lists to use a sequential access pattern and allows the range query operator to exploit SIMD instructions. However, the improved search performance comes at the cost of more complicated updates. While inserting into a regular, pointer-based skip list with m fast lanes only requires changing up to m pointers, CSSL need to periodically rebuild the complete fast lane hierarchy, which blocks insert operations.

In this chapter, we only discussed the single-threaded case. However, driven by the rise of multi-core CPUs (see Section 2.3.3), modern database systems often execute multiple queries concurrently and therefore also need to access index structures simultaneously with multiple threads. The current implementation of CSSL allows concurrent read access without requiring any locks. Also, insert operations could be implemented lock-free¹⁰, as demonstrated by Fomitchev and Ruppert [Fomitchev et al. 2004] for regular skip lists. However, the rebuild operation would need to lock the fast lane array for a short amount of time when replacing it with the new one. Also, the delete operation requires a write lock on the linearized fast lane

¹⁰Here, we refer to locks as used in parallel programming. Database people sometimes use the term *latch* to refer to locks used in programming.

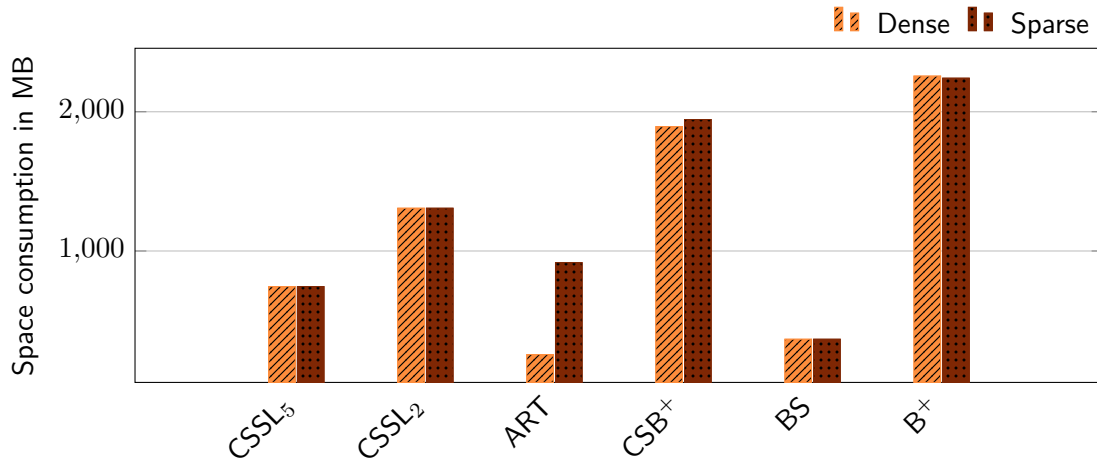


Figure 3.9: Space consumption for 16 Million four-byte integer keys.

array when updating the fast lane entries. In summary, we can adapt CSSL to a multithreaded setting. However, the rebuild and delete operations would have to lock the complete fast lane array, which will hurt their performance.

In our experiments, we only considered integers as data type for keys. Nonetheless, we could easily adapt the implementation of CSSL to other key types that provide a lexicographical order, e.g., long values, floating-point values, or even strings. Though, note that we use AVX intrinsics, which support only integers and floating-point values, to vectorize the range query operator. Therefore, for instance, when using CSSL to index sets of strings, we could not leverage SIMD parallelism. Also consider the space efficiency of CSSL when handling more complex data types. For instance, when indexing a set of strings with strongly varying sizes, CSSL determines the size of the longest string and allocates that amount of space for each key wasting a lot of space. For such data, other index structures, like main-memory optimized tries [Leis et al. 2013], may be a better choice.

3.6 Summary

In this chapter, we presented the cache-sensitive skip lists (CSSL) as a cache-optimized approach to processing one-dimensional range queries in main memory. We studied the foundational concepts behind CSSL, discussed why we chose conventional skip lists as basis data structure, and showed which steps are necessary to adapt their memory layout to modern CPUs. Moreover, we introduced algorithms for executing search queries and described how to dynamically update CSSL.

In a comprehensive evaluation, we compared CSSL with state-of-the-art main-memory index structures. We showed that, regardless of the distribution or the cardinality of the data set, CSSL clearly outperform all other competitors for range queries, sometimes by up to four orders of magnitude. CSSL also provide fast lookups, where they are only outperformed by the recent

3 CSSL: Processing One-Dimensional Range Queries in Main Memory

ART index structure. Using a mixed lookup and range query workload, we emphasized the importance of efficient range query implementations. For such workloads, CSSL also achieve the best performance.

In the following chapters, we continue with studying the processing of range queries in main memory, but delve into the more complex multidimensional domain.

4 An Analysis of Multidimensional Range Queries on Modern Hardware

Multidimensional range queries (MDRQ) are selection queries that specify query intervals for some, many, or all dimensions of a multidimensional data space. They are a common part of many database workloads, especially in the scientific community, and are used in various applications, such as genomic analysis [Thorvaldsson et al. 2013], online analytical processing [Ho et al. 1997; Liang et al. 2000], internet of things [X. Li et al. 2003], etc (see Introduction). MDRQ can be answered either by sequential scans over the complete data or by dedicated multidimensional index structures (MDIS).

In the context of traditional disk-based server machines, Weber et al. [Weber et al. 1998] showed that a scan-based query evaluation outperforms MDIS when roughly 20% or more of all data have to be visited, assuming that accesses of consecutive blocks (as in a scan) are at least five times faster than random accesses (as necessary for most MDIS). Since then, this threshold has been used as the basic rule of thumb for choosing access paths in MDRQ, while completely ignoring the advancements in server hardware that occurred over the last years, like large main-memory capacities or multi-core CPUs (see Section 2.3).

Thus, it is time to re-evaluate the performance of MDRQ on modern hardware architectures to see whether the traditional selectivity-based rule of thumb for access path selection still holds or should be updated. Our work is motivated by recent analyses of one-dimensional selection queries, which showed that main-memory database systems strongly favor scanning [Das et al. 2015].

In this chapter, we conservatively adapt three popular MDIS to a parallel and in-memory setting, namely (1) the R*-tree [Beckmann et al. 1990], an R-tree [Guttman 1984] variant minimizing overlap between different index regions, (2) the kd-tree [Bentley 1975], an index structure already originally designed for in-memory computations, and (3) the VA-file [Weber et al. 1998], which can be considered as a mixture between an MDIS and a sequential scan. Section 2.2 provides detailed descriptions of the approaches.

Our adaptation is conservative in the sense that we withstood the temptation to design new, highly-tuned, parallel MDIS, because we want to study classical index structures, which are employed by many mature database systems, e. g., SQLite uses R*-trees¹, PostgreSQL uses kd-trees², etc. Our main goal is to evaluate techniques, which can be applied to existing MDIS with as few adaptations as possible, thus raising the practical relevance of our experimental results.

We compare the adapted variants of the MDIS to the performance of a parallel, in-memory, scan-based implementation of MDRQ using real-world and synthetic analytical workloads over

¹The SQLite R*Tree Module, <https://sqlite.org/rtree.html>, Last access: August 29, 2018.

²PostgreSQL: Documentation: 10: 11.2. Index Types, <https://www.postgresql.org/docs/current/static/indexes-types.html>, Last access: August 29, 2018.

real-world and synthetic data sets of varying size, dimensionality, and skew. We employ only basic parallelization schemes and refrain from using highly-tuned scan implementations, like the recent BitWeaving [Y. Li et al. 2013] or the ELF approach [Broneske et al. 2017a].

This chapter makes four main contributions:

- We present two approaches to dividing a multidimensional data set into disjoint partitions enabling a parallel evaluation of range queries.
- We describe a novel SIMD-based algorithm to compare tuples with MDRQ search objects, which is a frequently-needed task in query evaluation.
- We conservatively adapt three popular MDIS and two scan flavors to main-memory storage, SIMD instructions, and multi-core CPUs.
- We conduct a comprehensive evaluation to investigate whether the traditional selectivity threshold [Weber et al. 1998] still holds on modern hardware architectures or should be updated.

The remainder of this chapter is organized as follows. In the next section, we present two techniques for partitioning multidimensional data sets and explain how to use them for the parallelization of MDRQ. Section 4.2 shows how to vectorize the comparison of a MDRQ search object, defined by a lower and an upper boundary, with a multidimensional tuple (or data object). Section 4.3 presents our adaptations of three MDIS and two scan approaches to modern hardware. In Section 4.4, we conduct a comprehensive evaluation of the performance of multidimensional range queries in an in-memory, parallel setting applying synthetic and real-world query workloads to synthetic and real-world data sets. Finally, Section 4.5 critically discusses this chapter and Section 4.6 provides a summary.

Parts of this chapter have been previously published in [Sprenger et al. 2018b]³.

4.1 Partitioning for Parallelization

The execution of individual search queries can be parallelized using two techniques. We may either (a) build one large index structure over the complete data set and parallelize the index traversal or (b) create multiple index structures over subsets of the data and query these in parallel.

Especially for hierarchical MDIS, like R*-trees or kd-trees, a parallelization of the index traversal would require solving a complex load balancing problem. The number of tree branches would grow exponentially with the depth of the tree, creating over-provisioning in the upper levels and under-provisioning in the lower levels. As a consequence, some threads might finish earlier than others and would need to be reassigned to different tasks, requiring a barrier synchronization after each tree level [Buluç et al. 2011; Yoo et al. 2005]. Furthermore, this strategy would have to be tailored to each individual indexing approach individually, preventing an universal parallelization technique that can be applied to MDIS not considered in this thesis. For these

³Sections 4.4.8 and 4.4.9 provide unpublished results.

reasons, such schemes are beyond the scope of our work, but have been explored, for instance, in [Koudas et al. 1996] or [Schnitzer et al. 1999].

We parallelize the execution of MDRQ following the second scheme: We partition the data and manage each partition in an individual instance of an index (or array in the case of a scan). MDRQ are concurrently applied to each partition and results are concatenated to obtain the final result set. This strategy has the advantage that it can be applied to any MDIS, not only the ones considered here, because we essentially build a conventional MDIS for every partition. Only the orchestration of the different MDIS, which does not require any synchronization, and the result concatenation for obtaining the final result set have to be added. Furthermore, load balancing is quite simple, because it only requires adapting the granularity of partitioning, which may also suffice to adapt to the number of hardware threads available on a machine.

On the downside, this scheme performs many comparisons redundantly, because every partition-wise MDIS has to cover the entire space. This prohibits the pruning of entire partitions (or MDIS instances), which is a burden in the case of very high selectivities. In R*-trees, every search initially traverses similar layers, while in VA-files, multiple buckets with the same approximation have to be scanned. Note, however, that this redundant work is performed in parallel.

In this section, we describe two different partitioning schemes, namely horizontal partitioning and vertical partitioning, which are quite similar to row-based [Lemahieu et al. 2018] and column-based [Stonebraker et al. 2005] data storage formats. Figure 4.1 illustrates the techniques when dividing twenty five-dimensional tuples into five partitions.

Horizontal Partitioning

Horizontal partitioning divides a data set of n objects into p partitions. Each partition holds a near-identical number of tuples, i. e., partition size $\approx n/p$. To obtain a robust load balancing, we assign tuples at random to partitions and set $p = t$ given that t is the number of available hardware threads (or virtual cores). Such a scheme is simple to implement and maintain, also in the presence of inserts or updates⁴. The different partitions are treated as independent. For MDIS, this means that actually p indexes are built, each covering one partition, which allows using existing implementations without adaptation. For scans, this boils down to creating p arrays, each holding the data of a particular partition.

At search time, each partition is processed by a distinct CPU thread that produces a partial result containing all objects of the partition matching the search query. Once all threads have finished, their partial results are concatenated by a *main thread*, responsible for orchestration, to produce the overall result set. Horizontal partitioning is similar to Parallel VA-files [Weber et al. 2000], except that here we target multithreading provided by modern CPUs instead of distributing queries among different physical machines. Also, many parallel database management systems use similar schemes, like *intra-operation parallelism* or *partitioned parallelism* [Taniar et al. 2008], to concurrently apply multiple instances of the same query to different subsets of the data.

⁴Assuming that workloads follow certain known distributions, one could also consider assigning tuples to partitions such that an optimal load balancing is achieved, as, for instance, described in [Berchtold et al. 1997]. Such a scheme would have to be implemented individually for each approach considered. Again, we refrain from applying such optimizations to keep our comparisons fair.

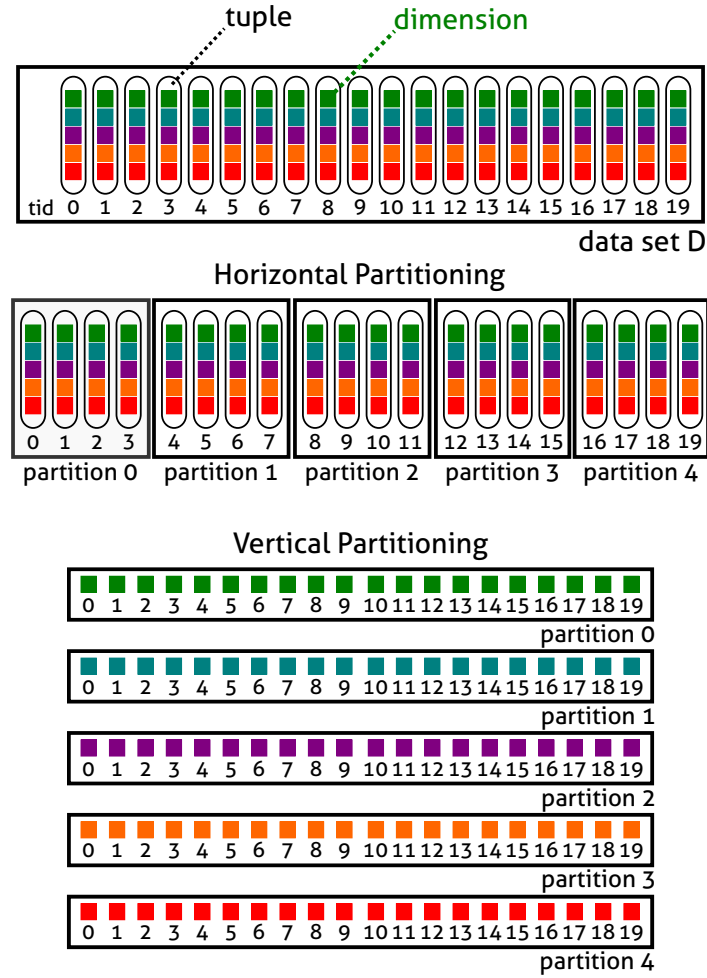


Figure 4.1: Horizontal and vertical partitioning used to divide twenty five-dimensional tuples into five partitions.

A major advantage of horizontal partitioning is its simplicity. If partitions are chosen at random, no further data structures are necessary for managing the mappings between data objects and partitions. The creation of the overall result set requires only concatenating the partial results, incurring only minimal synchronization effort. At initialization time, the number of partitions can be chosen freely to diminish the effect of stragglers, if suspected. At the downside, horizontal partitioning uses entire tuples as basic unit of access: In all cases, whole tuples have to be considered and must always be read completely into the CPU caches, even if only a subset of all dimensions are queried, as in the case of partial-match MDRQ.

Vertical Partitioning

Vertical partitioning slices tuples along their dimensions, creating one partition for each dimension. Hence, the number of partitions p is fixed and depends on the dimensionality of the data space, m . For query execution, an m -dimensional range query is split into m one-dimensional range queries, which are executed in parallel by distinct threads. Accordingly, also the degree of parallelism in this scheme is fixed at m .

Eventually, the one-dimensional results must be intersected to compute the final m -dimensional result set, which is more complicated than concatenation as used in horizontal partitioning. In our implementation, the processing of each partition creates a bitmask of size n that notes for every tuple whether it matches the search object in this dimension or not. Once all m bitmasks have been computed, we intersect them using a bitwise AND operation to determine the overall bitmask, which reflects all dimensions. The intersection can be performed in parallel on different chunks of the bitmasks. For modern hardware, different approaches to the intersection of sets have been presented. Schlegel et al. [Schlegel et al. 2011] showed how to intersect sets with SIMD instructions; however, their technique requires sets to be sorted. Tsirogiannis et al. [Tsirogiannis et al. 2009] presented multithreaded algorithms for intersecting sorted and unsorted sets. We also use a multithreaded approach, but can make use of efficient bit operators.

Clearly, vertical partitioning is only meaningful for scan-based MDRQ implementations, because MDIS would degenerate to one-dimensional indexes. The main advantage of vertical partitioning is its built-in support for partial-match queries, because it accesses only those dimensions (or partitions) that are referred to in the query. For complete-match queries, it offers no advantage over horizontal partitioning with regards to the amount of accessed data, because all dimensions have to be accessed in both cases. It has the general disadvantage that it requires large intermediate data structures, the bitmasks, and a complex technique to produce the final result set, the intersection. It is also rather complicated to achieve a good load balancing when $m < t$, since some threads remain idle, or $m > t$, because then not all partitions can be processed concurrently, scanning some partitions after others. In both cases, one should start to chop the partitions into subsets and parallelize the scan on this finer level of granularity. We do not tune our implementation in such manners but stick to the simple scheme of assigning one thread to each partition.

4.2 Vectorizing Range Queries

When evaluating range queries, both sequential scans and index structures need to compare search objects, defined by a lower and an upper boundary, to tuples to determine matches. As shown in Algorithm 3, this task is often implemented by iterating over all m dimensions of the data space with a *for* loop and comparing the MDRQ search object to the tuple, dimension by dimension. Clearly, this procedure can be sped up by aborting once a comparison fails. In this section, we vectorize this fundamental operation to compare multiple dimensions simultaneously. Obviously, the performance gains depend on the dimensionality of the data space: With a growing number of dimensions the benefits increase.

Listing 4.1 presents a SIMD-based implementation of Algorithm 3 that uses AVX intrinsics.

Algorithm 3 Scalar comparison of an m -dimensional MDRQ search object with an m -dimensional tuple.

lower_bound: The lower boundary of the MDRQ search object.
upper_bound: The upper boundary of the MDRQ search object.
tuple: The data object that is compared to the MDRQ search object.
m: The dimensionality of the data space.

```

1: function COMPAREMDRQWITHTUPLE(lower_bound, upper_bound, tuple, m)
2:   for dim  $\leftarrow$  0 to  $m - 1$  do
3:     if lower_bound[dim] > tuple[dim] OR upper_bound[dim] < tuple[dim] then
4:       return false
5:     end if
6:   end for
7:   return true
8: end function
  
```

Since AVX intrinsics work on 256-bit wide SIMD registers, the SIMD-based algorithm can process eight dimensions with one comparison, assuming that search queries and tuples are implemented as arrays of four-byte floating-point values.

In the beginning, the algorithm initializes SIMD registers and helper variables (see Lines 5-6). We determine the number of to-be-executed SIMD comparisons in Line 7 and store it in the variable *compares*. The dimensions are processed in chunks of eight (see Lines 8-20). If the dimensionality of the data space is not divisible by eight, we process some dimensions in a separate for loop using scalar instructions (see Lines 21-25).

The vectorized processing of dimension values works as follows: After loading chunks of eight dimensions of the tuple, the query's lower boundary and the query's upper boundary into SIMD registers, the algorithm compares the lower boundary (*lower_reg*) and the upper boundary (*upper_reg*) to the tuple (*data_reg*) using SIMD instructions. SIMD-based comparisons return bitmasks that indicate if a comparison, e.g., less or equal (*_CMP_LE_OQ*), was successful. The results of the comparisons are stored in the variables *mask_lower* and *mask_upper*. If all comparisons were successful, all bits are set to 1, which equals *0xFF* in our case (see Line 17). If a comparison failed, we can abort and prune further computations (see Line 18). Finally, if the tuple matches the given search query in all dimensions, we return *true* (see Line 26).

The SIMD-based comparison of search objects with data objects provides a theoretical speed-up of eight compared to the scalar variant. However, vectorization is only useful if the dimensionality of the feature space is greater than or equals eight. Otherwise it does not enter the SIMD-based for loop (see Lines 8-20), but automatically falls back to scalar instructions (see Lines 21-25). Also, since the presented algorithm processes eight dimensions at the same time, it is less flexible than the scalar variant with regards to early breaks⁵. Thus, the performance benefits decrease for selective queries.

⁵As soon as the first mismatch occurs, further comparisons are pruned resulting in an *early break*.

4.3 Conservative Adaptation of Multidimensional Index Structures

Listing 4.1: Vectorized comparison of an m-dimensional MDRQ search object with an m-dimensional tuple using AVX intrinsics.

```
1  bool CompareMDRQWithTuple(float[] lower_bound,
2                             float[] upper_bound,
3                             float[] tuple,
4                             int m) {
5      __m256 lower_reg, upper_reg, search_reg, lower_res, upper_res;
6      int i, mask_lower, mask_upper, mask;
7      const int compares = (m / 8) * 8;
8      for (i = 0; i < compares; i += 8) {
9          lower_reg = _mm256_loadu_ps(&lower_bound[i]);
10         upper_reg = _mm256_loadu_ps(&upper_bound[i]);
11         data_reg = _mm256_loadu_ps(&tuple[i]);
12         lower_res = _mm256_cmp_ps(lower_reg, data_reg, _CMP_LE_OQ);
13         upper_res = _mm256_cmp_ps(upper_reg, data_reg, _CMP_GE_OQ);
14         mask_lower = _mm256_movemask_ps(lower_res);
15         mask_upper = _mm256_movemask_ps(upper_res);
16         mask = mask_lower & mask_upper;
17         if (mask < 0xFF) {
18             return false;
19         }
20     }
21     for (; i < m; ++i) {
22         if (tuple[i] < lower_bound[i] || tuple[i] > upper_bound[i]) {
23             return false;
24         }
25     }
26     return true;
27 }
```

4.3 Conservative Adaptation of Multidimensional Index Structures

In this section, we describe our conservative adaptations of the R*-tree, the kd-tree, the VA-file, and two scan flavors. In particular, we describe our changes to the original data structures performed to adapt (1) to main-memory storage, (2) to the availability of multiple threads, and (3) to SIMD instructions. As our evaluation focuses on analytical workloads, we only discuss search algorithms. In all cases we tried to keep the original design and code untouched as much as possible to allow a re-usage of existing, proven implementations to the largest possible degree. Our overall strategy is to (a) keep the original storage layout but hold all blocks, e. g., inner and leaf nodes, in main memory, (b) partition the data horizontally at random into almost-equally sized chunks and build one MDIS per partition which work in parallel during the execution of MDRQ, and (c) use SIMD instructions only for the most time-consuming operation, i.e., the comparison of the tuples with the MDRQ search object.

R*-tree

We base our implementation of the R*-tree on the open-source library *libspatialindex*⁶ and perform the following adaptations:

- **Main Memory:** We keep all nodes of the tree in main memory. We do not adjust the node sizes to the sizes of disk blocks anymore, but choose node capacities such that inner and leaf nodes are aligned with cache lines, which increases the cache line utilization of search algorithms. To this end, we slightly change the default configuration of *libspatialindex* from a capacity of 100 to 96 tuples (MBR) for leaf nodes (inner nodes), providing a perfect alignment regardless of the dimensionality of the data set⁷. Note that we do not further tailor the node sizes to the sizes of the CPU caches, like the last level cache (LLC), because this did not offer any performance benefits in preliminary evaluations.
- **Multithreading:** We horizontally partition the data at random into almost-equally sized partitions and build one R*-tree instance per partition. When evaluating MDRQ, these are searched in parallel with one thread per partition and the partial result sets are concatenated once all R*-tree instances have finished their work. We choose the number of partitions to be equal to the number of hardware threads available on the evaluation machine. When only some of many partitions contain matches, some threads may finish earlier than others, a condition called *execution skew* [DeWitt et al. 1992]. However, such cases seldomly occur in horizontal partitioning, because of the random assignment of objects to partitions, and are more common in *range partitioning* [DeWitt et al. 1992]. Figure 4.2 illustrates the differences between sequentially querying a conventional R*-tree and executing a parallel MDRQ following our approach.
- **SIMD:** We use the SIMD-based algorithm from Listing 4.1 to compare MDRQ search objects with tuples (MBR) stored in leaf nodes (inner nodes). Since *libspatialindex* works with eight-byte double values, we can compare only four values (instead of eight) with one SIMD instruction.

kd-tree

We implement a kd-tree from scratch following the original proposal by Bentley [Bentley 1975] with the following adaptations to exploit modern hardware features:

- **Main Memory:** Kd-trees are already designed for main memory, thus no adaptations of the data layout were necessary in this regard. As proposed in the original paper, all nodes of a kd-tree store one data object. Hence, also leaf nodes hold only one object, prohibiting adaptations to cache line sizes.

⁶*libspatialindex* - *libspatialindex* 1.8.0 documentation, <https://libspatialindex.github.io/>, Last access: August 29, 2018.

⁷The used implementation stores dimension values as eight-byte double values. Hence, choosing a capacity that is a multiple of eight always aligns nodes to 64-byte cache lines.

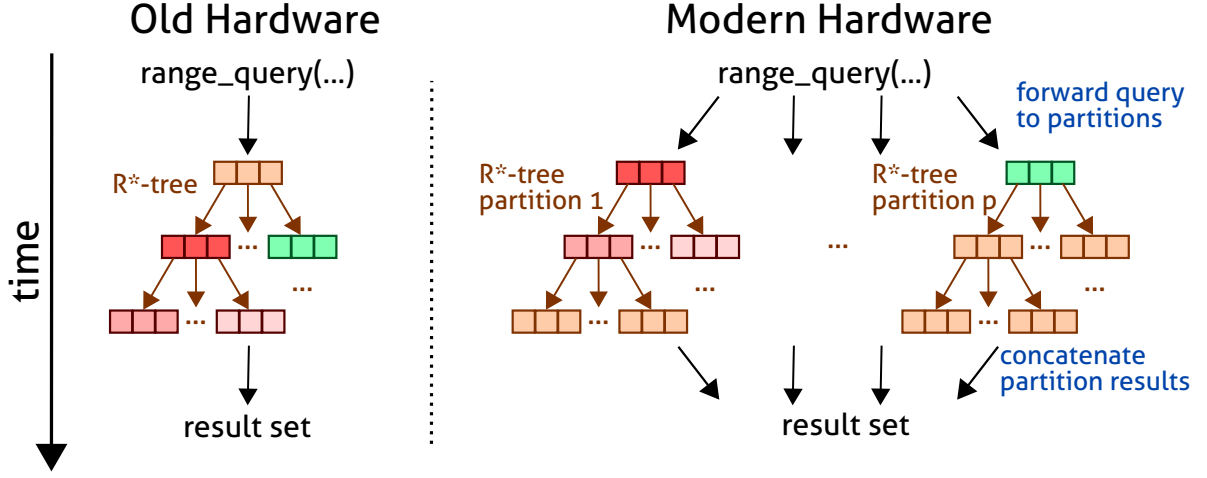


Figure 4.2: Single-threaded execution of an MDRQ on a conventional R*-tree vs. parallel execution of an MDRQ on p instances of an R*-tree, where each instance manages the data of one partition (horizontal partitioning) and is searched with a distinct thread (p threads in total).

- **Multithreading:** We adapt kd-trees to multithreading using the same approach as for R*-trees.
- **SIMD:** We exploit SIMD instructions as shown in Listing 4.1 for intersecting MDRQ search objects with the tuples stored in the nodes of the kd-tree.

VA-file

We also implement VA-files from scratch applying the following adaptations to the original method [Weber et al. 1998]:

- **Main Memory:** All buckets are stored in main memory. The sizes of the buckets are derived from the length of the approximations. Based on preliminary experiments, we use two bits per dimension for the approximations with values roughly evenly dividing the space, leading to buckets holding approximately $n/(4^m)$ tuples.
- **Multithreading:** As for all MDIS, we horizontally partition the data at random and build one VA-file per partition. When evaluating queries, these partitions are concurrently searched.
- **SIMD:** We use the algorithm from Listing 4.1 to compare MDRQ search objects with the tuples from those buckets whose approximations intersect the approximated query.

Parallel Scan on Horizontally-Partitioned Data

For scanning horizontally-partitioned data, we use the following techniques:

- **Main Memory:** We partition the data objects into p partitions, which are kept in m -dimensional arrays.
- **Multithreading:** These arrays are concurrently scanned using a multidimensional range scan (see Section 2.2.1). The results are concatenated once all threads have finished their work.
- **SIMD:** During the scan, every tuple is compared to the query object using SIMD instructions. We replace the inner loop of the multidimensional range scan (see Lines 4-13 from Algorithm 1) with the vectorized algorithm shown in Listing 4.1.

Parallel Scan on Vertically-Partitioned Data

We implement the scanning of vertically-partitioned data as follows:

- **Main Memory:** We partition the data objects into m partitions, which are stored in one-dimensional arrays, each holding the value of one dimension per tuple.
- **Multithreading:** When evaluating MDRQ, these arrays are concurrently searched with one-dimensional range scans. For partial-match queries, only the dimensions addressed in a query are accessed. Note that in this scheme the degree of parallelism of a MDRQ is constrained by the number of dimensions restricted in the query. Each scan of a partition creates one bitmask of length n . These bitmasks are concurrently intersected in t chunks of equal size, where t equals the number of available threads.
- **SIMD:** In vertically-partitioned data, single dimensions of tuples are compared to single dimensions of the MDRQ search object. Hence, the SIMD code from Listing 4.1 cannot be applied. Instead, we apply SIMD instructions using the approach proposed by Zhou and Ross [J. Zhou et al. 2002].

4.4 Evaluation

The objective of our evaluation is to investigate the performance of MDRQ on modern hardware. To this end, we compare three MDIS and two scan variants, which we adapted as described in Section 4.3, applying synthetic and real-world workloads to synthetic and real-world data sets. Specifically, the experiments aim to answer the following questions:

- How much does each contestant benefit from modern hardware features, such as vectorization and multithreading (see Section 4.4.3)?
- Is the rule of thumb for choosing between index probing and scanning depending on the query selectivity still valid on modern hardware (see Section 4.4.4 to Section 4.4.6)?

- What is the impact of data set characteristics, like dimensionality, cardinality, or skew, and workload properties, like selectivity, or number of restricted dimensions, on the performance of the competitors (see Section 4.4.4 to Section 4.4.6)?
- Can the parallel implementations of the contestants scale with the number of used threads (see Section 4.4.7)?
- How efficient do the competitors utilize memory (see Section 4.4.8)?

4.4.1 Experimental Setup

Hardware

We execute all experiments on a server equipped with two Intel Xeon E5-2620 CPUs (2 GHz clock rate, six cores, 12 hardware threads) and 32GB of RAM. In total, the machine features 12 cores and 24 hardware threads (or hyperthreads)⁸. The CPU supports AVX instructions executed on 256-bit SIMD registers.

We also run the experiments on another hardware platform to show that our findings do not depend on the used hardware architecture. The second evaluation machine features one Intel i7-5930K CPU (3.5 GHz clock rate, six cores, 12 hardware threads, AVX instructions) and 32GB of main memory.

Methodology

In all experiments, caches are warmed up. All data sets are inserted in random order. All experiments measure the throughput, which is the number of operations, in our case MDRQ, each contestant can execute per second. In most cases, we run query workloads consisting of 1,000 queries and measure the time t_s each contestant needs to execute all queries. We divide the number of executed queries by t_s to get the average throughput.

Competitors

In our experiments, we consider the R*-tree [Beckmann et al. 1990], the kd-tree [Bentley 1975], the VA-file [Weber et al. 1998] and parallel scans over horizontally- and vertically-partitioned data. All contestants are evaluated with and without multithreading and with and without SIMD instructions. All MDIS employ horizontal partitioning, which allows to re-use single-threaded search operators. Unless otherwise noted, we set $p = t$ for horizontal partitioning to exploit all available processing units (threads) and because we do not expect any stragglers that would benefit from using $p > t$.

⁸As described in Section 2.3.3, modern Intel CPUs provide a proprietary implementation of simultaneous multithreading, called hyperthreading. Depending on the CPU family, each physical CPU core can typically execute between two and four threads concurrently within one instruction pipeline.

Data Set	n	m	Distinct Values per Dimension	Raw Size (MB)
UNIFORM	10k	5	9,950 (avg)	0.19MB
	100k	5	95,175 (avg)	1.91MB
	1M	5 to 100	632,257 (avg)	19.07MB to 381.47MB
	10M	5	999,956 (avg)	190.74MB
CLUSTERED	1M	5	632,047 (avg)	19.07MB
POWER	10k	3	10,000; 627; 698	0.11MB
	100k	3	100,000; 2,089; 2,290	1.14MB
	1M	3	1,000,000; 4,325; 4670	11.44MB
	10M	3	9,875,681; 6,840; 7,634	114.44MB
GENOMIC	10M	19	See Section 2.4.2	724.79MB

Table 4.1: Data sets used in our experiments.

Software

All competitors were implemented in C++11 and were compiled with GCC 4.8 using the optimization flag `-O3`. We use an open-source thread pool library⁹ to enable the reuse of POSIX threads.

4.4.2 Experimental Data and Workloads

We evaluate MDRQ on four different data sets. For each data set, Table 4.1 provides the number of data objects (n), the number of dimensions (m), the number of distinct values per dimension (for synthetic data, we provide average values over all dimensions), and the raw data set size¹⁰. All competitors implement data objects with four-byte floating-point values. Only the used implementation of R*-trees, libspatialindex, employs eight-byte double values. For the data sets UNIFORM, CLUSTERED, and POWER, we use synthetic workloads consisting of randomly-generated complete-match MDRQ. For GENOMIC, we execute realistic complete-match and partial-match MDRQ from the benchmark described in Section 2.4.

Data Set UNIFORM

Synthetic data facilitates experiments with arbitrary cardinalities and dimensionalities. For UNIFORM, we generate values uniformly-distributed at random within $[0, 1]$.

For all experiments with UNIFORM, except those studying the impact of query selectivity (see Section 4.4.4), we generate MDRQ by randomly selecting two data objects from the generated data and use those as lower and upper boundaries. This results in queries with varying selectivities; we always provide the average selectivities.

⁹GitHub - vit-vit/CTPL: Modern and efficient C++ Thread Pool Library, <https://github.com/vit-vit/CTPL>, Last access: August 29, 2018.

¹⁰The presented values equal the amount of memory space needed when implementing data objects as arrays of four-byte floating-point values, i. e., it equals $n * m * 4$ bytes.

Data Set CLUSTERED

In contrast to the uniformly-distributed UNIFORM, the five-dimensional data set CLUSTERED features between one and twenty clusters. For CLUSTERED, we use a data generator provided by Müller et al. [Müller et al. 2009]. Within each cluster, data are uniformly distributed.

For CLUSTERED, we generate range query workloads using the same technique as for UNIFORM.

Data Set POWER

The real-world data set POWER is obtained from the DEBS 2012 challenge¹¹. It contains monitoring data (or events) recorded by hi-tech manufacturing equipment equipped with an embedded PC and various sensors (Industrial IoT). As in previous studies using this data set [Wang et al. 2016], we index three dimensions. The first dimension provides the time at which the event was recorded and can therefore be considered as an almost-unique identifier. The second and third dimension provide information about the power consumption of the monitored device.

As for the synthetic data, we generate MDRQ by randomly choosing two tuples from POWER and use those as lower and upper boundaries.

Data Set GENOMIC

We index real-world genomic variant data provided by the 1000 Genomes Project. Using our own data importer, we transform raw variant data into 19-dimensional data objects. Attributes originally stored as strings, like the population of a sample, are transformed into floating-point values by hashing. Section 2.4.2 provides a detailed description of all attributes including their domain and their number of distinct values.

For GENOMIC, we evaluate eight realistic MDRQ templates designed in collaboration with domain experts (see Section 2.4.3). We also consider a mixed workload that consists of all query templates randomly mixed together.

4.4.3 Impact of Multithreading and Vectorization

Figure 4.3 shows the throughput of MDRQ with an average selectivity of 0.1% ($\sigma = 0.002\%$) when applied to one Million uniformly-distributed tuples of moderate dimensionality (twenty dimensions) depending on the used modern hardware features. For all contestants, we evaluate a single-threaded (baseline) implementation, a single-threaded implementation exploiting SIMD instructions, a multithreaded implementation, and a multithreaded implementation using SIMD instructions.

Multithreading

We compare the single-threaded to the multithreaded implementations (both without using SIMD), where we used 24 software threads, which equals the number of hardware threads

¹¹DEBS 2012 Grand Challenge: Manufacturing equipment - DEBS.org, <http://debs.org/?p=38>, Last access: August 29, 2018.

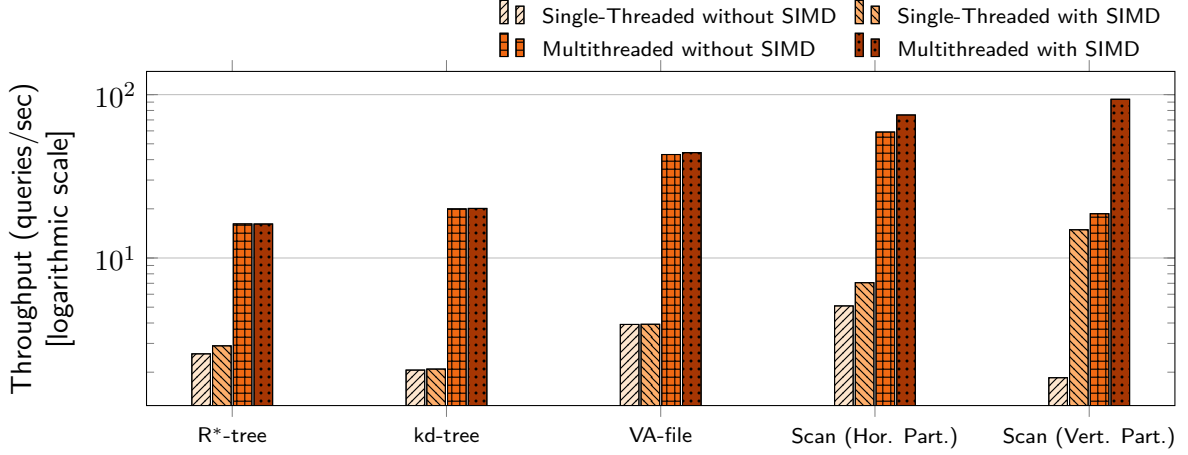


Figure 4.3: Throughput when executing MDRQ with an average selectivity of 0.1% on one Million twenty-dimensional tuples from UNIFORM depending on the used hardware features.

available on the evaluation machine. Due to a dimensionality of $m = 20$, the scan with vertical partitioning uses only twenty threads.

At first glance one would expect speed-ups almost as good as 24X, at least for the approaches using horizontal partitioning. However, none of the contestants shows such performance gains. On average, the competitors achieve speed-ups of up to 11.6 times. Apparently, performance gains are bounded by the number of physical cores, which equals 12 on the evaluation machine. Hyper-threading [Saini et al. 2011] is only beneficial for applications, where threads are frequently waiting for data to be loaded from the main memory into the CPU caches, making memory accesses the bottleneck [Plattner 2009]; this is also described as *blocking*.

The workload considered here uses query selectivities of only 0.1%, which enables hierarchical MDIS (R*-tree and kd-tree) to efficiently prune large parts of the data space. Thus, they are compute-bound. Even both scans and the VA-file are not memory-bound, because many early breaks occur, leading to branch mispredictions and causing compute-intensive pipeline flushes. See Section 4.4.7 for scalability experiments using lower query selectivities.

Vectorization

We study the impact of SIMD instructions on the performance of the competitors. Approaches employing horizontal partitioning show almost no performance gains when applying vectorization (only up to 1.4 times). While scalar variants of the search algorithms can prune comparisons once the first mismatch occurred (early break), the vectorized counterpart of the same algorithm has to process objects in chunks of eight dimensions, limiting the potential for early breaks. For query workloads with a lower selectivity and therefore fewer mismatches (or early breaks), the benefits from SIMD-based processing of horizontally-partitioned tuples increase, providing a speed-up of up to three times.

Scans using vertical partitioning notably benefit from SIMD instructions, because both SIMD-

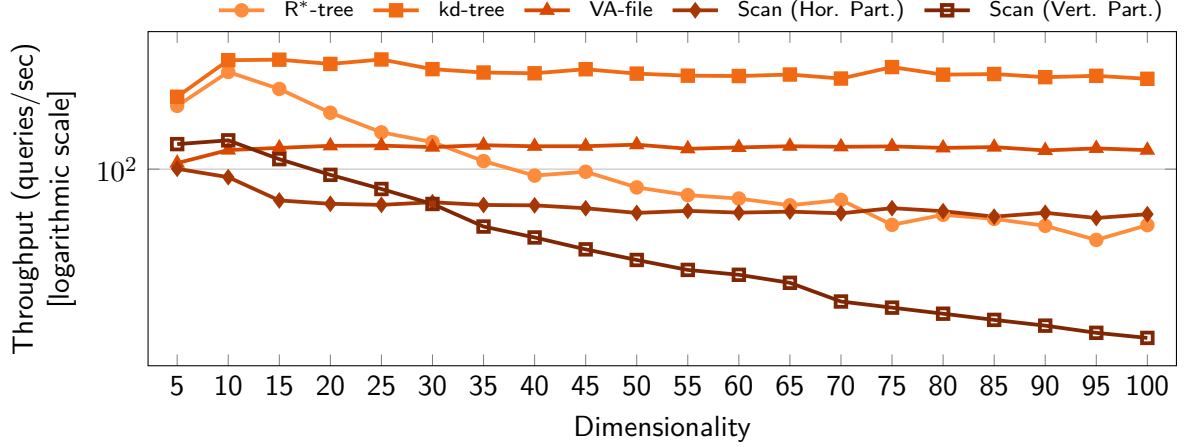


Figure 4.4: Throughput when executing MDRQ with an average selectivity of 0.4% (five dimensions) to 0.0002% (more than ten dimensions) on one Million tuples from UNIFORM using 24 software threads depending on the dimensionality.

based and scalar variants of the search algorithms must consider all data and can neither apply pruning nor use early breaks. Here, vectorization helps to strongly reduce the number of comparisons and leads to a speed-up of up to eight times on average.

Multithreading combined with Vectorization

Finally, we investigate the performance impact when applying both multithreading and vectorization. In this case, scans over vertically-partitioned data show the largest speed-ups compared to a single-threaded scalar implementation (up to 50 times faster on average). The remaining approaches, which all employ horizontal partitioning, benefit from multithreading but show very small performance gains when using SIMD instructions on top. They achieve speed-ups of up to 14 times on average. All subsequent experiments investigate the competitors when using both multithreading and vectorization.

4.4.4 Synthetic Data

The following experiments are conducted with synthetic data from UNIFORM and CLUSTERED.

Dimensionality

We measure the throughput of the contestants when applying MDRQ to one Million randomly-generated tuples with five to 100 dimensions. Clearly, query selectivity increases with a growing dimensionality. The average query selectivity equals 0.4% ($\sigma = 1.1\%$) for five dimensions, 0.002% ($\sigma = 0.01\%$) for ten dimensions, and 0.0002% ($\sigma = 0.00003\%$) for more than ten dimensions. Figure 4.4 shows the results.

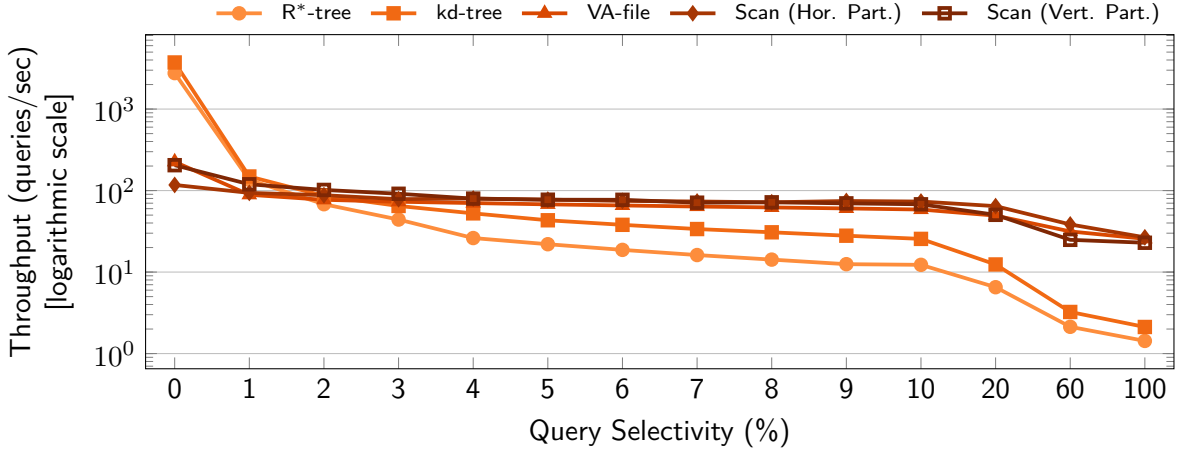


Figure 4.5: Throughput when executing range queries on one Million five-dimensional tuples from UNIFORM using 24 software threads depending on the query selectivity.

The kd-tree achieves the highest throughput among all contestants regardless of the dimensionality. For up to 30 dimensions, the R*-tree shows the second best performance. However, with an increasing dimensionality, the search efficiency of the R*-tree decreases, because its partitioning deteriorates, reducing its pruning power. For data sets with more than 30 dimensions, the R*-tree is outperformed by the VA-file, which is less affected by the dimensionality of the data set. For high dimensionalities (more than 70 dimensions), the R*-tree is even outperformed by scans over vertically-partitioned data, although the average query selectivity is very high.

While the throughput of the parallel scan employing horizontal partitioning is independent from the dimensionality of the data space, the performance of the scan using vertical partitioning decreases when the dimensionality increases. In vertical partitioning, the number of partitions depends on the dimensionality of the data space ($p = m$), whereas in horizontal partitioning the number of partitions is chosen depending on the number of available hardware threads. In particular, the performance of the scan using vertical partitioning decreases with an increasing dimensionality of the data space, because a growing number of partial result sets need to be managed and intersected when synchronizing the different threads.

Query Selectivity

We measure the throughput of all contestants when executing MDRQ on one Million randomly-generated five-dimensional tuples following an uniform distribution depending on the query selectivity. Figure 4.5 shows the results.

For queries with a very high selectivity ($\leq 1\%$), the kd-tree shows the highest throughput and is closely followed by the R*-tree. For queries with a lower selectivity ($> 1\%$), both parallel scans as well as the VA-file show the best performance and clearly outperform the hierarchical MDIS kd-tree and R*-tree. In this experiment, both scan variants and the (non-hierarchical) VA-file perform very similar.

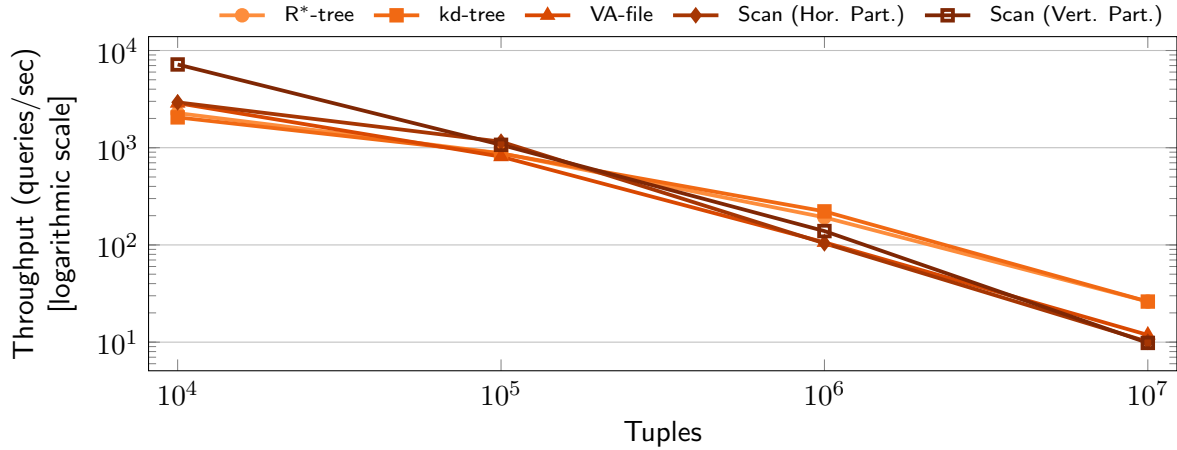


Figure 4.6: Throughput when executing range queries with an average selectivity of 0.4% on five-dimensional tuples from UNIFORM using 24 software threads depending on the size of the data set.

For multiple reasons, the throughput of all contestants decreases for less selective queries:

- In vertical partitioning, larger (partial) result sets need to be managed and synchronized, requiring a costly intersection.
- In horizontal partitioning, approaches can prune less dimensions when comparing MDRQ with tuples, resulting in fewer early breaks.
- Hierarchical MDIS (R*-tree and kd-tree) cannot prune subtrees but must visit the vast majority of the tree nodes, which requires lots of random accesses inducing cache misses.

While scans are less affected by query selectivities, hierarchical MDIS lose their pruning power when dealing with moderate and low selectivities. On modern hardware, the threshold after which MDIS are outperformed by scans is stupendously low, at a selectivity around 1%. Hence, it is much lower than on disk-based systems, where MDIS show a superior performance for selectivities of up to 20% [Weber et al. 1998].

Data Set Size

We measure the throughput when executing MDRQ with an average selectivity of 0.4% ($\sigma = 1.1\%$) on five-dimensional tuples following an uniform distribution depending on the size of the data set. Figure 4.6 shows the results.

As expected, when the number of tuples increases, the search throughput of all contestants decreases, because a growing number of tuples match the query. Interestingly, both parallel scans, especially the variant employing vertical partitioning, outperform MDIS for small data sets consisting of up to 10^5 tuples, although the average selectivity of the queries evaluated here is very high. MDIS are not worthwhile for such small amounts of data but suffer from

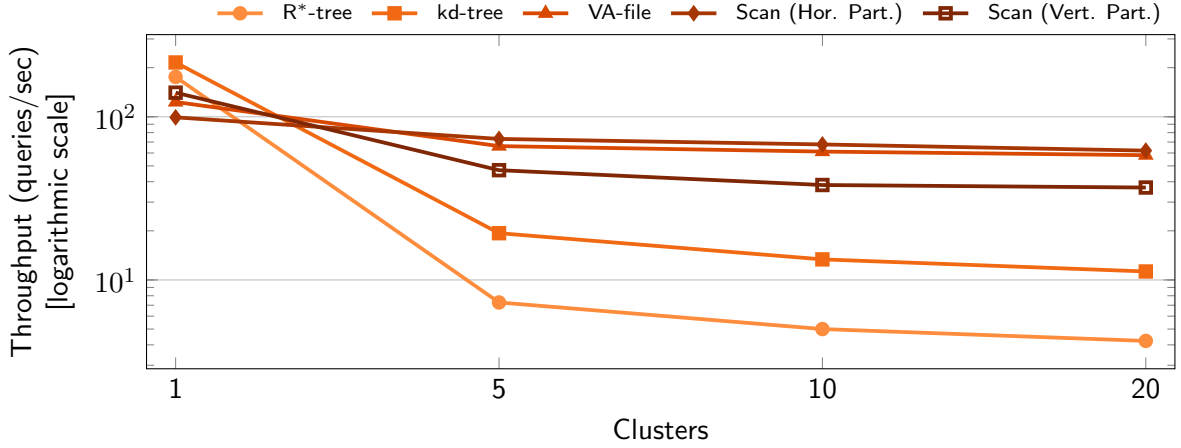


Figure 4.7: Throughput when executing range queries with an average selectivity of 0.38% (one cluster) to 27.40% (20 clusters) on one Million five-dimensional tuples from CLUSTERED using 24 software threads depending on the number of clusters.

the overhead induced by their index structure. When the data set size increases, the pruning capabilities of MDIS pay off: MDIS can efficiently reduce the data space while the parallel scans have to consider all tuples for query evaluation.

Clustered Data

We measure the throughput when applying MDRQ to one Million five-dimensional data objects from the data set CLUSTERED depending on the number of clusters. Recall that we are generating MDRQ by randomly picking two existing tuples as range boundaries. Thus, for CLUSTERED, one MDRQ may cross several clusters, which results in a decreasing query selectivity as the number of clusters increases: one cluster induces an average selectivity of 0.4% ($\sigma = 0.9\%$), five clusters induce an average selectivity of 16.2% ($\sigma = 19.1\%$), ten clusters induce an average selectivity of 23.1% ($\sigma = 21.9\%$), and 20 clusters induce an average selectivity of 27.4% ($\sigma = 22.7\%$). Figure 4.7 shows the results of this experiment.

Note that the data set with one cluster is very similar to UNIFORM, because the data objects are uniformly distributed across the entire data space. For that case, the R*-tree and the kd-tree achieve the best throughput among all competitors. When the number of clusters increases (which also implies decreasing query selectivities), the performance of hierarchical MDIS, the R*-tree and the kd-tree, decreases. Regardless of the number of clusters, the performance of the VA-file and both parallel scans remains almost the same.

We study whether the performance decrease of the MDIS is caused by the increase of the query selectivity or by the increase of the number of clusters. For this purpose, we generate range queries that retrieve only one single object, thus having an average selectivity of 0.0001% ($\sigma = 0\%$). Using this technique, we can ensure that the selectivities are the same for all instances of CLUSTERED regardless of the number of featured clusters. When applying this workload, all competitors show a very similar performance regardless of the number of clusters. Thus, their

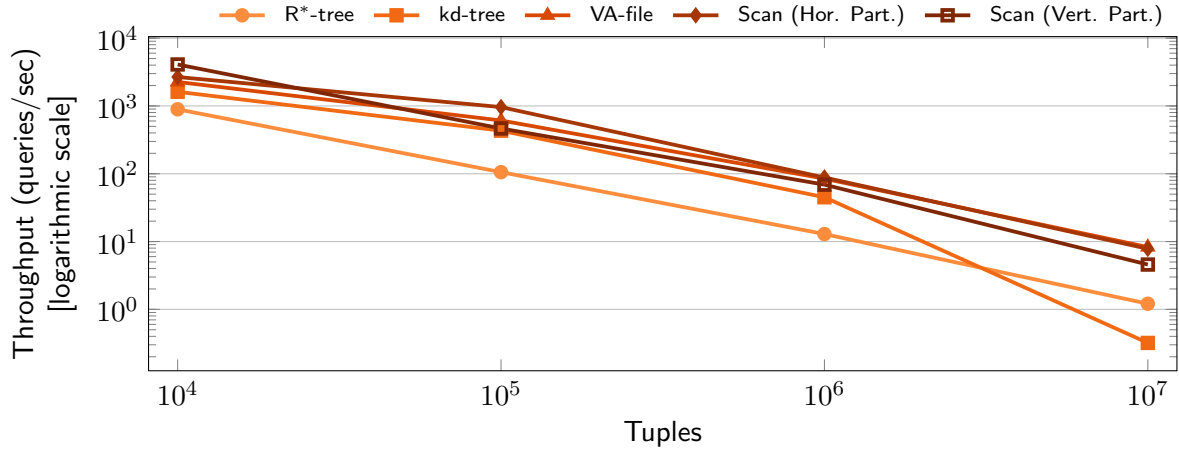


Figure 4.8: Throughput when executing range queries with an average selectivity of 11.12% on three-dimensional tuples from POWER using 24 software threads depending on the size of the data set.

performance is mainly affected by the query selectivities but not by the number of clusters.

4.4.5 Sensor Data from Hi-Tech Manufacturing Equipment

We measure the throughput of the contestants when applying MDRQ with an average selectivity of 11.12% ($\sigma = 13.43\%$) to the three-dimensional data set POWER. In contrast to Figure 4.6, which shows the throughput for uniformly-distributed data, Figure 4.8 visualizes the throughput of MDRQ on real-world data of varying size. This experiment confirms that the throughput of all contestants decreases when the number of tuples increases. In contrast to the experiments on synthetic data, the scan-based approaches always outperform the hierarchical MDIS regardless of the data set size. However, this experiment executes queries that are less selective than the ones applied to the synthetic data, as induced by the technique used to generate MDRQ workloads, which mainly causes the performance differences.

4.4.6 Genomic Variant Data

The following experiments are conducted with real-world data from GENOMIC. We study the contestants when executing the GMRQ Benchmark (see Section 2.4.3). Each template is instantiated one hundred times. Figure 4.9 shows the results.

Both parallel scans outperform all considered MDIS for the Mixed Workload, Query Template 1, Query Template 2 and Query Template 3. For Query Templates 4 to 8, which retrieve only few tuples and have a selectivity much below 1%, especially the kd-tree shows its strengths and outperforms scanning. The VA-file performs similar as the scan with horizontal partitioning, but is consistently outperformed by the scan with vertical partitioning.

As in Section 4.4.4, which studied the performance when executing MDRQ of varying selectivities on uniformly-distributed data, scanning outperforms the MDIS for low and moderate

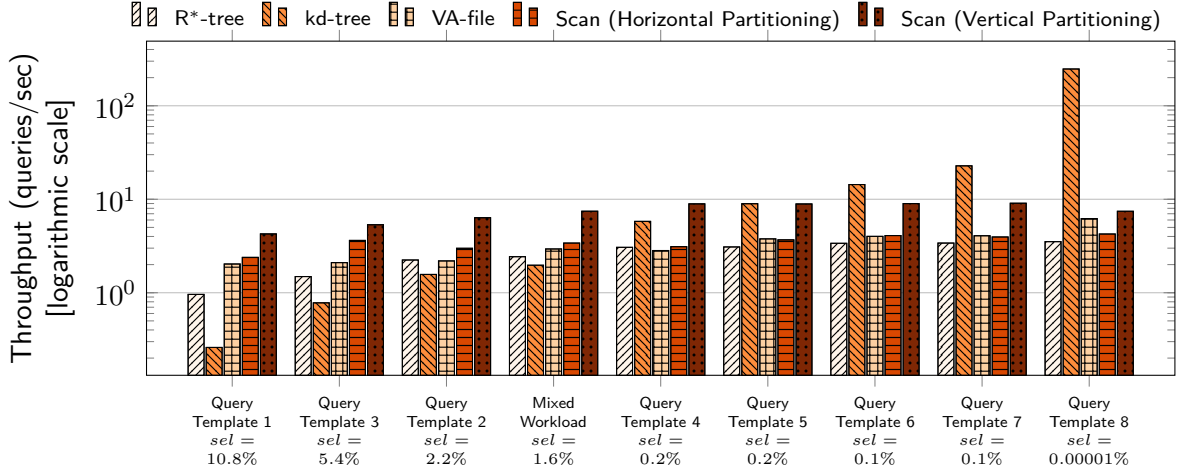


Figure 4.9: Throughput of the contestants when executing the GMRQB on ten Million 19-dimensional tuples from GENOMIC using 24 software threads (the query templates are ordered by average selectivity, from low (left) to high (right)).

selectivities. In both cases (synthetic and real-world data), the threshold after which scans are superior to MDIS is at a query selectivity around 1%.

The Mixed Workload consists of all query templates randomly mixed together and has an average selectivity of 1.58% ($\sigma = 3.58\%$). For this workload, the parallel scan with vertical partitioning achieves the highest throughput (7.45 queries/sec) and is followed by the parallel scan employing horizontal partitioning (3.40 queries/sec), the VA-file (2.94 queries/sec), the R*-tree (2.43 queries/sec), and the kd-tree (1.97 queries/sec). For Query Template 1, which has the lowest selectivity (on average 10.76%, $\sigma = 7.24\%$) among all evaluated queries, both parallel scan variants show the best performance and are followed by the VA-file, the R*-tree, and the kd-tree. For Query Template 8, which has the highest selectivity (on average 0.00001%, $\sigma = 0.00002\%$), the kd-tree achieves a throughput of 247.46 queries/sec. It is followed by the parallel scan with vertical partitioning (7.44 queries/sec), the VA-file (6.22 queries/sec), the parallel scan with horizontal partitioning (4.25 queries/sec), and the R*-tree (3.52 queries/sec). Despite the high selectivity, the R*-tree cannot apply its pruning capabilities, because it is negatively affected by the dimensionality of the data set, which leads to an inefficient partitioning.

4.4.7 Scalability

Using the Mixed Workload from GMRQB, we evaluate the scalability of all contestants depending on the number of used software threads. Note that the workload consists of partial- and complete-match MDRQ that query 5.81 dimensions on average ($\sigma = 4.11$), which limits the potential benefits from multithreading for vertical partitioning. Figure 4.10 shows the results. For most contestants, the speed-up from multithreading is bounded by the number of physical cores. Confirming a memory access bottleneck, only the R*-tree and the kd-tree benefit from using more software threads than available physical cores, because hierarchical MDIS require

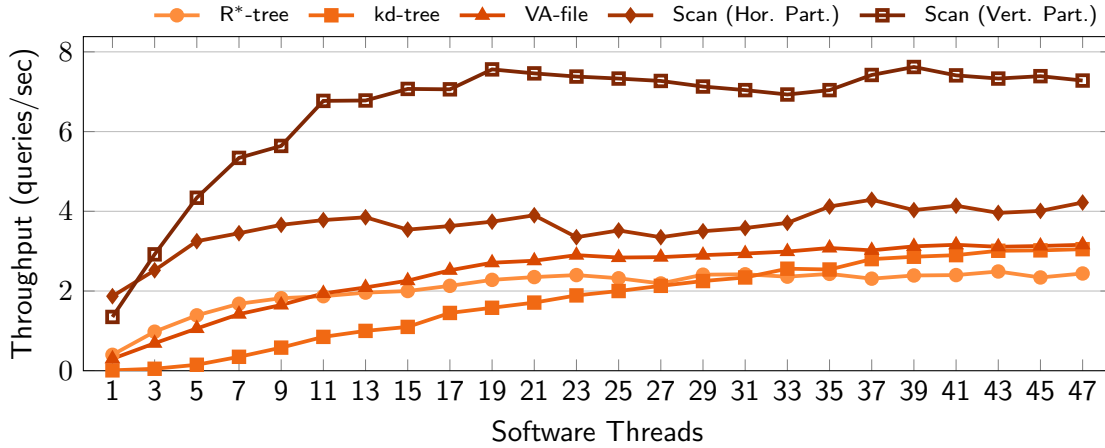


Figure 4.10: Throughput of the contestants when applying the mixed workload from GMRQB to ten Million 19-dimensional tuples from GENOMIC depending on the number of used software threads.

	UNIFORM (n=10M, m=5)	CLUSTERED (n=1M, m=5)	POWER (n=10M, m=3)	GENOMIC (n=10M, m=19)
R*-tree	1,037MB	110MB	833MB	3,057MB
kd-tree	1,567MB	159MB	1,644MB	2,864MB
VA-file	2,804MB	284MB	2,882MB	4,140MB
Scan (Hor. Part.)	537MB	57MB	538MB	1,169MB
Scan (Vert. Part.)	547MB	57MB	540MB	1,169MB

Table 4.2: Space consumption of the competitors.

many random accesses when evaluating queries having a moderate or low selectivity. When executing the mixed workload from GMRQB with one software thread, on average, R*-trees produce 20 Million LLC misses and kd-trees produce 16 Million LLC misses per executed query. In contrast, VA-files produce two Million LLC misses and both parallel scans produce 800,000 LLC misses. Thus, hierarchical MDIS can gain additional performance from hyper-threading, which helps to hide the latency of memory accesses to some degree. Using more software threads than available virtual CPU cores (more than 24) does neither yield performance benefits nor disadvantages.

4.4.8 Space Consumption

Figure 4.11 visualizes the memory consumption of each contestant depending on the data set and Table 4.2 provides the according numbers. For horizontal partitioning, we build 24 instances of an index or array, i.e., $p = 24$. For vertical partitioning, we create as many partitions as dimensions present in the data set. We measure the combined space consumption of all partitions. For UNIFORM, we show the memory consumption when storing ten Million five-

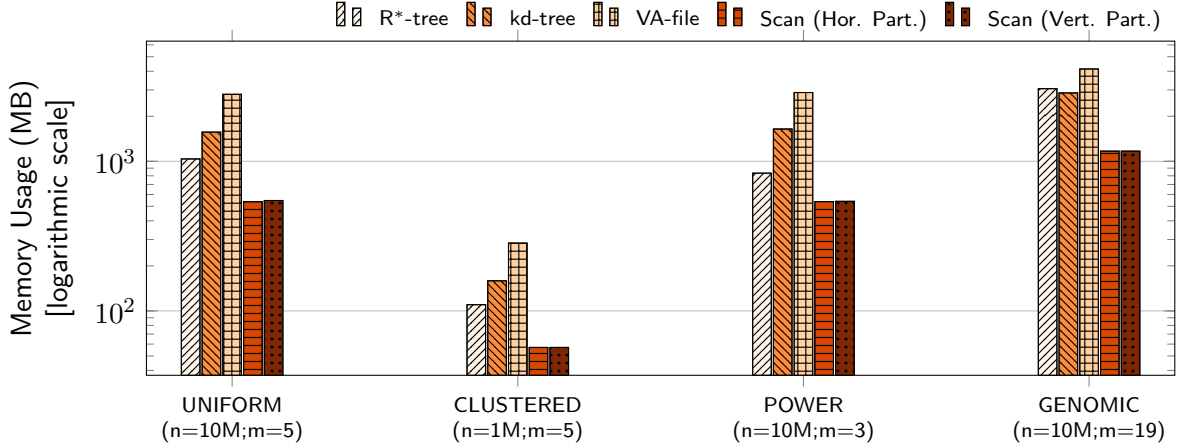


Figure 4.11: Memory usage of the competitors.

dimensional tuples. For CLUSTERED, we show the memory consumption when storing one Million five-dimensional tuples featuring twenty clusters. For POWER, we show the memory consumption when storing ten Million three-dimensional tuples. For GENOMIC, we show the memory consumption when storing ten Million 19-dimensional tuples.

The used implementation of R*-trees implements data objects with eight-byte double values, while all remaining competitors use four-byte floating-point values. To enable a fair comparison, we divide the measured memory usage of the R*-tree by two and report the resulting value. Please note that our VA-file implementation stores the approximation values in eight-bit integers (the smallest available data type) although only two bits are needed (recall that we create four partitions per dimension), thus wasting some memory space. The provided numbers take into account that approximation values are implemented with two-bit values.

As expected, both scans need the least amount of memory, because they do not employ additional data structures, like MDIS. Surprisingly, VA-files require much more space than scans, induced by the storage of the approximations and information about the partitioning. Comparing both hierarchical MDIS, R*-trees achieve a higher space efficiency than kd-trees. Only for GENOMIC, where the structure of the covering R*-tree degenerates due to an inefficient partitioning, kd-trees require less space than R*-trees.

4.4.9 Other Evaluation Platform

We also ran the experiments on another hardware platform to show that our findings do not depend on the used hardware architecture. We used a modern desktop machine that features one Intel i7-5930K CPU (3.5 GHz clock rate, six cores, 12 virtual cores) and 32GB of main memory. The CPU supports AVX instructions. The results on both hardware platforms are very similar, which confirms that our findings carry over to other hardware settings.

As exemplary results for the experiments on the desktop machine, Figure 4.12 shows the throughput when executing the query templates from GMRQB on ten Million 19-dimensional tuples from GENOMIC. The query templates are instantiated with the same values as in Section

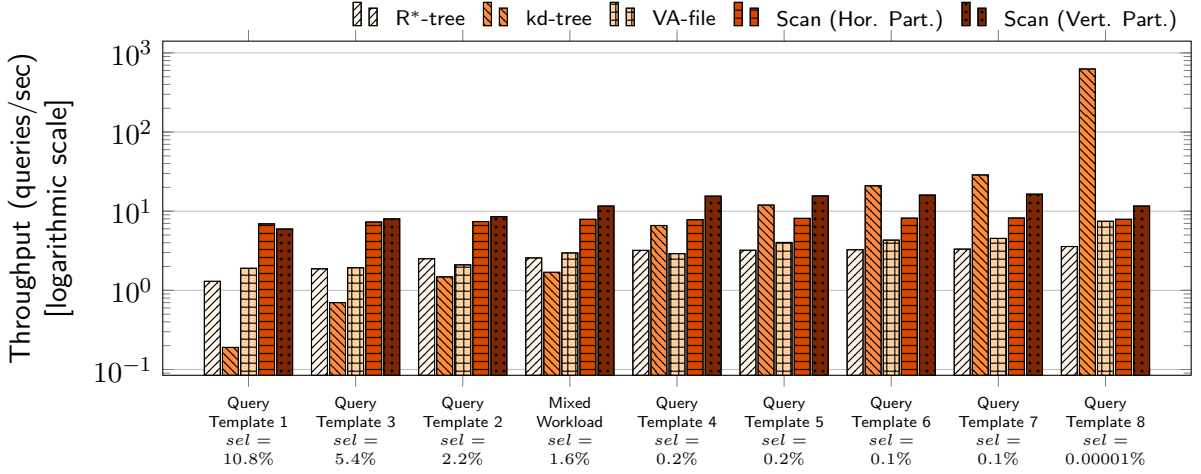


Figure 4.12: Throughput of the contestants on the desktop machine when applying the GMRQB to ten Million 19-dimensional tuples from GENOMIC using 12 software threads (the query templates are ordered by average selectivity, from low (left) to high (right)).

4.4.6. In contrast to the experiment on the primary evaluation machine, horizontal partitioning uses only 12 software threads because the desktop machine features less virtual cores. Again, the parallel scan with vertical partitioning achieves the highest throughput for queries with a moderate to low selectivity (Mixed Workload and Query Templates 1-5). Only for queries with a very high selectivity much below 1% (Query Templates 6-8), it is outperformed by the kd-tree.

4.5 Discussion

Our comparison of the two hierarchical, tree-based MDIS (the R*-tree and the kd-tree), the VA-file, and the two parallel scan variants after adapting them to the usage of main-memory storage, multithreading and SIMD instructions yields a number of interesting observations. As expected, MDIS in general excel for queries with very high selectivities as in such settings they can prune substantial parts of the search space and have to compare only a few data objects to the search object. The main goal of our study was to re-evaluate the break-even point at which these advantages supersede the major disadvantage of MDIS, namely accesses to random memory locations. Across various data sets, our experiments show that this point is surprisingly low, at around 1% selectivity, and thus much lower than the conventional rule of thumb (20%), which targeted IO-based index structures. Although similar findings have been reported for one-dimensional range scans on modern hardware [Das et al. 2015], to the best of our knowledge, we are the first to confirm these performance characteristics for the multidimensional domain.

Following the results of our study, scanning should be favored over indexing except for very selective queries. For horizontal partitioning, highly-selective queries are fast anyway, because many early breaks occur, which means that the absolute savings in time MDIS offer in such settings are very small. On top, scan-based MDRQ are much easier to handle and to implement,

require no additional storage, are almost unaffected from the dimensionality of the data, lead to simple and effective load balancing, and offer predictable runtimes, which is a major advantage when it comes to orchestrating the various operations of a complex analytical query.

Although our study focused on the parallelization of search queries, the proposed partitioning techniques can also be used to speed-up updates¹², which either add new data or remove an existing object. When inserting a new data object into a set of horizontally-partitioned data, we randomly select one of the partitions and apply the insertion to it; the remaining partitions must not be touched. In contrast, to add a new data object to a set of vertically-partitioned data, we split the object among its dimensions and add each value to the corresponding partition, i. e., insertions affect all partitions. When removing an existing data object from a set of horizontally-partitioned data, we pass the delete operation to all partitions. Concurrently, all partitions apply the deletion to their data and remove the object if found. Similarly, to remove an object from a set of vertically-partitioned data, we forward the delete operation to all partitions, which concurrently process it on their data. Horizontal partitioning enables efficient updates, because they involve only one partition¹³. In vertical partitioning, although updates always regard all partitions, they are executed in parallel, thus also allowing for an efficient implementation.

There are also a number of further observations:

- Among the two hierarchical MDIS in our evaluation, the kd-tree clearly outperforms the R*-tree, which is not too much of a surprise as R*-trees were originally designed to manage spatial objects and not points as was the case here. In particular for data sets of moderate and high dimensionality (we evaluated up to one hundred dimensions), kd-trees are superior.
- In Section 4.4.4, we studied synthetic data sets with varying numbers of subspace clusters and showed that the skewness of data does neither impact scans nor MDIS. Instead, query selectivity is the primary factor affecting the performance of the competitors.
- The used R*-tree implementation is the only competitor that stores data objects as arrays of eight-byte values; all other contestants use four bytes per dimension. As less values fit into one SIMD register, R*-trees need to execute twice as much SIMD instructions as the other competitors when searching (see Section 4.2). However, the implementation of data objects has negligible impact on the search performance, because SIMD parallelism does not provide huge performance benefits for horizontal partitioning anyway (see Section 4.4.3).
- The efficiency of VA-files depends on the accuracy (or length) of the approximation values. While more accurate approximations provide a more fine-grained partitioning increasing pruning capabilities, they also need more bits for storage decreasing space efficiency. In our evaluation, we used a pragmatically-chosen configuration for the approximation lengths

¹²We consider an update operation to affect only one data object; duplicates are not allowed.

¹³Note that deletions are passed to all partitions, because we do not maintain knowledge about which data object is stored in which partition, yet the to-be-deleted object is stored in only one partition, allowing efficient pruning in all other partitions.

that balances pruning power with space utilization. Despite their pruning techniques, VA-files offer almost no advantages when compared to scans. They appear to be a sensible choice only for data with very high dimensionalities.

- When comparing the two scan methods, horizontal partitioning is preferable for complete-match queries, whereas partial-match queries, especially when addressing only a few dimensions of the data space, are handled more efficiently by scans over vertically-partitioned data.

However, when considering our results, one should always keep in mind that our adaptations of the index structures and scans were rather conservative. We already sketched several ideas how the different methods could be further improved to take full advantage of modern hardware:

- We could improve the comparisons of MDRQ search objects with data objects by re-ordering the dimensions of the data space such that the dimensions, which are typically queried with the highest selectivity, are compared first. That would facilitate early breaks, allowing the pruning of (many) dimensions once a mismatch occurs. Selectivity-based re-ordering of the dimensions can be applied to both MDIS and scan approaches and is especially beneficial for high-dimensional data spaces, where one data object spans multiple cache lines (when using four-byte floating-point values, a data object with more than 16 dimensions spans multiple cache lines assuming 64-byte cache lines). In such cases, it can reduce the number of accessed cache lines, effectively improving the search performance. However, such an optimization requires knowledge about the current average single-dimensional selectivities. Frequently-changing workloads would require frequent reorganizations of the ordering of the dimensions, causing maintenance overhead.
- In the current scheme of vertical partitioning, we assign one software thread to each partition for query processing. As the number of partitions depends on the dimensionality of the data space, it is very likely that we do not obtain the perfect degree of parallelism, especially when a data space features less dimensions than threads available on a computing machine. We could improve the resource utilization by introducing another partitioning level. At the first level, vertical partitioning would work as usual: We create one partition for each dimension of the data space. However, when more threads are available than dimensions featured in the data space, we assign multiple threads to each partition, allowing a further partitioning of the one-dimensional partitions into different chunks, where each chunk is processed by a distinct thread.
- Although main-memory capacities are considerably large these days, it remains important to use as little space as possible for storage. We could improve the space efficiency of the competitors by compressing data objects. Compression is especially useful for vertically-partitioned data, as demonstrated by different schemes available for column stores [D. J. Abadi et al. 2006; Raman et al. 2013; Stonebraker et al. 2005]. Depending on the concrete data distribution, we might choose between multiple compression techniques [D. Abadi et al. 2013]. We describe two examples.

Run-length encoding (RLE) compresses long *runs* of the same value. For instance, given that the value 42 occurs 50 times in a row, the 50 instances of 42 are replaced by (42, 50). RLE would be beneficial for dimensions containing many duplicate values, reducing these to single representations. Another popular compression technique is *dictionary encoding*, which maintains the different elements of a set in a dictionary and replaces the instances of these values with references to the corresponding entries in the dictionary. Dictionary encoding would be especially useful for dimensions with low cardinalities. Interestingly, compression does not prohibit the usage of vectorized instructions, as shown by Willhalm et al. for column scans [Willhalm et al. 2009].

- In this analysis, we used the original implementation of kd-trees as proposed in [Bentley 1975]. We may improve the cache line utilization of their search algorithms as follows. Instead of storing data in both inner and leaf nodes, we might exclusively keep data objects in leaf nodes, similar to K-D-B-trees [Robinson 1981]. Finally, we would align the capacity of the nodes to the sizes of cache lines, an adaptation technique previously applied to one-dimensional index structures [Rao et al. 2000].

4.6 Summary

In this chapter, we studied how to adapt the execution of MDRQ to current hardware architectures featuring large main-memory capacities and multi-core CPUs with SIMD instructions. We proposed two techniques for dividing a multidimensional data set into multiple partitions, namely horizontal and vertical partitioning. While horizontal partitioning offers more flexibility with regards to the granularity of the partitioning leading to a better resource utilization, vertical partitioning is superior for partial-match MDRQ. We presented a novel SIMD-based algorithm for comparing MDRQ search objects with data objects. The algorithm is beneficial for data spaces with moderate to large dimensionalities and queries with low selectivities.

We conservatively applied these techniques to three index structures, namely the R*-tree, the kd-tree, and the VA-file, and two scan flavors. In a comprehensive evaluation, we compared the competitors executing synthetic and real-world complete-match and partial-match MDRQ workloads on synthetic and real-world data sets. We showed that, on current hardware architectures, scanning, which uses a sequential access pattern, outperforms MDIS probing, which requires random accesses, for query selectivities of 1% or more. Our results are in contrast to previous rule of thumbs [Weber et al. 1998], which are only valid for traditional disk-based systems, indicating that the selectivity threshold is at around 20%. Based on these findings, the next chapter proposes a novel multidimensional index structure, the BB-Tree. The BB-Tree is designed to be stored in main memory, provides a high cache efficiency due to an almost-sequential access pattern, allows updates and exploits the parallel capabilities of modern processors.

5 BB-Trees: Processing Multidimensional Range Queries in Main Memory

The experiments presented in the previous chapter demonstrated that neither multidimensional index structures (MDIS) nor full-table scans can be considered as perfect means to executing multidimensional range queries (MDRQ). Instead, like for single-column predicates [Das et al. 2015], database systems must choose access paths at query time depending on the concrete selectivity of a MDRQ: Queries with high selectivities are evaluated with MDIS, and queries with low selectivities are processed using scans.

In practice, selectivity-based access path selection has the main drawback that, at the time of query evaluation, it requires accurate estimates of the number of matching tuples. Typically, database management systems estimate query selectivities using techniques, like histograms [Kooi 1980; Poosala 1997] or sampling [Lipton et al. 1990]. Traditional histograms designed for one-dimensional data are highly inaccurate when applied to multidimensional data spaces, where dimensions can be correlated [Poosala et al. 1997]. Dedicated multidimensional histogram techniques can better represent joint data distributions, but are expensive to build and maintain [Chakrabarti et al. 2001]. On the other hand, sampling is not affected by the dimensionality of the data, but requires large sample sizes to correctly reflect skewed data sets incurring considerable performance overhead at query time. Therefrom, selectivity-based access path selection is impractical when applying MDRQ to large sets of non-uniformly distributed data.

The aim of research in MDIS is thus to create an index structure that can efficiently process search queries of both high and low selectivity, making access path selection for MDRQ superfluous. Motivated by the recent changes in server hardware described in Section 2.3, the index structure should be tailored to modern hardware architectures, show high space efficiency when stored in main memory and leverage the parallel capabilities of modern CPUs. To support a wide range of different applications, it should offer robustness towards different data characteristics, e. g., cardinality, dimensionality, or skew, and perform gracefully in mixed read and write workloads.

This chapter presents the BB-Tree, a novel fast and space-efficient MDIS supporting point and range queries over multidimensional point data stored in main memory. Conceptually, BB-Trees are *almost-balanced* k -ary search trees. Inner nodes recursively split the data space into k partitions according to a delimiter dimension and $k - 1$ delimiter values, and leaf nodes are responsible for storing data objects. When too many data points are inserted (or deleted) and leaf nodes overflow (or underflow), the *inner search tree* (IST) is rebuilt to achieve a beneficial balance regarding the depths of leaves. Search algorithms use the inner nodes to narrow down the leaf nodes, which may hold objects matching the query object. These leaf nodes are evaluated with a sequential scan to find the true results. Within this general and well-known layout, the

BB-Tree employs a number of novel techniques that yield its superior performance:

- As first contribution, BB-Trees introduce the *bubble buckets* (BB), which are elastic leaf nodes that can efficiently handle strongly fluctuating node fill degrees and that strongly reduce the frequency of rebalancing operations. BB automatically morph between different representations depending on their number of stored data objects. We distinguish between *regular* and *super* BB. Regular BB can hold up to b_{max} data objects and are implemented with dynamic arrays. Super BB are composites and consist of a routing node and a set of k regular BB. Hence, super BB locally add a further level to the tree, introducing slight imbalance.

BB can dynamically grow and shrink: Overflowing regular BB let them morph into super BB, and underflowing super BB let them morph back into regular BB. Both operations leave the rest of the BB-Tree unchanged. Since overflows of regular BB create k new leaf nodes, a BB can cater for a rather large number of inserts. BB increase the robustness towards *hammered inserts*, i.e., series of insertions into the same small region of the space. Here, BB significantly reduce the pressure on rebalancing and thus greatly improve the performance of writes with only minimal influence on the query performance, due to a locally slightly deeper tree.

- As second contribution, we adapt the capacities of inner nodes to the sizes of cache lines, the basic unit of data transfers between main memory and on-die CPU caches. We always choose the fan out of inner nodes, k , depending on how many delimiter values fit into one cache line to improve cache line utilization. For instance, when implementing delimiter values with four-byte floats and running on a machine with 64-byte cache lines, k is set to 17.

We store the inner nodes of the BB-Tree in a flat and immutable array to avoid pointer chasing during search, to decrease random access patterns, and thus to reduce cache misses, especially at the last cache level. Typically, such an optimization either makes the index structure completely static [Schlegel et al. 2009; C. Kim et al. 2010] or raises the need to maintain updates in delta stores [Levandowski et al. 2013; Plattner 2009]. In contrast, the elasticity of BB enables BB-Trees to manage most changes in-place and allows to process a large number of updates without requiring index rebuilds. Eliminating pointers additionally increases space efficiency.

- Our third contribution is an efficient technique for the parallelization of MDIS queries, which effectively avoids difficult-to-maintain data partitioning. In the parallel BB-Tree, search queries are evaluated by first navigating the IST to determine all buckets that may hold matching data objects. This step is performed by a single thread, as the tree, due to its high fan out, is quite low even for very large data sets. In the next step, which strongly dominates the runtime of queries as shown in Section 5.7.5, all qualifying BB are scanned in parallel. The parallel BB-Tree can scale its performance with the number of physical cores without requiring a complex load balancing scheme.
- In a comprehensive evaluation, we compare the BB-Tree to sequential and parallel scans

and to four popular MDIS, namely the recent PH-tree [Zäschke et al. 2014], and main-memory adapted variants of the R*-tree [Guttman 1984], the kd-tree [Bentley 1975], and the VA-file [Weber et al. 1998]. We use different real and synthetic data sets of varying size with dimensionalities between three and 100. We evaluate complete-match and partial-match range and point queries. As opposed to Chapter 4, we also consider mixed read and write workloads.

The remaining chapter is organized as follows. The next section covers the memory layout and data organization of BB-Trees. Section 5.2 presents the concepts of elastic bubble buckets. In Section 5.3, we describe how to efficiently rebuild the index after many inserts or deletions. Section 5.4 describes the effects of low-cardinality dimensions. In Section 5.5, we show how to execute point and range queries in BB-Trees. Section 5.6 introduces the parallel BB-Tree, which leverages multiple threads to speed-up search queries. In Section 5.7, we compare BB-Trees with other state-of-the-art MDIS using synthetic and realistic query workloads executed over synthetic and real-world data sets. Finally, Section 5.8 critically discusses the advantages and limitations of BB-Trees and Section 5.9 summarizes this chapter.

5.1 Data Organization

BB-Trees consist of two main components: the *inner search tree* (IST) and the set of leaf nodes. The IST is a k -ary search tree, where each node recursively splits a m -dimensional data space according to $k-1$ delimiter values into k disjoint subsets of *almost-equal*¹ size. The granularity of the partitioning increases with the depth of the tree. At the highest granularity, i.e., the lowest tree level, the IST partitions the data space into regions, of which each is completely covered by a leaf node. While the inner nodes are solely used for partitioning, leaf nodes keep the data objects and are implemented as *bubble buckets* (BB). BB can dynamically expand and shrink to cope with varying number of objects in the index region they represent. When searching, the inner nodes are navigated to reduce the data space. Once all certainly irrelevant regions (or BB) have been pruned, the remaining BB are scanned to determine the true query results. Table 5.1 shows notations and input parameters frequently used in the following sections.

BB-Trees are *almost* balanced. While the IST is always perfectly balanced, BB are allowed to introduce slight imbalance to the index structure when expanding to handle a growing amount of objects. However, these imbalances are local and limited to one additional tree level to prevent degeneration of the search tree. Section 5.2 provides further insights into the elasticity properties of BB. The following sections describe in detail how BB-Trees manage multidimensional data in main memory and study the advantages and limitations of their data layout.

Inner Search Tree

In BB-Trees, inner nodes recursively split the data space into k partitions, according to a *delimiter dimension*, chosen from the data space, and $k-1$ *delimiter values*, which belong to

¹Dimensions with few distinct values, namely low-cardinality dimensions, or dimensions with high-frequency values occurring many times, may hinder equal splits when used as delimiter and create slight imbalances in partitioning.

Notation	Description
n	Cardinality of the data set.
m	Dimensionality of the data set.
h	Height of the search tree.
k	Inner nodes split the space into k subparts according to a delimiter dimension and $k - 1$ delimiter values.
t	Number of available hardware threads (relevant for parallel BB-Trees).
B_{match}	Number of bubble buckets that need to be scanned to evaluate a certain range query.
Parameter Description	
b_{max}	Capacity of a regular bubble bucket.
$R_{samples}$	When reorganizing the index, we use $R_{samples}\%$ of all data as samples.

Table 5.1: Frequently used notations and input parameters.

the domain of the delimiter dimension. The height of the IST, h , depends on the fan out of the inner nodes and on the number of BB, but not on the dimensionality of the data space. BB-Trees employ only as many tree levels as needed to distinguish between all BB. Assuming that n data objects are perfectly distributed among n/b_{max} BB, i.e., each BB is completely filled, $h = \log_k(n/b_{max})$.

All nodes of the same tree level use the same delimiter dimension, but can employ individual delimiter values. BB-Trees choose delimiter dimensions in the order of their cardinalities, moving dimensions with a large number of distinct values to the upper tree levels and dimensions with a small number of distinct values to the lower tree levels. If $h < m$, low-cardinality dimensions, which have low pruning power anyway, are kept out of the IST and are omitted for partitioning.

The structure of the IST offers several advantages in the context of main-memory database systems:

- BB-Trees can store the delimiter dimensions and the delimiter values separately, which enables a simple and space-efficient memory layout. As every node belonging to the same tree level uses the same dimension for partitioning, BB-Trees can manage this information in one array of length h , where entry i holds the i -th delimiter dimension. Furthermore, BB-Trees can linearize the inner nodes, which is facilitated by the balancedness of the IST. Technically, BB-Trees store the delimiter values of all inner nodes together in a single, immutable array using a breadth-first order, similar to the linearized fast lane array of CSSL (see Section 3.3), which eliminates pointers, strongly increases cache efficiency and saves memory space.

When implementing the array of delimiter dimensions with one-byte integer values (supporting up to 256 dimensions) and the array of delimiter values with four-byte floating-point values, the space requirements of the IST boil down to $h + \sum_{i=0}^h k^i * 4 * (k - 1)$ bytes.

- Cache lines are the basic unit for transferring data between main memory and CPU caches. By choosing an appropriate value for k , BB-Trees tailor the capacities of their inner nodes to the individual cache line size of the CPU, which increases cache line utilization and thus reduces the amount of data transferred through the cache hierarchy. For point queries or highly-selective workloads, navigation of the IST accesses exactly one cache line at each tree level. When implementing delimiter values with four-byte floating-point values, we set $k = 17$, because $k - 1 = 16$ delimiter values perfectly fit into one 64-byte cache line.
- Searching the delimiter values of an inner node, as necessary during query evaluation, can be efficiently implemented with a binary search. A binary search requires only $\log_2(k - 1)$ instead of $(k - 1)$ comparisons. When searching large arrays, a binary search accesses non-consecutive memory locations with a random access pattern and produces many cache misses. In our case, the binary search operates the delimiter values of inner nodes, which are stored in a single cache line. Thus, using a binary search does not produce more cache misses than searching the delimiter values with a sequential scan, but saves comparisons.

However, the structure of the IST has also certain drawbacks:

- For all nodes of the same tree level, BB-Trees always employ the same dimension as delimiter. This increases space efficiency, but leads to less flexibility with regards to the partitioning. Depending on the skew of and correlation between dimensions of a data set, using different delimiter dimensions at the same tree level may provide a more balanced partitioning.
- Equally splitting a subtree by one dimension is not always possible, which may lead to subtrees with different numbers of contained objects, thus causing slight imbalances. For instance, low-cardinality dimensions prevent a partitioning of objects into k subsets of equal size. Instead, we may consider multiple delimiter dimensions, similar to a quadtree [Finkel et al. 1974].
- Storing all delimiter values in an immutable array hinders updates and requires a complete rebuild when changing data. Frequent updates imply frequent rebuilds and lead to a poor write performance. To overcome this limitation, BB-Trees introduce the concepts of BB, which can, up to a certain point, dynamically expand and shrink to locally deal with data updates without affecting the IST. As demonstrated in our evaluation (see Section 5.7), rebuilding the array of delimiter values is a seldomly-needed operation in BB-Trees.

When considering these limitations, note that they are shared by most other updateable MDIS. For instance, the structure of kd-trees strongly depends on the order of the insertions. In the worst case, when all inserts are always applied to the left or right sub tree, kd-trees degenerate to a linked list and lose all their pruning capabilities. The K-D-B-tree turns kd-trees into balanced search trees, but these improvements come at the cost of complicated and slow update operations, which need to handle node overflows, possibly affecting the entire index structure.

Leaf Nodes

BB-Trees store all data objects in their leaf nodes implemented as bubble buckets. Ignoring the elasticity of BB, every BB has a maximum capacity of b_{max} data objects. At query time, the bucket capacity b_{max} determines the balance between the time spent in tree searching, which results in pruning, and the time spent in evaluating the BB, which results in scanning.

A large BB capacity leads to leaf nodes storing more objects, which in turn requires less inner nodes and thus a less deep tree. Such a structure is preferable for query workloads with predominantly low selectivities: More work is put on scans, where the comparisons between the data objects and the query object lead to many matches, and less time is spent in pruning which, for low selectivities, is not effective anyway. On the other side, a small BB capacity results in smaller leaf nodes but a deeper tree structure, which is beneficial for highly-selective queries as more time is invested in successful pruning and less in scans producing almost no matches.

Delimiter Values

For a good search performance, it is crucial that the inner nodes allow to prune subtrees and non-relevant BB as high in the tree as possible. To this end, BB-Trees choose their delimiter dimensions in the order of the number of distinct values, assuming that dimensions with a high cardinality offer high pruning power.

While all inner nodes of the same tree level use the same delimiter dimension, these nodes can choose individual delimiter values. In the most common case, where the associated delimiter dimension has more than k different values, inner nodes determine the delimiter values such that each subtree contains a roughly equal number of objects. A special case occurs when *low-cardinality dimensions*, which feature less than k different values, are used for partitioning. Such dimensions do not allow to divide the objects into k sub trees of equal size (see Section 5.4).

Note that if the number of tree levels, h , is smaller than the dimensionality of the data set, m , BB-Trees omit the dimensions with the smallest cardinalities in the IST. In practice, this scenario is quite frequent due to the high fan out of BB-Trees, which makes the shape of the tree rather flat even for very large data sets. As an example, assume a BB-Tree over one Billion data objects, a fan out of $k = 17$, a BB capacity of $b_{max} = 1,000$, and an average BB fill degree of 50%. Addressing the resulting two Million BB requires only six IST levels. Thus, low-cardinality dimensions, which are anyway problematic in terms of their minor pruning capabilities, do not clutter the search tree.

In the case that h is larger than m , we assign dimensions multiple times as delimiters in a round-robin fashion. This scenario mainly occurs for large data sets with a low dimensionality. Assuming $k = 17$, $b_{max} = 1,000$, an average BB fill degree of 50%, and a three-dimensional data space, the according BB-Tree can index up to 2,456,500 data objects without employing any dimension multiple times as delimiter.

Example

Figure 5.1 illustrates a BB-Tree with an inner node fan out of $k = 3$, $h = 2$ levels of inner nodes, and nine BB managing three-dimensional data objects (buckets 3 to 6 are not displayed). Each BB can hold up to $b_{max} = 4$ data objects. Individual data objects are identified by unique TIDs.

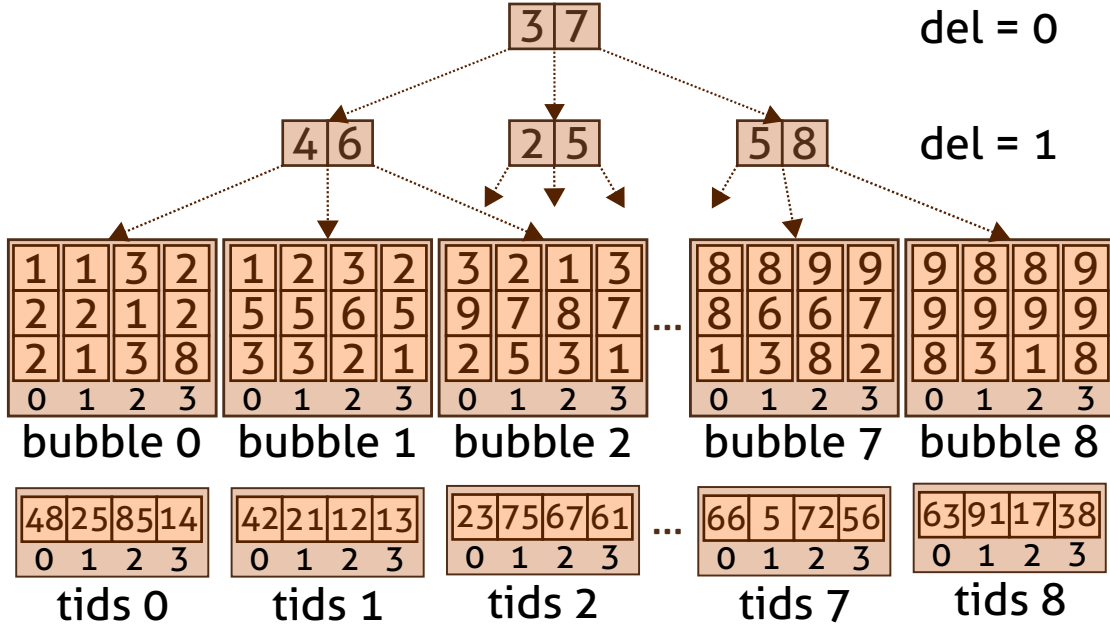


Figure 5.1: A BB-Tree of height $h = 2$ with an inner node fan out of $k = 3$ and a BB capacity of $b_{max} = 4$ managing $n = 36$ data objects of dimensionality $m = 3$; buckets 3 to 6 are omitted.

At the first tree level, the shown BB-Tree splits the data space into $k = 3$ partitions according to delimiter dimension 0. All data objects having a value of 3 or less in dimension 0 are held in the left subtree. All data objects having a value of 7 or less, but larger than 3, in dimension 0 are held in the middle subtree. All other data objects can be found in the right subtree. At the next tree level, the data space is recursively split according to delimiter dimension 1. Note that the shown BB-Tree uses only two out of the three dimensions of the data space as delimiter. Given a fan out of $k = 3$, two tree levels are sufficient for distinguishing between nine BB.

Figure 5.2 illustrates the corresponding array storing the linearized IST. BB-Trees link the linearized IST with the corresponding BB by maintaining an array of pointers, where entry i references the i -th BB.

SIMD Instructions

Although processing inner nodes with *Single Instruction Multiple Data* (SIMD) sounds promising at first glance, especially because the IST is linearized and packed into a dense array, we were not able to obtain any performance benefits through SIMD parallelism. Compared to a binary search, scanning inner nodes with SIMD instructions does not save many comparisons yet incurs overhead. For instance, when employing 16 delimiter values ($k = 17$), which perfectly fit into one 64-byte cache line, a binary search requires $\log_2(16) = 4$ comparisons, whereas a SIMD search using 256-bit registers needs two comparisons (or four comparisons if only 128-bit SIMD registers

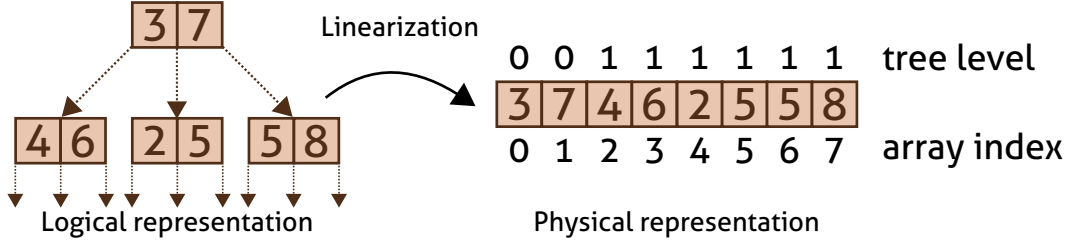


Figure 5.2: The linearized storage of the inner search tree.

are available). These small savings in terms of comparisons are outweighed by the overhead induced by SIMD, especially data transfers between regular and vector registers [Broneske et al. 2017b], and the necessary scalar evaluation of the results of SIMD instructions.

5.2 Bubble Buckets

The previous section described BB-Trees as static index structures and omitted the treatment of overflowing or underflowing leaf nodes. In the following, we lift this restriction and propose two techniques to cope with changing data, namely bubble buckets, described in this section, and index rebuilds, covered by the next section.

BB-Trees implement their leaf nodes as elastic *bubble buckets* (BB). We distinguish between two types of BB, namely *regular BB* and *super BB*. *Regular BB* are implemented with a C++ `std::vector`, which is a dynamically growing and shrinking array, and take inserts up to their maximum capacity of b_{max} data objects. *Super BB* locally increase the depth of the tree by one additional level and contain k regular BB. As usual, super BB employ the dimension of their data region, which features the largest number of distinct values, as delimiter, and choose the $k - 1$ delimiter values such that the data objects are as evenly distributed among the child BB as possible. An overflowing regular BB morphs into a super BB, and an underflowing super BB morphs into a regular BB. In the following, we describe how BB-Trees implement insertions and deletions.

Insertions

In the first step of an insertion, we traverse over the IST to determine the leaf node that is responsible for the data space region that the to-be-inserted object belongs to. Depending on the type and fill degree of the leaf node, insertions are handled differently (see Algorithm 4). If the leaf node is a regular BB and has free space, we insert the data object and are done. If the leaf node is a regular BB but has no free space, we morph the regular BB into a super BB and insert the data object into the corresponding child BB. If the leaf node is a super BB and has free space, we determine the appropriate child BB, which is a regular BB, and insert the object. If the super BB, which has a maximum capacity of $k * b_{max}$ objects, is already completely filled, we must reorganize the index as described in Section 5.3.

Algorithm 4 Inserting an object into a bubble bucket.

data_object: The to-be-inserted data object.
bubble_bucket: The bubble bucket that the new object is inserted into.

```

1: function INSERTOBJECT(data_object, bubble_bucket)
2:   if bubble_bucket.type == 'regular' then
3:     if bubble_bucket.full? then
4:       bubble_bucket.morphIntoSuperBubbleBucket()
5:     end if
6:     bubble_bucket.insert(data_object)
7:   else
8:     bubble_bucket.insert(data_object)
9:     if bubble_bucket.full? then
10:      InvokeRebuild()
11:    end if
12:  end if
13: end function

```

BB-Trees can thus accommodate up to $k * b_{max}$ inserts into the same region, covered by one BB, before the index needs a rebuild. If some of its data objects are removed during insert-heavy workloads, this period gets even longer. Within this time, the IST of the BB-Tree remains stable, and the depth of the search tree is locally increased by one level at most. To keep the algorithms as simple as possible, we do not balance the size of the child nodes of a super BB, which, in theory, could lead to cases where all inserts accumulate in one child node. This would for instance happen when objects of an one-dimensional data set are inserted in a certain sort order.

Deletions

Deletions work similar as insertions, but require less rebalancing, because BB-Trees do not specify minimal fill degrees for regular BB. We first navigate the IST to determine the BB responsible for the to-be-deleted object and remove it from the leaf node.

In the case of a regular BB, no further processing is performed. This implies that some leaf nodes of a BB-Tree may become empty. However, due to the dynamic size of their implementation, the memory overhead is minimal. We nevertheless rebuild the index when more than 10% of all BB are empty to get rid of superfluous inner nodes and reduce the height of the IST.

In contrast to regular BB, super BB can underflow. When removing a data object from a super BB, the delete operator checks the total number of objects stored in the super BB. If the number is smaller than $p * b_{max}$, the super BB morphs into a regular BB. In the default setting, we set $p = 0.5$ to prevent pathological cases of constantly morphing BB when the b_{max} -th object is inserted and deleted iteratively.

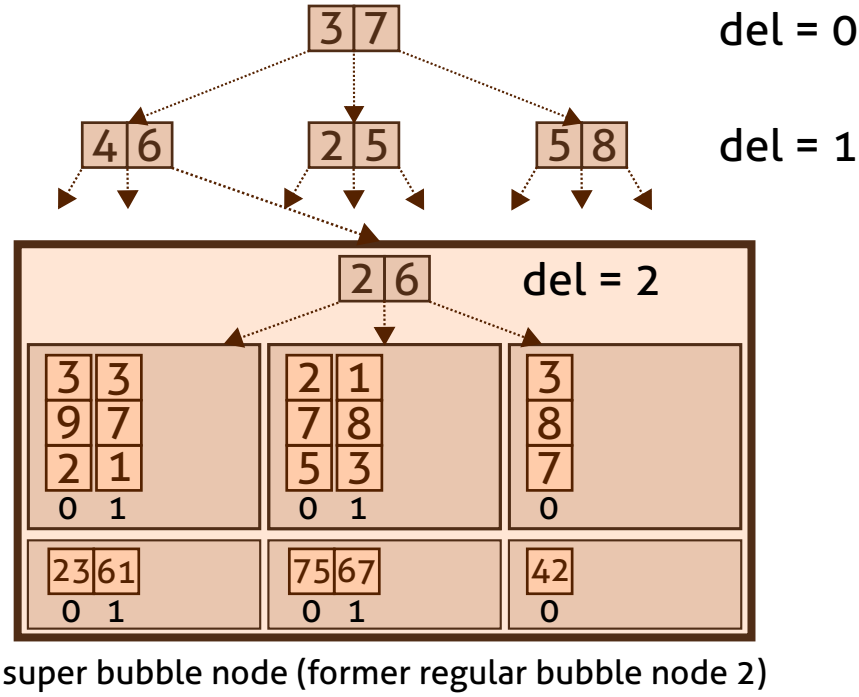


Figure 5.3: When inserting a new data object (3 8 7) with TID 42 into the BB-Tree from Figure 5.1, the regular BB 2 morphs into a super BB that contains k regular nodes and partitions data objects according to dimension 2.

Example

Consider again the example from Figure 5.1. When we insert the data object (3 8 7) into the BB-Tree, bucket 2 overflows and morphs into the super BB shown in Figure 5.3. Here, the super BB chooses dimension 2 as delimiter to partition the data objects into $k = 3$ regular BB according to the delimiter values 2 and 6.

5.3 Building and Reorganizing BB-Trees

Initially, BB-Trees consist of one regular BB and an empty IST, as no inner nodes are needed in the case of a single leaf node. After b_{max} objects have been inserted, this regular BB overflows and morphs into a super BB holding k regular BB, but still leaving the IST empty. With $(k - 1) * b_{max}$ more inserts, this super BB also overflows and triggers a rebuild of the index, creating the first level of inner nodes.

All operations on the BB-Tree, except for the very first, operate on a structure that was the result of an index rebuild. A rebuild of the IST consists of the following four steps:

- In the first step, we determine how many regular BB are needed to manage the current amount of indexed data, while leaving enough capacity for new inserts. From this number,

we also derive the necessary number of levels of inner nodes. By default, when rebuilding, we set the number of BB to $n/(10\% * b_{max})$ allowing each leaf node to ingest further $90\% * b_{max}$ data objects until morphing into a super BB. This parameter may be changed depending on the expected workload. For insert-heavy workloads, we recommend using a low value that leads to seldom rebuilds enabling a high write performance, whereas read-heavy workloads can benefit from a high value leading to frequent rebuilds and ensuring that the IST always reflects the current data distribution.

- We randomly sample $R_{samples} * n$ data objects to obtain representatives of the whole data set. By scanning the sampled data, we estimate the number of distinct values of each dimension. The dimensions are sorted by their cardinality and assigned to the h levels of the new IST in descending order. If h is larger than the dimensionality of the data space, we assign dimensions multiple times in a round-robin fashion. For instance, when indexing a two-dimensional data set, where the second dimension contains more distinct values than the first one, with a BB-Tree of height five, we assign the dimensions to the IST levels as follows: (1 0 1 0 1).
- We determine the delimiter values for the inner nodes, starting at the root node and recursively working down to the lower tree levels. To this end, using the sample data, we compute an equi-depth frequency histogram covering the values of the delimiter dimension of the current level. We choose delimiter values such that each interval covers an almost-equal number of objects. We use an efficient greedy assignment algorithm, processing the values from left to right, and always find the next delimiter value such that approximately $1/k$ of the objects are covered by the current interval.

Note that, by using an equi-depth histogram, we can find partitions of almost-equal size even in the case of dimensions, where some values occur with a higher frequency than others. However, if the differences in the frequencies of the values are very large, we inevitably end up with intervals of different size. Clearly, this procedure also fails for low-cardinality dimensions containing less than k distinct values, as described in Section 5.4.

- In the last step of the rebuild, according to the derived IST, all data objects are inserted into their new respective BB.

A periodic reorganization of the IST is mandatory to support updates, because BB-Trees store the entire IST in an immutable array. However, index reorganization obviously is an expensive operation. A random sample must be determined which is scanned multiple times, a new IST is constructed, and data objects must be moved to their new location. We chose pragmatic and fast methods for these steps, which come at certain drawbacks. First, splitting a subtree by one dimension into intervals of equal size is not always possible, as in the case of low-cardinality dimensions (see Section 5.4). Second, we globally assign dimensions to tree levels, which again can lead to imbalances when dimensions are strongly correlated. Third, we compute the IST structure only on a sample. If the sample is small, the tree is found quickly yet might not optimally represent the data. Contrary, if the sample is large, building the tree needs more time yet probably leads to a better tree structure.

We make two notes regarding these issues. First, they are shared by most other updateable MDIS. For instance, the structure of kd-trees strongly depends on the order of the insertions. The K-D-B-tree turns kd-trees into balanced search trees, but at the price of complicated and slow update operations. Second, though we cannot give formal guarantees, for the data sets we used in our evaluation, we never observed any notable imbalance. We are thus confident that unbalanced BB-Trees with regions largely differing in terms of covered objects, which are possible in theory, remain very rare in practice.

5.4 Low-Cardinality Dimensions

We consider a dimension of the data space to have a low cardinality, when its number of distinct values is smaller than the fan out of the IST, k . Low-cardinality dimensions are common in real-world data sets and challenge MDIS, because they make fine-grained partitions impossible and hurt pruning power. The problem is less severe for the BB-Tree, because it sorts the delimiter dimensions by their number of distinct values, which keeps low-cardinality dimensions completely out of the IST, assuming that $h < m$ and that the data set contains less than $m - h$ low-cardinality dimensions.

However, if a data set contains less than h dimensions having at least k distinct values, the resulting BB-Tree would use at least one low-cardinality dimension as delimiter. In such cases it is impossible to find distinct delimiter values, which split the data space into k subparts of equal size.

Low-cardinality dimensions are less severe for the performance of insertions, which, when navigating inner nodes with duplicate delimiter values, can randomly choose one of the child nodes to proceed with. However, low-cardinality dimensions may hurt the performance of operations relying on the pruning capabilities of the IST. Range queries may need to consider more child trees than when navigating inner nodes using high-cardinality dimensions for partitioning. Even point queries and deletions may need to follow multiple parallel paths instead of having to consider only one child tree.

5.5 Search Algorithms

BB-Trees support point queries and partial- and complete-match range queries. All search queries have in common that they exploit the linearized IST to find those candidate BB that may hold data objects matching the search query while pruning all others. The navigation of the IST is followed by sequential scans over all candidate BB to determine the true results of the query. Query evaluation may have to follow multiple parallel paths through the tree: Partial-match range queries must consider k paths when an inner node splits on a dimension which is not part of the query and have to follow multiple paths when the queried range intersects with more than one subtree. Complete-match range queries have to consider multiple paths when the queried range covers more than one subtree. Even point queries may need to consider multiple paths when a low-cardinality dimension is used as delimiter.

Point Queries

Point queries (or lookups) are used to verify that a certain data object exists. If the object has been found, BB-Trees return its TID. Otherwise, they return the value -1 to indicate that the search was not successful.

When inner nodes do not feature any duplicate delimiter values, applying point queries to the IST has a complexity of $O(h * \log(k))$, because they perform h times a binary search within inner nodes holding $k - 1$ values. In such a case, navigation of the IST determines exactly one candidate BB. When the BB-Tree uses low-cardinality dimensions for partitioning, up to k search paths have to be followed per inner node, leading to a complexity of $O(k^h)$ for the navigation of the IST. In such a case, multiple (or up to all) candidate BB have to be scanned. Scanning one BB has a complexity of $O(b_{max} * m)$, because up to b_{max} objects have to be considered, each requiring up to m comparisons.

In the best case, the navigation of the IST determines only one candidate BB and the searched object is stored at the first position of the candidate BB, leading to a complexity of $O(h * \log(k) + m)$, because the query evaluation terminates once the searched object has been found. In the average case, the navigation of the IST determines one candidate BB and the search algorithm has to scan half of the entries of the candidate BB, leading to a complexity of $O(h * \log(k) + b_{max} * m)$. Even searching for a non-existing object has a complexity of $O(h * \log(k) + b_{max} * m)$, when only one candidate BB has to be considered, as non-existence can be confirmed once the complete BB is scanned. In the worst case, when searching for a non-existing data object and when the navigation of the IST does not prune any BB, the evaluation of a point query has a complexity of $O(n)$, because every candidate BB has to be completely investigated.

Range Queries

Multidimensional range queries (MDRQ) specify predicates for some, many or all dimensions of a multidimensional data space. MDRQ can be either complete-match or partial-match: Complete-match MDRQ specify predicates for all dimensions of the data space, whereas partial-match MDRQ specify predicates for a subset of the dimensions. Compared to point queries, range queries may involve much more data objects. The range query operator of BB-Trees returns the TIDs of the objects satisfying the range boundaries of the search object (or an empty set if no object matches the query).

When the evaluation of a range query on the IST can prune all but one BB, it has a complexity of $O(h * \log(k))$. On the other hand, when applying a range query to the IST cannot prune any BB, but must follow all search paths, it has a complexity of $O(k^h)$.

In the best case, when range queries have to scan only one candidate BB, their overall complexity boils down to $O(h * \log(k) + b_{max} * m)$. The best case can only occur, when at most b_{max} of all data objects match a query, which requires a highly-selective query object, assuming that n is much larger than b_{max} . The average case complexity strongly depends on the expected average query selectivity. Assuming that the navigation of the IST must follow half of all search paths and returns B_{match} candidate BB, its complexity is $O(k^h + B_{match} * b_{max} * m)$. In the worst case, all candidate BB must be considered, which leads to the same worst case complexity as for point queries: $O(n)$.

Compared to point queries, range queries are typically less selective, which increases the probability of multiple candidate BB. On the other hand, low selectivities make scans more attractive as more of their comparisons actually lead to matches, without requiring any tree navigation in between.

Determining an optimal BB capacity would only be possible if all queries had the same, a-priori known selectivity across the entire data space, an assumption that is rather impractical. In practice, every setting of b_{max} implements an expectation on the average selectivities of queries in the future workload. In our evaluation, we show that our default value leads to a performance that is almost on-par with MDIS specialized in point queries, while clearly outperforming all competitors for range queries.

5.6 Parallel BB-Trees

As described in Section 2.3.3, hardware manufacturers install an ever growing number of cores on one CPU to accelerate performance through concurrent execution of instructions. Index structures should provide multithreaded search operators to fully leverage the parallel capabilities of modern processors.

To this end, we propose the *parallel BB-Tree* as a parallel implementation of regular BB-Trees. It uses multiple threads to execute search queries, yet relies on the same algorithms as the single-threaded BB-Tree for insertions, updates and deletions. Recall that BB-Trees execute search queries in two phases: In the first step, they navigate the IST to obtain candidate leaf nodes, which are scanned in the second step. Obviously, both steps cannot be executed concurrently, as the scan depends on the results of the traversal over the IST. Instead, each phase must be parallelized individually.

Navigation of the IST with multiple threads is very complicated, because it would need a barrier synchronization after each tree level [Buluç et al. 2011; Yoo et al. 2005]. At the same time, there is only little performance to gain as the runtime of a search query is strongly dominated by the scan of the candidate BB, as shown in Section 5.7.5.

In contrast, the second phase, which scans all candidate BB to find the search results, can be easily parallelized. Let B_{match} denote the number of candidate BB that have been determined by traversing over the IST (super BB are considered as multiple BB), and let t equal the number of threads available on the machine. If $B_{match} \geq t$, we divide the set of candidate BB into B_{match}/t partitions², where each partition contains one or multiple BB. Each partition is processed by a distinct thread using the same algorithm as in the regular, single-threaded BB-Tree. If $B_{match} = t$, we obtain the perfect degree of parallelism. If $B_{match} < t$, we assign multiple threads to single candidate BB to avoid wasting computing resources. To this end, we split the data objects of a single candidate BB into multiple chunks, and scan each chunk with a distinct thread. The search results of individual threads are concatenated once they are finished.

²If B_{match} cannot be divided by t , one thread has to do slightly less work than the others, creating a small imbalance. For instance, given that the navigation of the IST determined five candidate BB and three threads are available, two threads would have to scan two candidate BB while one thread would have to process only one candidate BB. However, the effects of such imbalances are rarely, if at all, observable in practice.

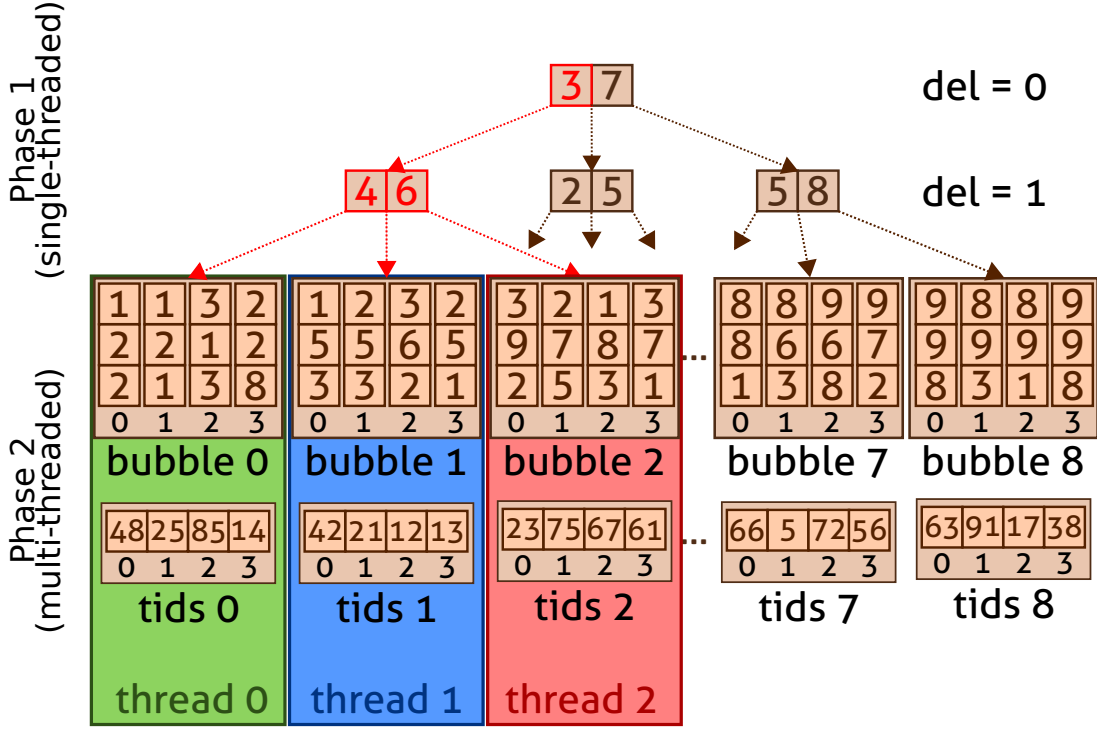


Figure 5.4: Parallel execution of an exemplary range query, defined by the lower boundary $[1, 0, 3]$ and the upper boundary $[3, 7, 6]$, using three threads.

Figure 5.4 illustrates the execution of range queries in a parallel BB-Tree. The search query retrieves all data objects that satisfy the lower boundary $[1, 0, 3]$ and the upper boundary $[3, 7, 6]$. In this example, the first, single-threaded phase of the query execution determines that BB 0, 1 and 2 may hold data objects matching the range boundaries (the search path is marked in red). In the second, multithreaded phase, these candidate BB are searched concurrently with one CPU thread per bucket assuming that our imaginary machine features three threads ($B_{match} = t$).

Our parallelization approach yields very good scalability. We can easily split the scan over the candidate BB into sub tasks of almost-equal complexity, which are processed by distinct threads, enabling a perfect utilization of the available computing resources. Moreover, stragglers are avoided, because every thread scans an almost-identical amount of data. At the downside, we do not parallelize the entire evaluation of search queries, but still traverse over the IST with a single thread. However, navigation of the IST requires much less time than the scan over the candidate BB and therefore has negligible effects on scalability.

5.7 Evaluation

We compare BB-Trees with state-of-the-art approaches to general-purpose indexing of multidimensional data by executing synthetic and real-world query workloads over synthetic and real-world data sets. Specifically, we aim to answer the following questions:

- Does the performance of BB-Trees depend on data- or workload-specific characteristics, e. g., data dimensionality, data skew, or query selectivity (see Sections 5.7.4 to 5.7.6)?
- How do BB-Trees perform on mixed workloads that contain both read and write operations (see Section 5.7.9)?
- What is the effect of parallelization (see Section 5.7.10)?
- How efficient do BB-Trees utilize memory space (see Section 5.7.11)?

5.7.1 Experimental Setup

Hardware

We executed all experiments on a server equipped with two Intel Xeon E5-2620 CPUs (2 GHz clock rate, 64-byte cache lines, six cores, 12 hardware threads) and 32GB of RAM. In total, the machine features 12 cores and 24 hardware threads. Most experiments are single-threaded; Section 5.7.10 investigates parallel BB-Trees and therefore makes use of multiple threads.

Methodology

In our evaluation, data for all competitors are completely kept in main memory. All MDIS are built using single-tuple inserts, with data sets inserted in random order; only VA-files require a bulk insert. All experiments measure the average execution time of an operation, e. g., range query. We run experiments three times and present the arithmetic mean.

Competitors

We compare the BB-Tree with multiple approaches to general-purpose multidimensional indexing: the kd-tree [Bentley 1975], the PH-tree [Zäschke et al. 2014], the R*-tree [Beckmann et al. 1990], the VA-file [Weber et al. 1998] and the sequential scan [Sprenger et al. 2018b]. Section 2.2 provides a detailed description of the competitors.

For the R*-tree, we used an open-source implementation³ and mostly relied on the default configuration; we only adjusted the node capacities such that the nodes are aligned to cache lines (see Section 4.3). For the PH-tree, we used the publicly available C++ implementation shared by the authors⁴. For the kd-tree, the VA-file and the sequential scan, we used our own implementations based on the original publications (see Chapter 4 for details).

³libspatialindex - libspatialindex 1.8.0 documentation, <https://libspatialindex.github.io/>, Last access: August 29, 2018.

⁴tzaeschke/phtree-1, <https://github.com/tzaeschke/phtree-1>, Last access: August 29, 2018.

Most contestants use four-byte floating-point values to manage dimension data, including the BB-Tree. The R*-tree implementation uses eight-byte floating-point values; the PH-tree implementation uses eight-byte integer values. The usage of larger data types for the implementation of data objects affects both search and space efficiency: (1) Less dimension values fit into one cache line, requiring more cache lines to be accessed increasing the probability for cache misses, and (2) more memory space is occupied by the index structure. Section 5.7.11 studies the memory utilization of the competitors; for fairness, we provide results assuming that all approaches were using four-byte values to implement data objects.

We evaluated BB-Trees with an inner node fan out of $k = 17$, because $k - 1 = 16$ four-byte floating-point values fit into one cache line of the evaluation machine. Based on the observations described in Section 5.7.3, we empirically set the BB capacity to $b_{max} = 2,500$. When reorganizing the index, we use $R_{samples} = 10\%$ of all data as samples to estimate the current data distribution.

Software

All software was implemented in C++ and was compiled with GCC using optimization flag `-O3`. We measured hardware performance counters with PAPI⁵ and space consumption with valgrind⁶. For the parallel BB-Tree, we used an open-source thread pool library⁷ to enable reuse of POSIX threads.

5.7.2 Experimental Data and Workloads

We evaluate the competitors on four different data sets, which are almost identical to the ones used in Chapter 4; for CLUSTERED, we here use a larger data set size. Table 5.2 provides the number of data objects (n), the dimensionality (m), the number of distinct values per dimension (for UNIFORM and CLUSTERED, we provide average values for the complete data space), and the raw size of each data set.

We primarily evaluate synthetic workloads. Unless noted otherwise, we generate synthetic range queries by randomly choosing two objects from the data set and, for each dimension, we use the minimum value of both objects as lower boundary and the maximum value as upper boundary. For GENOMIC, we also consider the realistic range query workload described in Section 2.4, consisting of several MDRQ templates of varying selectivity.

Data Set UNIFORM

Synthetic data facilitates experiments with arbitrary data set sizes, dimensionalities and query selectivities. We generate uniformly distributed objects within the domain $[0, 1]$.

⁵PAPI, <http://icl.cs.utk.edu/papi/>, Last access: August 29, 2018.

⁶Valgrind Home, <http://valgrind.org/>, Last access: August 29, 2018.

⁷vit-vit/CTPL: Modern and efficient C++ Thread Pool Library, <https://github.com/vit-vit/CTPL>, Last access: August 29, 2018.

Data Set	n	m	Distinct Values per Dimension	Raw Size
UNIFORM	10k	5	10k (average)	0.2MB
	100k	5	95k (average)	1.9MB
	1M	5	632k (average)	19.1MB
	10M	5 to 100	1M (average)	190.7MB to 3,814.7MB
CLUSTERED	10k	5	10k (average)	0.2MB
	100k	5	95k (average)	1.9MB
	1M	5	632k (average)	19.1MB
	10M	5	1M (average)	190.7MB
POWER	10k	3	10k; 1k; 1k	0.1MB
	100k	3	100k; 2k; 2k	1.1MB
	1M	3	1M; 4k; 5k	11.4MB
	10M	3	10M; 6k; 8k	114.4MB
GENOMIC	10k to 10M	19	see 2.4.2	0.7MB to 724.8MB

Table 5.2: Data sets used in our experiments.

Data Set CLUSTERED

We evaluate the performance of BB-Trees when confronted with non-uniform data distributions using the five-dimensional data set CLUSTERED. Using a generator provided by Müller et al. [Müller et al. 2009], CLUSTERED was created within the same domain as UNIFORM, i. e., $[0, 1]$, but features up to twenty clusters. Within each cluster, data are uniformly distributed.

Data Set POWER

The real-world data set POWER is obtained from the DEBS 2012 challenge⁸ and resembles sensor data of hi-tech manufacturing equipment. As in previous studies using this data set for evaluations [Wang et al. 2016], we index three dimensions.

Data Set GENOMIC

The data set GENOMIC consists of real-world genomic variant data obtained from the 1000 Genomes Project [The 1000 Genomes Project Consortium 2015]. Section 2.4.2 provides a detailed description. Raw variation data are transformed into 19-dimensional data objects. Attributes originally stored as strings, like the population of a sample, are transformed into floating-point values by hashing. In our experiments, we apply the eight different MDRQ templates as well as the mixed workload from GMRQB to the data set GENOMIC. The templates are in-

⁸DEBS 2012 Grand Challenge: Manufacturing equipment - DEBS.org, <http://debs.org/?p=38>, Last access: August 29, 2018.

stantiated with concrete values obtained from the 1000 Genomes Project data. The average query selectivities of the instantiated templates range from 10.76% (low) to 0.00001% (high).

5.7.3 Impact of Bubble Bucket Capacities

The capacity of BB, as defined by b_{max} , controls the ratio between index probing (navigation of IST) and scanning (evaluation of leaf nodes) when searching in BB-Trees. While small BB put more work on index probing, large BB increase the time spent on scanning. As shown in previous work [Weber et al. 1998; Sprenger et al. 2018b], index probing is beneficial for highly-selective queries and scanning is superior for less selective workloads. We study the impact of different configurations of b_{max} on the performance of range queries with varying selectivities (1%, 10%, and 20%) when applied to ten Million five-dimensional data objects from UNIFORM and CLUSTERED. Our goal is to find a pragmatic configuration for BB capacities providing a robust performance for a wide range of query selectivities and data distributions. Figure 5.5 shows the results.

For uniformly-distributed data, this experiment confirms that small (large) capacities are beneficial for highly (less) selective queries. While small BB capacities are more efficient than large BB capacities for queries with an average selectivity of 1%, they become less efficient with increasing query selectivity (10% and 20%). For the selectivities considered here, BB with a maximum capacity of 2,500 objects provide the best performance. Clustered data lead to a less optimal partitioning, which lessens the pruning power of the IST and puts more work on scanning. As a result, small BB capacities become less efficient, even for small selectivities, because less BB can be pruned. Also for clustered data, BB with a maximum capacity of 2,500 objects provide either the best performance or are on a par with other configurations. Taking the results of this experiment into account, we set $b_{max} = 2,500$ for all following experiments.

5.7.4 Point Queries

Figure 5.6 shows the average execution time of point queries for the four data sets depending on the number of data objects (n). We provide the average execution time of n point queries retrieving randomly-chosen, existing objects. Note that, when indexing 10^4 data objects, BB-Trees do not employ inner nodes, but manage all objects within one super BB consisting of $k = 17$ regular BB each holding up to $b_{max} = 2,500$ objects.

In general, the point query performance of the BB-Tree is very similar to that of the other tree-based PAM considered here, the kd-tree and the PH-tree. For UNIFORM, the BB-Tree requires $8.76\mu s$ on average per point query, the kd-tree requires $10.91\mu s$, and the PH-tree needs $4.8\mu s$; for CLUSTERED, the BB-Tree requires $10.27\mu s$ on average per point query, the kd-tree needs $10.46\mu s$, and the PH-tree requires $4.81\mu s$; for POWER, the BB-Tree takes $9.81\mu s$ on average per point query, the kd-tree requires $8.7\mu s$, and the PH-tree needs $2.33\mu s$; for GENOMIC, the BB-Tree requires $29.34\mu s$ on average per point query, the kd-tree needs $18.70\mu s$, and the PH-tree requires $13.42\mu s$ (note that the space requirements of the PH-tree exceeded the available 32GB of main memory for more than 10^5 tuples from GENOMIC).

BB-Trees clearly outperform R*-trees, VA-files and sequential scans for all data sets, often by multiple orders of magnitude. For the largest instance of GENOMIC, consisting of 10^7 objects,

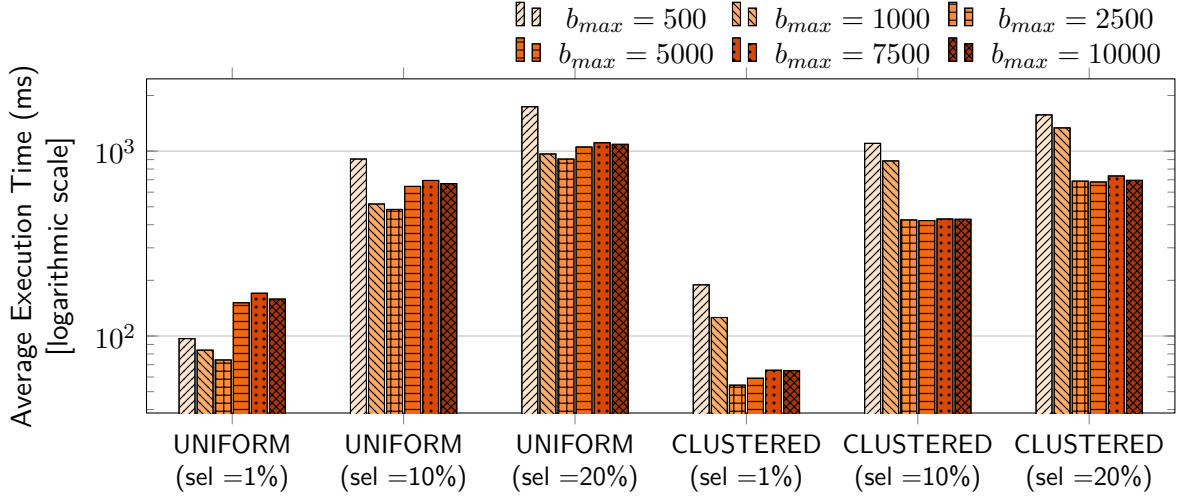


Figure 5.5: Performance of BB-Trees with different BB capacities (B_{max}) when executing range queries with varying selectivities (1%, 10%, and 20%) on ten Million data objects from UNIFORM and CLUSTERED.

the lowest level of the IST contains duplicate delimiter values, caused by highly-correlated dimensions, which require lookup operations to scan multiple candidate BB, resulting in minor performance drops. This experiment shows that the search (and space) efficiency of the PH-tree decreases with an increasing dimensionality, because the fan out of its search tree depends on the dimensionality of the data space. We also investigated searching for non-existing points (data not shown). While the R*-tree, the kd-tree and the PH-tree can apply pruning to efficiently handle such workloads, the performance of the BB-Tree, the VA-file and the scan is worse than when searching for existing points. In particular, BB-Trees show a performance decrease of 43.29% on average, because they need to scan the entire candidate BB to verify the nonexistence of a certain point.

5.7.5 Range Queries

Figure 5.7 shows the average execution time of synthetic range queries that were generated by randomly choosing two objects from the data set and using these as range boundaries. Depending on the distribution and size of the data set, the obtained ranges have a different average selectivity; UNIFORM: 0.4% ($\sigma = 0.9\%$), CLUSTERED: 19.8% ($\sigma = 19.7\%$), POWER: 12.6% ($\sigma = 13.1\%$), GENOMIC: 0.2% ($\sigma = 0.2\%$). For CLUSTERED, one range query may involve multiple clusters, therefore average selectivities are higher than for UNIFORM, although both data sets have an identical size and both are generated within the domain $[0, 1]$.

The BB-Tree achieves the best overall performance and outperforms all other contestants. For the uniform distribution, the R*-tree shows a range query performance similar to that of the BB-Tree. For GENOMIC, kd-trees are almost as fast as BB-Trees. We omit the PH-tree for all MDRQ experiments on GENOMIC, because the range query operator of the implementation

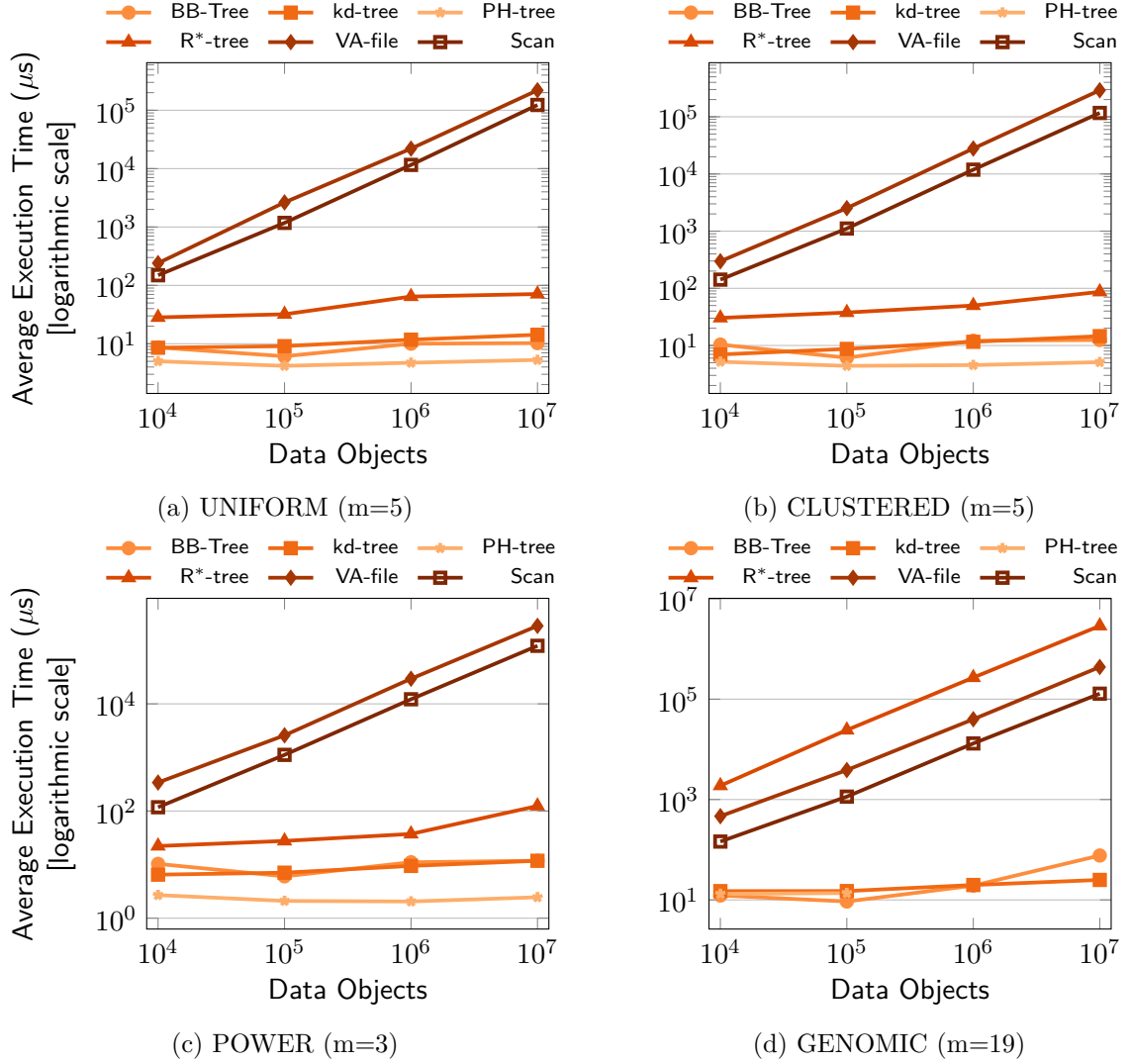


Figure 5.6: Performance of point queries on the different data sets depending on the number of data objects.

given by the authors crashed with a runtime error.

Figure 5.8 presents the average execution time of the eight MDRQ templates and the mixed workload from GMRQB when applied to ten Million data objects from GENOMIC. Also for this realistic range query workload, BB-Trees achieve the best MDRQ performance. Only for Query Template 8, which resembles a point query as it selects a single data object on average, kd-trees beat BB-Trees.

Figure 5.9 presents the performance of the competitors when applying MDRQ of varying selectivity to ten Million data objects from UNIFORM. We omit the kd-tree because its range query performance decreases severely for less selective queries, which would make the figure

5 BB-Trees: Processing Multidimensional Range Queries in Main Memory

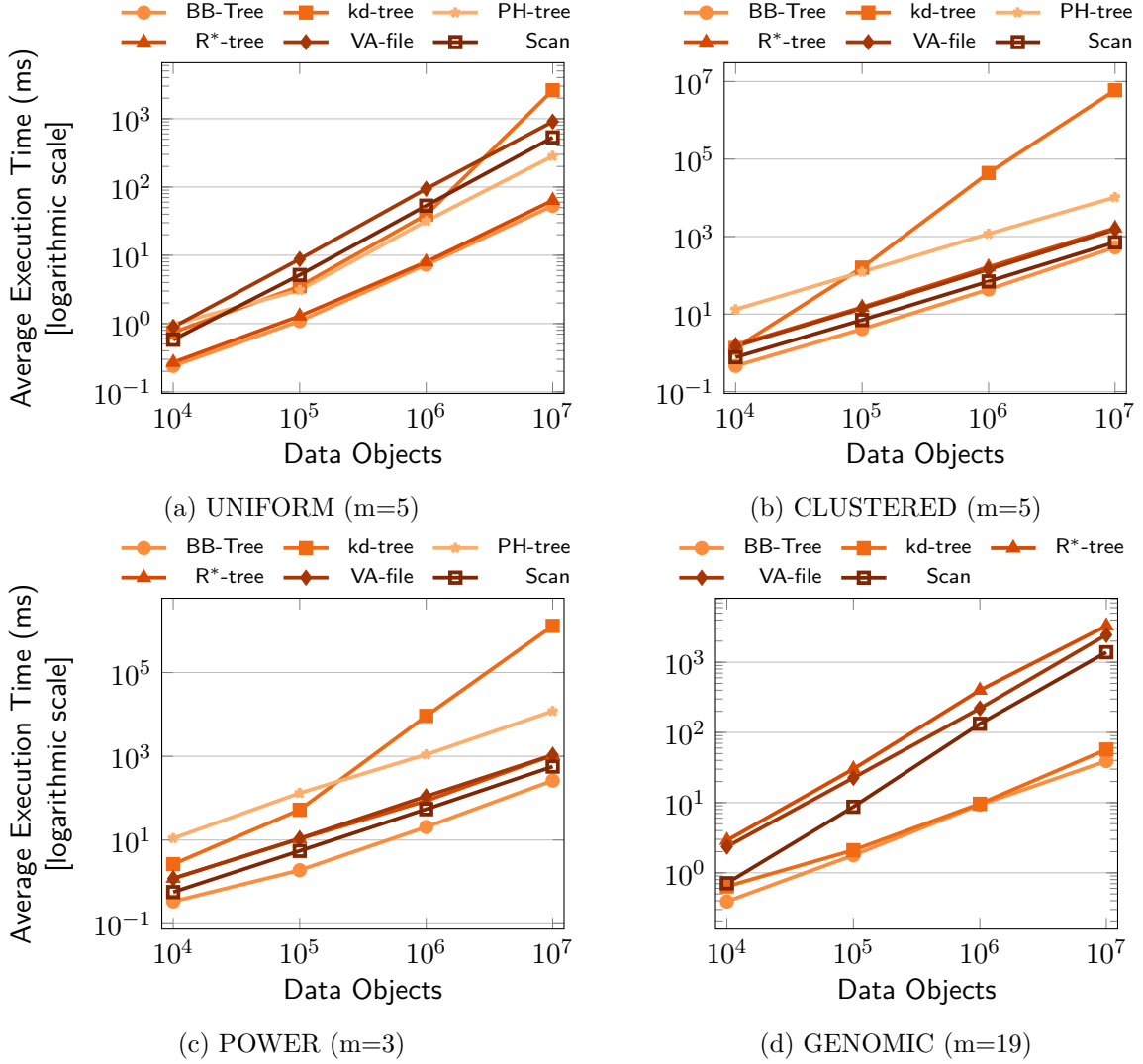


Figure 5.7: Performance of synthetic range queries on the different data sets depending on the number of data objects. Average query selectivities are as follows: UNIFORM: 0.4% ($\sigma = 0.9\%$), CLUSTERED: 19.8% ($\sigma = 19.7\%$), POWER: 12.6% ($\sigma = 13.1\%$), GENOMIC: 0.2% ($\sigma = 0.2\%$).

very hard to read, despite the logarithmic scale of the y axis. For instance, in this experiment, the kd-tree requires 3,075s on average to execute a range query with a selectivity of 20%, while BB-Tree needs only 879ms.

The BB-Tree outperforms the kd-tree, the PH-tree, the R*-tree and the VA-file regardless of the query selectivity. It also beats the sequential scan for queries with a selectivity of up to 20%. For less selective queries ($\geq 30\%$), the performance of BB-Trees remains close to that of

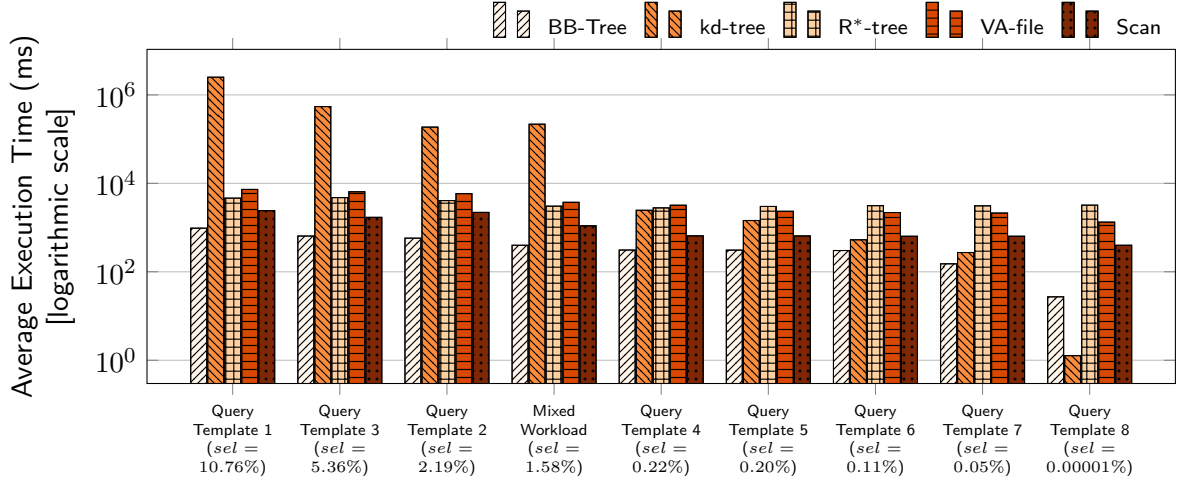


Figure 5.8: Performance of the Genomic Multidimensional Range Query Benchmark when executed on ten Million 19-dimensional data objects from GENOMIC. Query templates are ordered by selectivity, from low (left) to high (right).

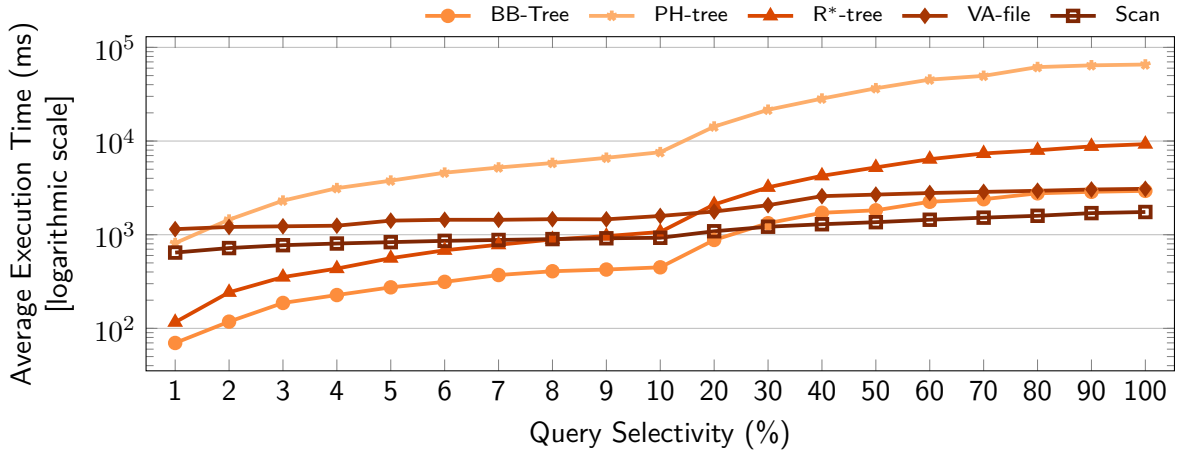


Figure 5.9: Performance of range queries on ten Million data objects from UNIFORM (five dimensions) depending on query selectivity. We omitted the kd-tree.

scans.

For different query selectivities, Table 5.3 presents the time BB-Trees spent on the navigation of the IST and on the scanning of the BB when applying range queries to ten Million five-dimensional data objects from UNIFORM. The results confirm that the second step of the range query algorithm, the scanning of the BB, strongly dominates the first step, the navigation of the IST, in terms of runtime.

We also compare the competitors in terms of their cache efficiency. We study important cache-related performance metrics, such as number of occurred last level cache (LLC) accesses,

5 BB-Trees: Processing Multidimensional Range Queries in Main Memory

Query Selectivity	Navigation of IST	Scanning of BB	Ratio IST/BB
1%	10ms	100ms	1:10
5%	20ms	320ms	1:16
10%	40ms	530ms	1:13
20%	70ms	1,030ms	1:15
50%	120ms	2,030ms	1:17
100%	160ms	2,990ms	1:19

Table 5.3: Time spent on navigation of the IST and scanning of the BB when executing range queries of varying selectivity on ten Million five-dimensional objects from UNIFORM.

	BB-Tree	kd-tree	PH-tree	R*-tree	VA-file	Scan
CPU Cycles	164M	8,306M	1,908M	252M	2,934M	1,582M
LLC Accesses	1.0M	824M	1.2M	2.5M	1.8M	0.5M
LLC Misses	0.7M	0.9M	0.8M	0.5M	1.6M	0.3M
TLB Misses	0.3M	1.0M	0.3M	0.3M	0.2M	0.1M
Branch Mispredictions	0.1M	0.7M	3M	0.2M	10M	7M

Table 5.4: Performance counters per range query with a selectivity of 1% when applied to ten Million objects from UNIFORM (five dimensions).

number of produced LLC misses, and number of generated translation lookaside buffer (TLB) misses, but also investigate the number of spent CPU cycles and number of occurred branch mispredictions. Table 5.4 presents the performance counters for the competitors when executing range queries with an average selectivity of 1% on ten Million five-dimensional data objects from UNIFORM.

Enabled by their sequential access pattern, scans produce only few LLC accesses, LLC misses and TLB misses, thus showing high cache efficiency. BB-Trees, which sequentially access their BB but navigate their IST with random accesses, produce twice as much LLC and TLB misses as scans. However, as opposed to most other competitors, BB-Trees follow most predicted branches, leading to few instruction pipeline flushes and benefiting the number of executed CPU cycles.

5.7.6 Impact of Dimensionality

Most MDIS, especially spatial access methods, tend to become less efficient with an increasing dimensionality of the data space (see Section 4.4.4) [C. Böhm et al. 2001]. We measure the performance of point and range queries executed on ten Million data objects from UNIFORM depending on dimensionality. For this experiment, we generate range queries with an average selectivity of 1% ($\sigma = 0.7\%$). With a growing dimensionality, this results in very low single-dimension selectivities posing serious challenges to MDIS because pruning becomes less useful. For instance, for MDRQ with an overall selectivity of 1% on 100-dimensional uniformly

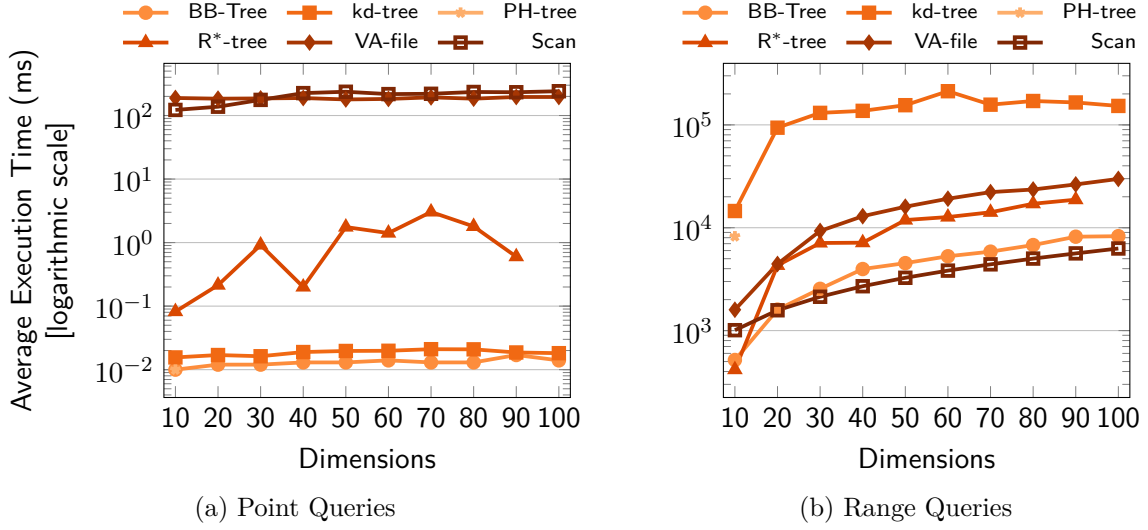


Figure 5.10: Performance of point and range queries (average selectivity = 1%, $\sigma = 0.7\%$) when applied to ten Million uniformly distributed data objects depending on dimensionality.

distributed data, single-dimension selectivities are approximately 95.50%. Figure 5.10 shows runtimes for dimensionalities between ten and 100. Note that the space requirements of the PH-tree exceeded the available 32GB of main memory for dimensionalities higher than ten. Similarly, the R*-tree ran out of space for 100 dimensions.

When evaluating point queries, all methods except the R*-tree are mostly unaffected by the dimensionality of the data. For range queries, all methods behave similar and show a search performance degradation roughly proportional to the dimensionality of the data space. This experiment shows that BB-Trees are rather robust towards varying dimensionalities. In all considered cases, BB-Trees are either the best approach or achieve a performance very close to the fastest competitor.

5.7.7 Low-Cardinality Dimensions

Low-cardinality dimensions are challenging for BB-Trees because they make it impossible to find k different delimiter values, which limits the pruning power of the IST. We first study this effect using range queries applied to ten Million five-dimensional data objects from UNIFORM with different moderately low cardinalities for all dimensions. Results are shown in Figure 5.11. At these cardinalities, none of the competitors is affected severely as the differences only correspond to the different query selectivities. Note that in the cases of eight and 16 distinct values per dimension, the data space includes duplicate data objects which are not supported by the PH-tree; therefore, we omit this method in this experiment.

We also performed an experiment with extremely low cardinalities (between two and 12) yet used data of higher dimensionality. Figure 5.12 shows the performance of range queries when applied to ten Million 50-dimensional objects from UNIFORM. The PH-tree had to be

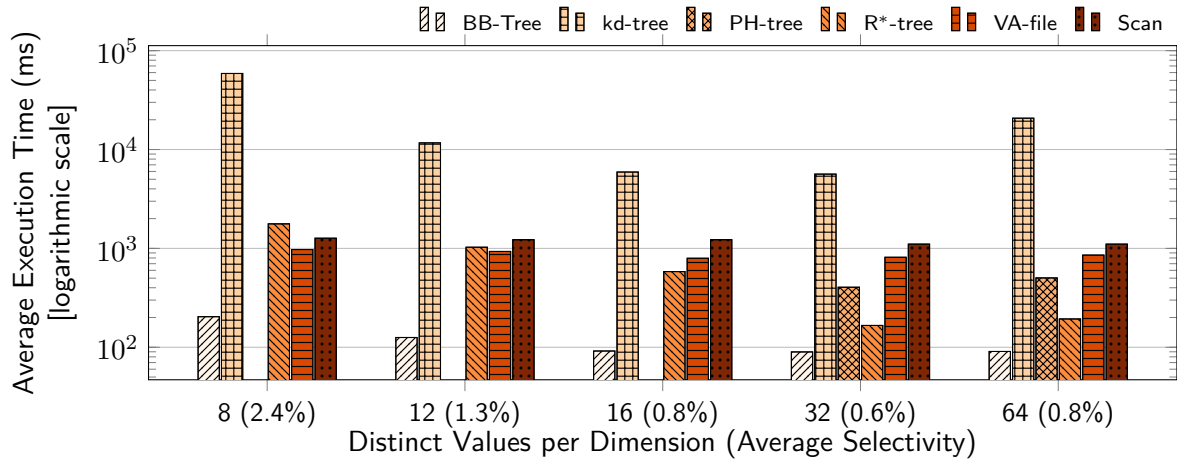


Figure 5.11: Performance of synthetic range queries with a varying selectivity executed on ten Million five-dimensional data objects from UNIFORM depending on the number of distinct values per dimension.

omitted because it produced incorrect results. Induced by the high dimensionality, queries had an average selectivity of 0.00002% ($\sigma = 0.0\%$). This experiment shows that the pruning power of all MDIS drops considerably for lower cardinalities, whereas scans and VA-files are much less effected. Though, for such low cardinalities other approaches, like bitmaps [Chan et al. 1998], are probably a better choice anyway.

5.7.8 Insertions and Deletions

Figure 5.13 presents the average time a contestant needs to ingest an object into its data. The measured times include the reorganizations of the BB-Tree⁹. We do not insert entire data sets at once (bulk inserts), but load data object by data object. Therefore, this experiment does not include the VA-file, which supports only bulk inserts, because it requires to know the data distribution at initialization time. For instances of GENOMIC containing more than one Million data objects, the space requirements of the PH-tree exceeded the available 32GB of main memory. The sequential scan, which implements inserts by appending new objects to a dynamic array, achieves the highest insert performance, because it does not need to deal with node overflows, like the R*-tree, or index reorganizations, like the BB-Tree. BB-Trees show the second best insert performance outperforming kd-trees, PH-trees and R*-trees.

BB-Trees offer very fast inserts, as these need to (1) traverse the IST to locate the responsible BB, which can be performed very efficiently¹⁰, and to (2) append the new data object to the corresponding array, similar to the scan. Thus, BB-Trees perform better than kd-trees, which need to chase many pointers when searching for the leaf node that becomes the parent object

⁹In a production environment, we would advise to handle reorganizations in background jobs executed by separate threads, which strongly increases the write performance.

¹⁰See Table 5.3 for runtimes of the evaluation of range queries on the IST; insertions are even faster as they boil down to point queries.

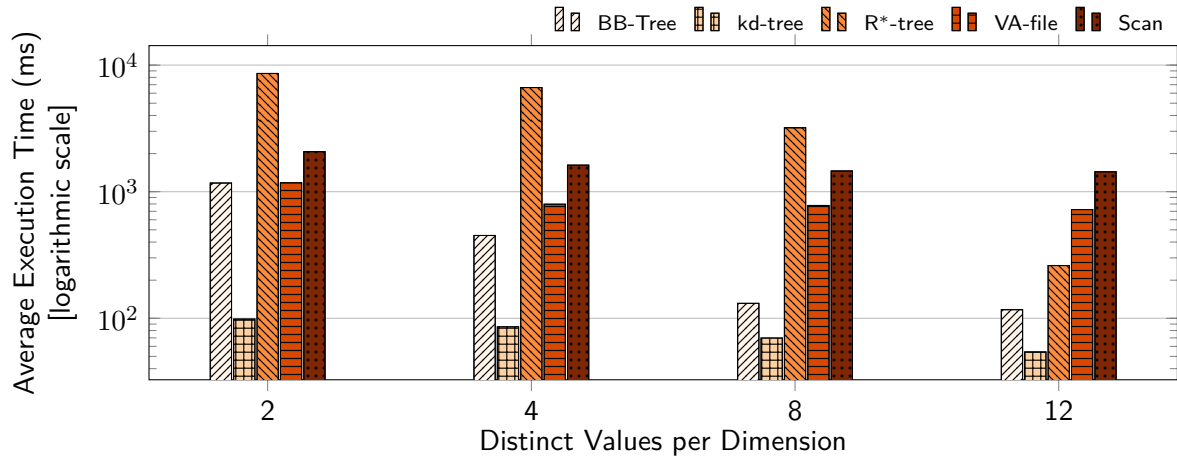


Figure 5.12: Performance of synthetic range queries with an average selectivity of 0.00002% ($\sigma = 0.0\%$) executed on ten Million 50-dimensional data objects from UNIFORM depending on the number of distinct values per dimension; PH-tree is omitted.

of the new node. The concept of elastic BB effectively reduces the frequency of rebalancing operations. When dynamically inserting ten Million data objects, regardless of the data set, BB-Trees needed only three reorganizations, which took 6.55s on average ($\sigma = 8.75s$). Smaller data sets require even less rebuilds.

Figure 5.14 shows the average execution time of deletes when removing an entire data set point by point. Note that the sizes of the indexes steadily decrease during the runtime of the experiment the more deletions have been processed. While the first delete operation is applied to n objects, the last delete operation must consider only one single object. The reported execution times of the BB-Tree include reorganizations that were conducted when more than 10% of all BB became empty. The used implementation of the PH-tree did not provide a delete operator. The delete performance of BB-Trees correlates with their point query performance, because they first execute a point query to obtain the BB holding the to-be-deleted-object. Once determined, they can remove the object from the respective leaf node. In this experiment, BB-Trees outperform all other competitors, even scans, except for the largest instance of GENOMIC.

5.7.9 Mixed Read/Write Workloads

Most real-life applications initialize databases with a bulk insert before applying search queries. Once built, inserts and deletes rarely happen. This experiment studies the contestants when running such workloads on 19-dimensional objects from GENOMIC. We exclude the VA-file, because it does not support single-tuple inserts or deletions. The vast majority of the data are inserted at the beginning of the workload, simulating a bulk insert. We consider ten Million data objects, of which we first insert 9,999,900. Subsequently, we run 100 inserts, 100 deletes, 2,800 point queries and 7,000 range queries in random order. For inserts, we use the remaining data objects, which were not bulk loaded. For point queries and deletes, we randomly choose existing data objects from the data set. This may result in point queries asking for non-existing data

5 BB-Trees: Processing Multidimensional Range Queries in Main Memory

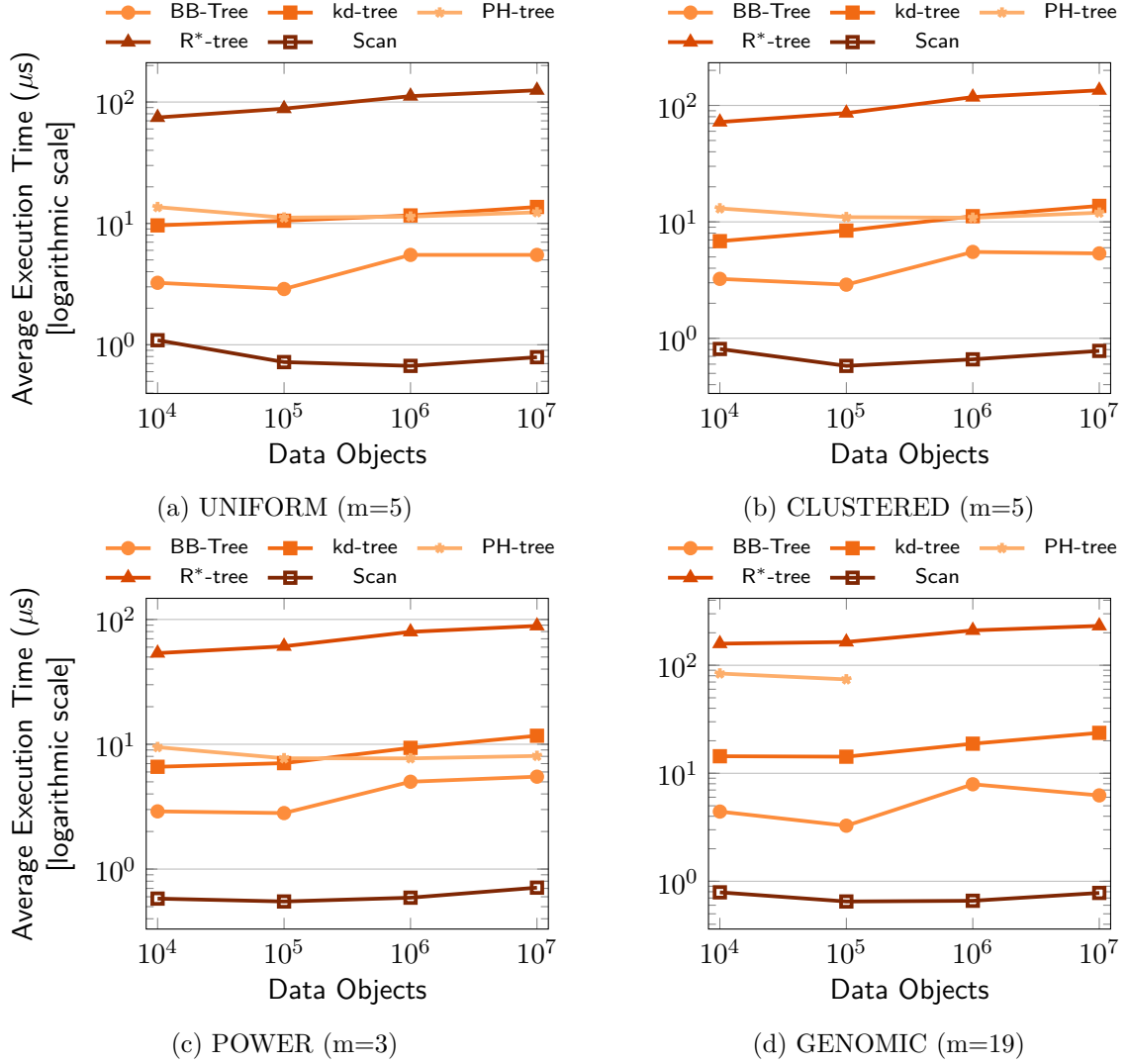


Figure 5.13: Performance of insert operations on the different data sets depending on the number of data objects.

objects, because they were previously deleted. For range queries, we use the Mixed Workload from GMRQB, which has an average selectivity of 1.58% ($\sigma = 3.58\%$) and consists of different query templates randomly mixed together, resembling a constantly changing query pattern. Once again the PH-tree ran out of memory and was excluded.

The box plot in Figure 5.15 summarizes the execution times excluding the bulk insert. Table 5.5 shows the runtime of the bulk insert and provides the average, minimum and maximum execution time of the remaining 10,000 queries. The BB-Tree achieves the highest performance in all cases. Only for the initial bulk insert, it is outperformed by the scan. The results of

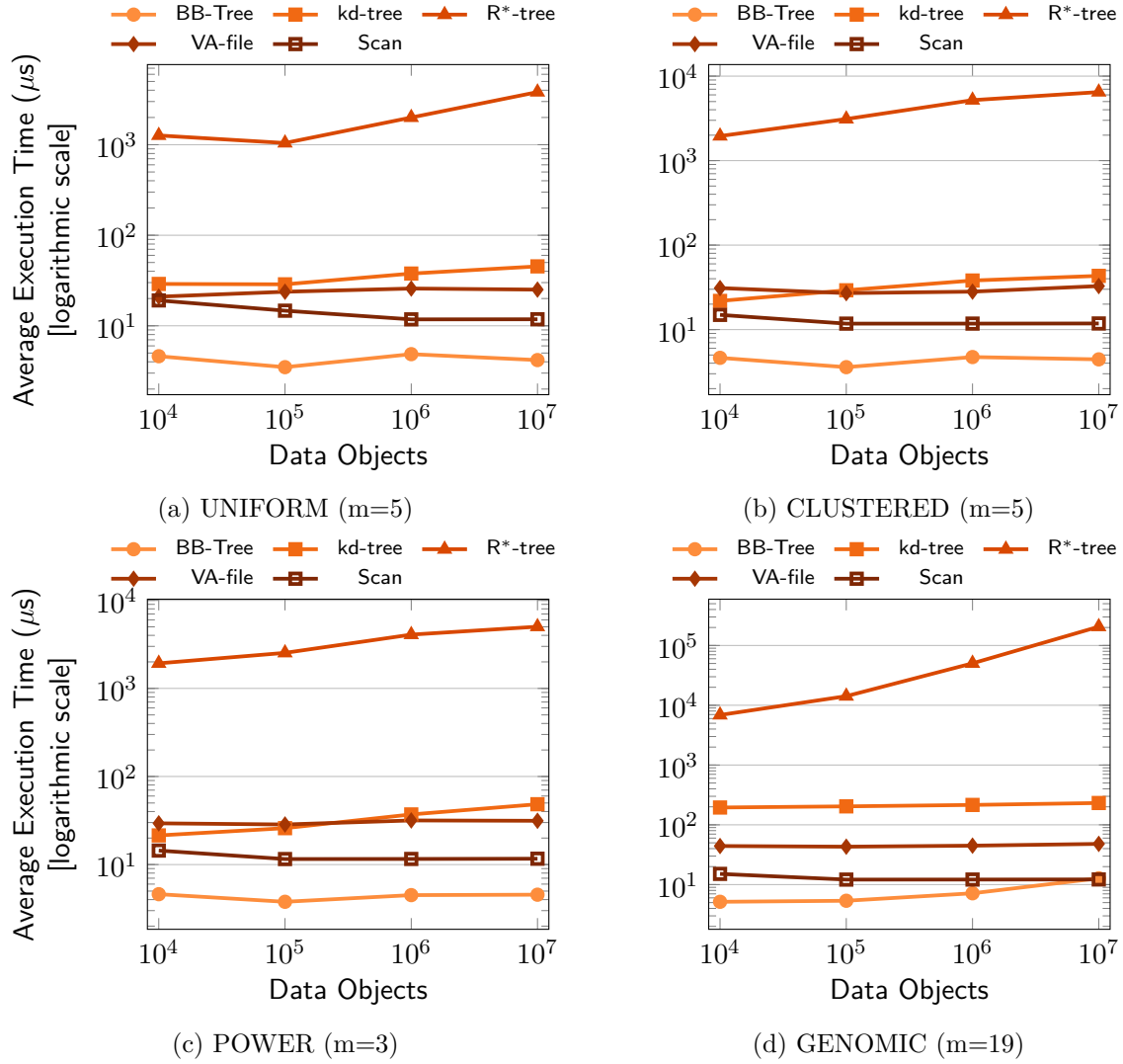


Figure 5.14: Performance of delete operations on the different data sets depending on the number of data objects.

this experiment prove that BB-Trees provide an high search performance without sacrificing insertions or deletions. On average, BB-Trees are 3.08X faster than the second best competitor, the sequential scan.

5.7.10 Scalability

Since modern CPUs provide an ever growing number of cores, it is important that multithreaded applications can scale their performance with an increasing degree of parallelism. This ex-

5 BB-Trees: Processing Multidimensional Range Queries in Main Memory

	Bulk Insert (s)	Average/Minimum/Maximum execution time (ms)
BB-Tree	54.7s	262.66ms / 0.005ms / 1,866.73ms
kd-tree	236.7s	128,735.5ms / 0.011ms / 4,842,752ms
PH-tree		Ran out of memory.
R*-tree	2,316s	2,735.16ms / 0.008ms / 7,735.76ms
VA-file		Does not support single-tuple inserts or deletes.
Scan	7.8s	809.83ms / 0.002ms / 3,117.46ms

Table 5.5: (1) Total execution time of the bulk insert and (2) average, minimum and maximum execution time of the remaining queries, both read and write operations, of the mixed workload.

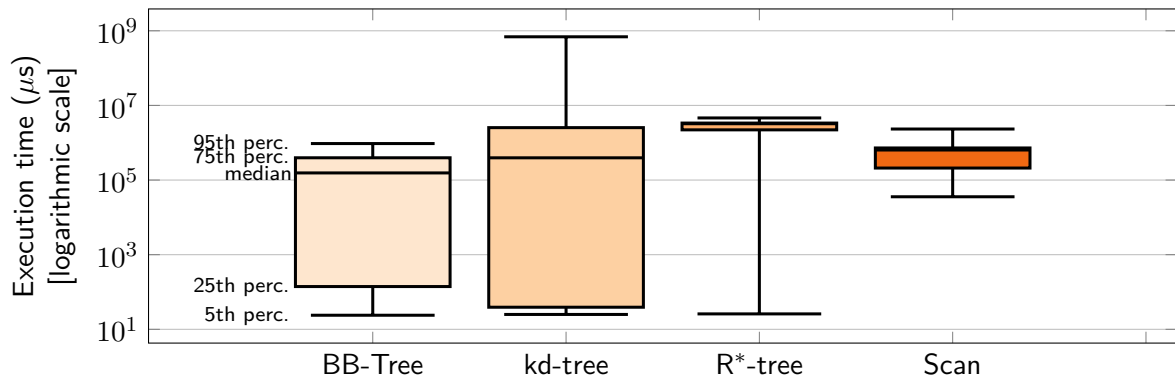


Figure 5.15: Execution times of the mixed workload consisting of inserts, deletes, point and range queries executed in random order (bulk insert is not included). The PH-tree ran out of memory. The VA-file was excluded, because it does not support single-tuple inserts.

periment studies the behavior of parallel BB-Trees when applying the Mixed Workload from GMRQB to ten Million objects from GENOMIC. We compare the runtimes to that of a single-threaded BB-Tree and a single-threaded sequential scan. We also consider a parallel scan, which (1) divides the data objects into t partitions, (2) concurrently scans each partition with one thread, and (3) concatenates the results of the individual partitions to obtain the total result set.

As shown in Figure 5.16, the performance of the parallel BB-Tree improves with the number of used threads up to a barrier established by the number of available physical cores (twelve on our evaluation machine). When using more threads than physical cores, the processor relies on hyper-threading, which provides only few benefits for the mostly compute-bound BB-Tree. Hyper-threading is mainly useful for memory-bound applications, like the scan, because it can hide memory latencies [Bulpin et al. 2004; Tian et al. 2003]. Using moderately more threads than supported by the hardware (24 on our evaluation machine), does neither provide benefits nor disadvantages. The scan (10.9X speed-up) benefits more from multithreading than the

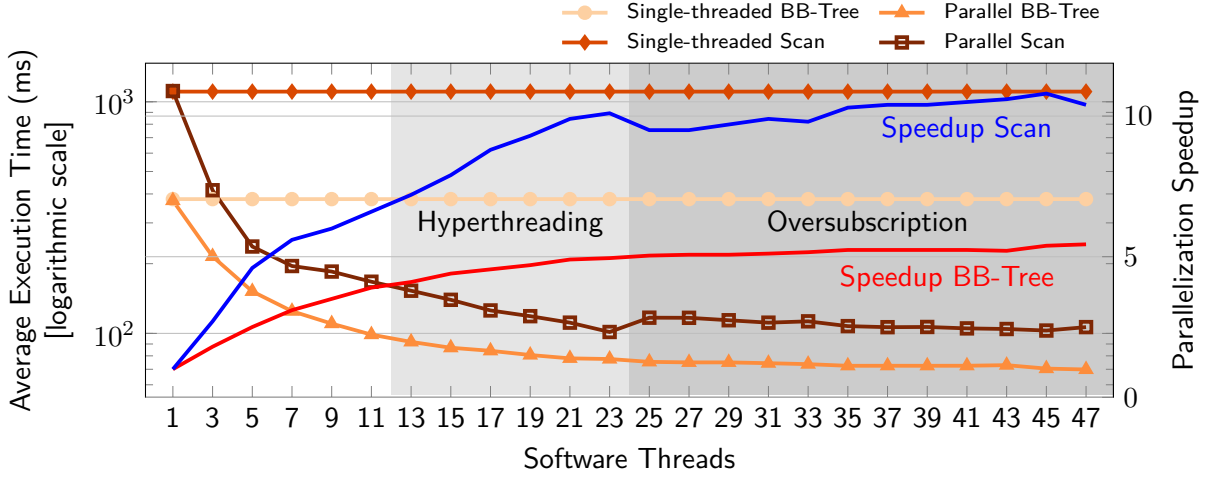


Figure 5.16: Performance of the Mixed Workload from GMRQB with an average selectivity of 1.58% ($\sigma = 3.58\%$) when applied to ten Million data objects from GENOMIC depending on the number of used software threads.

BB-Tree (5.5X speedup), because (a) the parallel scan can leverage hyper-threading and (b) scan-based MDRQ can be fully parallelized while BB-Trees must navigate the IST with a single thread. Nonetheless, the parallel search operator of BB-Trees still outperforms parallel scans regardless of the number of used threads.

5.7.11 Space Consumption

Especially in main-memory settings, it is important that index structures are as space efficient as possible. Figure 5.17 shows the space consumption of the contestants when holding ten Million data objects from the four data sets used in the evaluation. For GENOMIC, we omit the PH-tree, because it required more than the available 32GB of main memory, caused by its inefficiency for high dimensionalities. To enable a fair comparison, we report the space consumption of R*-trees and PH-trees as if they were using four-byte values to implement data objects. The BB-Tree offers the highest space efficiency among all index structures, showing the smallest space overhead over the sequential scan. For UNIFORM, BB-Trees require 614.42MB more memory than scans; for CLUSTERED, BB-Trees require 618.1MB more memory than scans; for POWER, BB-Trees require 546.13MB more memory than scans; and for GENOMIC, BB-Trees require 547.78MB more memory than scans. On average, for storing the four data sets, BB-Trees require 305.31MB less memory than kd-trees; BB-Trees require 160MB less memory than PH-trees; BB-Trees require 1,036.81MB less memory than R*-trees; and BB-Trees require 1,280.17MB less memory than VA-files.

BB are implemented with dynamic arrays, using C++’s `std::vector`, which strongly reduces the waste of memory space in the case of sparsely-filled BB. When using BB capacities larger than the one considered here ($b_{max} = 2,500$), the BB-Tree could further improve its space usage, employing an IST that is less deep and therefore contains less nodes. However, these

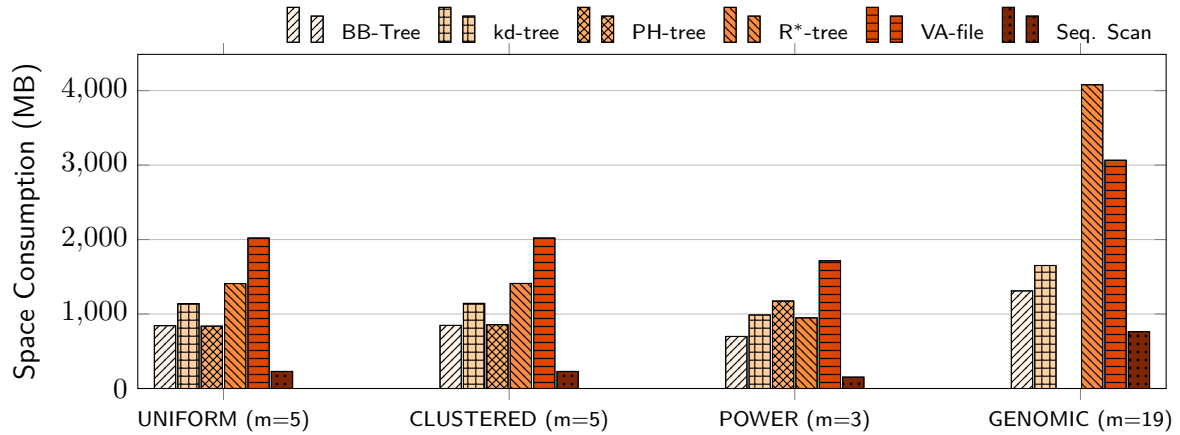


Figure 5.17: Space consumption of the competitors when storing ten Million data objects from the four data sets used in our evaluation.

improvements would come at the cost of decreased search performance for highly-selective query workloads.

5.8 Discussion

One of the foundational concepts behind BB-Trees are the elastic bubble buckets (BB), which (1) strongly reduce the pressure on rebuilding the linearized inner search tree (IST), (2) allow to instantly ingest changes into the index, and (3) increase the robustness towards hammered inserts. BB-Trees are designed for modern hardware architectures and show several advantages when deployed in main-memory settings:

- **High Cache Efficiency:** As shown in Table 5.4, BB-Trees are almost as cache efficient as sequential scans. Multiple properties of BB-Trees enable this behavior. First, inner nodes are tailored to cache line sizes, which offers high cache line utilization and reduces the number of data accesses when traversing the search tree. Second, BB-Trees use an almost-sequential access pattern to navigate their IST and evaluate BB with scans. Sequential data accesses utilize prefetched cache lines and strongly reduce cache and TLB misses.
- **High Space Efficiency:** Compared to the other MDIS considered in our evaluation, BB-Trees have low space requirements, adding only some indexing overhead. On top of raw data, which are kept in elastic BB, BB-Trees employ a k-ary search tree for pruning. The IST is typically very wide, as its fan out is tailored to the sizes of the cache lines. Moreover, BB-Trees linearize the entire IST and store all inner nodes together in one dense array, which eliminates pointers and further improves memory utilization.
- **Robustness:** BB-Trees are robust towards the dimensionality of the data space, as shown in Section 5.7.6. Foremost, the structure of the IST does not depend on the number of dimensions. The width of the tree depends on the sizes of the cache lines and the height

of the tree depends on the number of BB (or number of indexed data objects). Note that BB are evaluated with sequential scans making use of early breaks, which are likely to occur in high-dimensional data spaces.

- **Efficient Parallelization Scheme:** When searching in BB-Trees, the navigation of the IST is typically very fast, whereas the evaluation of the BB dominates the execution time of a query (see Section 5.7.5). As BB are processed with sequential scans, we can easily parallelize large parts of the search without introducing complex load balancing schemes, like, for instance, needed when concurrently traversing a tree structure [Yoo et al. 2005]. The performance of the parallel BB-Tree scales with the number of available physical cores (see Section 5.7.10).

Although BB-Trees represent a viable solution for a wide range of different applications, we also identify some limitations, which may be addressed in future work:

- The IST must be updated to reflect the current data distribution when super BB overflow or too many empty regular BB exist. Although the elasticity property of BB ensures that such rebalancing operations are seldom, the rebuild of the static array holding the linearized IST remains an expensive operation that, depending on the number of indexed data objects, can easily take several seconds and blocks insert or delete operations. For that reason, it is critical for write-heavy applications desiring a low update latency. Instead of handling the rebuild within the update operation causing the underflow or overflow, one could periodically rebuild the index in a background job executed by a separate thread, even if no BB has yet overflowed or underflowed. However, that raises a new challenge, namely finding a balanced rebuild frequency.
- Data sets with extremely correlated dimensions challenge the pruning power of BB-Trees, because they increase the probability that the IST contains duplicate delimiter values. In such cases, BB-Trees can prune less BB resulting in higher query execution times. However, also most other MDIS are negatively affected by correlated dimensions, as shown in Section 5.7.7.
- Different query selectivities call for different BB capacities: While high query selectivities are better handled in BB-Trees with small BB, low query selectivities are processed faster in BB-Trees with large BB. Instead of pinning the BB capacity to a constant value for the complete life cycle of a BB-Tree, one could periodically adapt the sizes of the leaf nodes to the current workload, which is especially useful for very high or very low query selectivities. However, such adaptations require statistics about the current query selectivities and are only useful for slowly-changing workload patterns.

5.9 Summary

In this chapter, we presented BB-Trees as a fast and space-efficient MDIS for storing and querying multidimensional point data in main memory. BB-Trees support point queries, complete- and partial-match range queries, while allowing dynamic updates. We compared BB-Trees

with state-of-the-art MDIS applying different synthetic and real-world workloads to different synthetic and real-world data sets with three to 100 dimensions. BB-Trees outperform all considered MDIS in executing range queries. Only for very low selectivities of 30% or more they are beaten by sequential scans. BB-Trees execute point queries almost as fast as the best competitor, the PH-tree; for high dimensionalities they even provide the best performance. BB-Trees also achieve the best insert and delete performance among all MDIS. Moreover, we presented a parallel variant that can scale its performance with the number of available CPU cores. In summary, BB-Trees combine high query and space efficiency with a pronounced robustness towards a wide range of data and workloads.

6 Summary and Outlook

6.1 Summary

In this thesis, we studied indexing for one-dimensional and multidimensional range queries on modern hardware architectures. We proposed two novel index structures, namely *cache-sensitive skip lists* (CSSL) aiming at one-dimensional domains and *BB-Trees* targeting multidimensional data spaces. Both index structures are designed for hardware features commonly available on modern server machines, especially large main-memory capacities, SIMD instructions and multithreading. In comprehensive evaluations, we compared CSSL and BB-Trees to state-of-the-art competitors.

Chapter 2 introduced fundamental terminologies and concepts relevant for this thesis. We provided an overview of popular multidimensional access methods offering search operators for MDRQ. We also discussed important features of modern server machines and showed how index structures can leverage these characteristics to improve performance. We proposed the Genomic Multidimensional Range Query Benchmark, which consists of eight realistic MDRQ templates that are applied to real-world genomic data.

Chapter 3 proposed CSSL as novel index structure for processing one-dimensional range queries in main memory and on modern CPUs. We showed how to adapt the memory layout of regular skip lists to the cache hierarchies of modern CPUs, strongly improving the cache efficiency of their range query operator. We also described how to exploit SIMD instructions to further speed-up the execution of range queries in skip lists. We compared CSSL to several state-of-the-art main-memory index structures. Using different data sets and workloads, we demonstrated the superiority of CSSL's range query operator in main-memory settings.

Chapter 4 studied the performance of traditional index structures for multidimensional range queries when deployed on modern hardware. We followed two techniques to partition a multidimensional data set, enabling parallel range query operators to process individual partitions with distinct threads, and introduced a SIMD-parallel algorithm for comparing an MDRQ search object to a multidimensional point object. Using these techniques, we conservatively adapted three popular index structures, namely the R^* -tree, the kd-tree, and the VA-file, and two scan flavors to the characteristics of modern server machines. A comprehensive evaluation indicated that the performance ratio between index probing and scanning changes when moving from traditional disk-based machines to current hardware architectures, as sequential scans become more efficient.

Chapter 5 presented BB-Trees as fast and space-efficient means to processing multidimensional point and range queries. Motivated by the observations described in Chapter 4, BB-Trees use a CPU-friendly memory layout that enables a mostly-sequential access pattern when evaluating typical queries, which maximizes cache line utilization and reduces cache misses. BB-Trees

show high robustness towards a wide range of different data and workloads, e. g., moderate and high dimensionalities (we evaluated up to 100 dimensions), low-cardinality dimensions, partial- and complete-match queries, etc. We also proposed a multithreaded variant that can leverage the parallel capabilities of modern CPUs to accelerate search queries. We compared BB-Trees to state-of-the-art multidimensional index structures and successfully demonstrated that BB-Trees achieve an outstanding range query performance, provide a fast point query operator, can efficiently ingest updates, and can accelerate the performance of their parallel search operator proportional to the number of available physical cores.

6.2 Outlook

We identify multiple future research directions based on the results of this thesis.

BB-Trees: Dynamic Adaptation of Bubble Bucket Capacities

In BB-Trees, the capacity of leaf nodes controls the ratio between navigation of the search tree, i. e., pruning, and scanning when evaluating search queries. Large *bubble buckets* (BB) are beneficial for low query selectivities, whereas small BB are preferable for high query selectivities. In Chapter 5, we empirically determined a BB capacity that offers high robustness towards a wide range of workloads and used it for all experiments. However, for very high or very low selectivities, a smaller or larger BB capacity could provide higher search efficiency. Instead of sticking to the same BB capacity for the entire lifetime of a BB-Tree, we could periodically adapt the capacities to the observed average selectivity of the current workload. In the optimal case, this adaptation is conducted immediately once the workload changes.

First, we would need to gain knowledge about the selectivities of the most recently processed search queries. To this end, we could extend the range query operator of BB-Trees such that it monitors the selectivities, defined by the size of the result set divided by the cardinality of the data set, at the end of the query evaluation without introducing much additional complexity. Second, we would need to decide when to change the BB capacities. As rebuilds need to rewrite the index anyway, we could integrate the adaptation into the reorganization operator. However, rebuilds are only invoked by update operations. To also enable adaptation in the case of read-heavy workloads, we would need to ensure that the reorganization is invoked periodically, even when no updates are performed. For instance, we might rebuild the index every 10,000-th read or write operation. To avoid the situation, where rebuilds triggered by search queries halt the database system, we could handle reorganization in background jobs executed by separate threads.

BB-Trees: Dynamic Adaptation of Delimiter Dimensions

By default, BB-Trees choose their delimiter dimensions in the order of the cardinalities when building the *inner search tree* (IST). Dimensions with many distinct values are moved to the upper tree levels and dimensions with few distinct values are moved to the lower tree levels. Consider a scenario where a m -dimensional data set is indexed by an IST of height $h < m$, i. e., the search tree splits in h of the m dimensions. If a BB-Tree is primarily queried with

workloads of partial-match MDRQ, where queries restrict less than m dimensions, the performance of the index crucially depends on the selection of its h delimiter dimensions, because any dimension constrained in the workload but not used in the index leads to a loss of pruning opportunities. If the particular selection of the queried dimensions changes over time, it would be highly beneficial to restructure the MDIS from time to time by adapting to the currently *hot* dimensions. Such scenarios are not uncommon in practice and occur, for instance, when different (interactive) analysis processes subsequently work on the data, where each process generates series of queries for a particular purpose, leading to a particular selection of dimensions in queries [Thorvaldsdóttir et al. 2013].

When rebuilding the IST, instead of only considering characteristics of the data, we may also take the current workload into account, moving dimensions queried *often* with a *high selectivity* to the upper tree levels and dimensions queried *rarely* and with a *low selectivity* to the lower tree levels. To this end, we would need to maintain statistics about the usage frequencies and the average selectivities that each dimension is queried with. Usage frequencies could be easily obtained by maintaining a counter for each dimension of the indexed data space, which is incremented whenever the dimension is restricted by a search query. Determining the average selectivities for single dimensions is more complicated, because, for each executed range query, we would need to split the m -dimensional MDRQ search object into m selection predicates and apply these to single dimensions of the data space. However, this step could be handled during index rebuilds by considering only the most recent queries, which minimizes the impact on the query evaluation (we still need to monitor the search objects), but increases the cost of the reorganization operator. Though, recall that we could speed-up this procedure by considering samples, which are instantiated anyway when rebuilding, instead of utilizing the entire data.

An adaptation of the delimiter dimensions to the recent query workload is promising, but there are also limitations. First, ordering the delimiter dimensions by their workload selectivity and usage frequency is only beneficial for BB-Trees, where less than m dimensions are used as delimiter. It does not provide any advantage in pruning leaf nodes for BB-Trees having a height larger than or equal to the dimensionality of the data space. Second, the benefits of an adaptation to the recent workload are the higher the more stable the running workload is and the more drastic it changes, when it changes. Third, although rebuilds of the IST are periodically invoked, they may not always represent the perfect point in time to perform the adaptation. Ideally, the delimiter dimensions are adapted whenever the workload pattern changes. However, a fast and efficient detection (or prediction) of workload changes is very challenging [Holze et al. 2007]. Fourth, though sampling reduces the complexity of determining single-dimension selectivities, our adaptation approach would still add overhead to the rebuild operator.

CSSL and BB-Trees: Cost Models

Database management systems require precise estimations of the cost of individual operators to choose between different execution plans when optimizing query performance [Manegold et al. 2002; Jarke et al. 1984]. To enable an integration of CSSL and BB-Trees into fully-fledged main-memory database systems, we must provide accurate cost models that can predict the performance of their query operators depending on a given search object. While traditional database systems optimize disk accesses, modern database systems aim to work as much as

6 Summary and Outlook

possible on data held on the CPU, hence reducing LLC misses requiring accesses to the main memory. Depending on the underlying hardware (cache sizes, cache replacement policies, etc.) and previous search requests, an access of a cache line may either produce a costly cache miss or be served from on-die caches. Thus, when proposing cost models for main-memory index structures, we should use the number of accessed cache lines as fundamental cost measure, providing an upper limit for the LLC miss rate. In the following, we sketch how we would design a cost model for the range query operators of CSSL and BB-Trees.

In CSSL, a range query navigates the skip list hierarchy to find the segment of the data list containing the smallest key satisfying the search object. Once this segment is determined, the search algorithm scans over the lowest fast lane to locate the segment of the data list holding the largest matching key. Given that keys are implemented with four-byte values, 16 keys fit into one cache line. Hence, assuming that each fast lane skips over less than 16 keys, which is very common in practice, the initial traversal over the h fast lanes accesses h cache lines at most (one cache line per fast lane level). Given that r keys satisfy the query object, processing the lowest fast lane accesses $r/16$ cache lines at most. In addition, the search algorithm visits two elements of the data list (the smallest and largest matching keys), which additionally loads two cache lines. Thus, in CSSL, the cost of a range query could be approximated with $h + r/16 + 2$ cache line accesses. Of course, this cost formula would have to be verified through experiments with different data distributions and different query workloads.

In BB-Trees, predicting the cost of a range query is more complicated, as its performance strongly depends on how many candidate BB need to be scanned to find the true results (B_{match}). B_{match} critically depends on the structure of the IST, i.e., the delimiter dimensions and values, and the query object. When implementing data objects with four-byte values, 16 dimension values fit into one 64-byte cache line. Given that BB have a maximum capacity of B_{max} m -dimensional objects, scanning a BB accesses $B_{max} * m/16$ cache lines at most. As BB may morph into super BB, consisting of up to k regular BB, we must also consider how many of the B_{match} candidate BB are super BB. Thus, when deriving a cost model for BB-Trees, we must consider multiple properties: (1) The number of candidate BB (B_{match}), which depends on the concrete IST and is the major challenge for accurate cost estimation, (2) the number of candidate BB that are super BB (B_{match_super}), (3) the BB capacity (B_{max}), and (4) the dimensionality of the data space (m). The total cost of a range query therefore boils down to roughly $((B_{match} - B_{match_super}) + (B_{match_super} * k)) * (B_{max} * m/16)$ cache line accesses. This cost formula would also have to be validated through comprehensive experiments.

BB-Trees: Nearest-Neighbor Search

This thesis focused on the processing of range queries in main memory. We proposed BB-Trees as a novel multidimensional index structure supporting range and point queries. In future work, we intend to extend the capabilities of BB-Trees with regards to executing *nearest-neighbor* (NN) search queries, which is another popular query type frequently applied to multidimensional data.

NN queries retrieve the point that is closest to a given search object, which is a point itself. NN queries use metrics, e.g., euclidean distance, to define the distance between two points in the data space. While NN queries retrieve the point closest to a search object, *k-nearest neighbor* (k-NN) search retrieves the k points that are closest to a given search object. Interestingly, NN

queries can be also used for finding the object most similar to a search object when defining similarity as an inverse of the distance function, assuming that similar objects are nearby in the data space and non-similar objects are far away from each other.

BB-Trees partition the entire data space into disjoint regions, which are covered by distinct bubble buckets. BB-Trees could implement NN queries by executing three steps. First, locate the BB, which covers the region the search object belongs to. Also, all neighboring regions (and respective BB) must be considered. Second, scan these candidate BB to determine the distances between the data objects and the search object. Technically, we could manage the obtained distances (and the associated data objects) in a binary min heap. Third, return the data object that has the smallest distance to the search object, which boils down to extracting the minimum element from the heap built in the previous step. Similarly, BB-Trees could implement k-NN queries by extracting k elements from the heap. Note that if the regions obtained in the first step contained less than k elements, we would have to extend the search to further regions.

For low-dimensional data, BB-Trees could implement NN efficiently, leading to a complexity of $O(h * \log(k) + B_{match} * b_{max} * m)$ ¹. The first step of the algorithm requires a navigation of the IST to locate the BB covering the search object, which has a complexity of $O(h * \log(k))$ as shown in Section 5.5. The second step investigates the B_{match} candidate BB, each holding up to b_{max} m -dimensional data objects. For instance, in the two-dimensional domain, up to nine candidate BB ($B_{match} = 9$) would have to be considered, as every region has eight neighbors. The third step can be done in $O(1)$. When the dimensionality of the data space increases, a growing number of data objects (or candidate BB) would have to be investigated, as implied by the *curse of dimensionality* [Beyer et al. 1999], thus strongly increasing the complexity of the second step, which becomes linear in n . However, most MDIS also cannot efficiently handle NN search queries in high-dimensional data spaces, but degenerate to sequential scans [Weber et al. 1998].

¹Note that k here denotes the fan out of the inner nodes.

Appendix A: Genomic Multidimensional Range Query Benchmark

Query Template 1

Listing 1: Query Template 1.

```
1 SELECT * FROM variants
2   WHERE chromosome = ?
3     AND location BETWEEN ? AND ?;
```

Query Template 2

Listing 2: Query Template 2.

```
1 SELECT * FROM variants
2   WHERE chromosome BETWEEN ? AND ?
3     AND location BETWEEN ? AND ?
4     AND quality BETWEEN ? AND ?
5     AND depth BETWEEN ? AND ?
6     AND allele_freq BETWEEN ? AND ?;
```

Query Template 3

Listing 3: Query Template 3.

```
1 SELECT * FROM variants
2   WHERE chromosome BETWEEN ? AND ?
3     AND location BETWEEN ? AND ?
4     AND gender = ?;
```

Query Template 4

Listing 4: Query Template 4.

```
1 SELECT * FROM variants
2   WHERE chromosome BETWEEN ? AND ?
3     AND location BETWEEN ? AND ?
4     AND gender = ?
5     AND population = '?';
```

Query Template 5

Listing 5: Query Template 5.

Appendix A: Genomic Multidimensional Range Query Benchmark

```
1 SELECT * FROM variants
2   WHERE chromosome BETWEEN ? AND ?
3         AND location BETWEEN ? AND ?
4         AND gender = ?
5         AND population = '?'
6         AND relationship = '?';
```

Query Template 6

Listing 6: Query Template 6.

```
1 SELECT * FROM variants
2   WHERE chromosome BETWEEN ? AND ?
3         AND location BETWEEN ? AND ?
4         AND gender = ?
5         AND population = '?'
6         AND relationship = '?'
7         AND family_id BETWEEN ? AND ?;
```

Query Template 7

Listing 7: Query Template 7.

```
1 SELECT * FROM variants
2   WHERE chromosome BETWEEN ? AND ?
3         AND location BETWEEN ? AND ?
4         AND gender = ?
5         AND population = '?'
6         AND relationship = '?'
7         AND family_id BETWEEN ? AND ?
8         AND variant_id BETWEEN ? AND ?;
```

Query Template 8

Listing 8: Query Template 8.

```
1 SELECT * FROM variants
2   WHERE chromosome BETWEEN ? AND ?
3         AND location BETWEEN ? AND ?
4         AND quality BETWEEN ? AND ?
5         AND depth BETWEEN ? AND ?
6         AND allele_freq BETWEEN ? AND ?
7         AND ref_base = '?'
8         AND alt_base = '?'
9         AND ancestral_allele = '?'
10        AND variant_id BETWEEN ? AND ?
11        AND sample_id BETWEEN ? AND ?
12        AND gender = ?
13        AND family_id BETWEEN ? AND ?
14        AND population = '?'
15        AND relationship = '?'
16        AND variant_type = '?'
17        AND genotype = '?';
```

```
18     AND genotype_quality BETWEEN ? AND ?
19     AND read_depth BETWEEN ? AND ?
20     AND haplotype_quality BETWEEN ? AND ?;
```

Appendix A: Genomic Multidimensional Range Query Benchmark

Bibliography

- Abadi, Daniel J., Samuel Madden, and Miguel Ferreira (2006). “Integrating compression and execution in column-oriented database systems”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 671–682.
- Abadi, Daniel, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden (2013). “The Design and Implementation of Modern Column-Oriented Database Systems”. In: *Foundations and Trends in Databases* 5.3, pp. 197–280.
- Ailamaki, Anastassia, David J. DeWitt, Mark D. Hill, and David A. Wood (1999). “DBMSs on a Modern Processor: Where Does Time Go?” In: *Proceedings of the 25th International Conference on Very Large Data Bases*, pp. 266–277.
- Alvarez, Victor, Stefan Richter, Xiao Chen, and Jens Dittrich (2015). “A comparison of adaptive radix trees and hash tables”. In: *Proceedings of the 31st IEEE International Conference on Data Engineering*, pp. 1227–1238.
- Arge, Lars, Mark de Berg, Herman J. Haverkort, and Ke Yi (2004). “The Priority R-Tree: A Practically Efficient and Worst-Case Optimal R-Tree”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 347–358.
- Bakkum, Peter and Kevin Skadron (2010). “Accelerating SQL database operations on a GPU with CUDA”. In: *Proceedings of the 3rd Workshop on General Purpose Processing on Graphics Processing Units*, pp. 94–103.
- Bayer, Rudolf (1997). “The Universal B-Tree for Multidimensional Indexing: general Concepts”. In: *Proceedings of the International Conference on Worldwide Computing and Its Applications*, pp. 198–209.
- Bayer, Rudolf and Edward M. McCreight (1972). “Organization and Maintenance of Large Ordered Indices”. In: *Acta Informatica* 1, pp. 173–189.
- Beckmann, Norbert, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger (1990). “The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 322–331.
- Bentley, Jon Louis (1975). “Multidimensional Binary Search Trees Used for Associative Searching”. In: *Communications of the ACM* 18.9, pp. 509–517.
- Bentley, Jon Louis and Jerome H. Friedman (1979). “Data Structures for Range Searching”. In: *ACM Computing Surveys* 11.4, pp. 397–409.
- Berchtold, Stefan, Christian Böhm, Bernhard Braunmüller, Daniel A. Keim, and Hans-Peter Kriegel (1997). “Fast Parallel Similarity Search in Multimedia Databases”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1–12.
- Berchtold, Stefan, Christian Böhm, Daniel A. Keim, Hans-Peter Kriegel, and Xiaowei Xu (2000). “Optimal Multidimensional Query Processing Using Tree Striping”. In: *Proceedings of the Second International Conference on Data Warehousing and Knowledge Discovery*, pp. 244–257.

Bibliography

- Berchtold, Stefan, Daniel A. Keim, and Hans-Peter Kriegel (1996). “The X-tree : An Index Structure for High-Dimensional Data”. In: *Proceedings of the 22th International Conference on Very Large Data Bases*, pp. 28–39.
- Beyer, Kevin S., Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft (1999). “When Is ”Nearest Neighbor” Meaningful?” In: *7th International Conference on Database Theory*, pp. 217–235.
- Böhm, Christian, Stefan Berchtold, and Daniel A. Keim (2001). “Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases”. In: *ACM Computing Surveys* 33.3, pp. 322–373.
- Böhm, Matthias, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, Dirk Habich, and Wolfgang Lehner (2011). “Efficient In-Memory Indexing with Generalized Prefix Trees”. In: *Datenbanksysteme für Business, Technologie und Web, 14. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme"*, pp. 227–246.
- Boncz, Peter A., Stefan Manegold, and Martin L. Kersten (1999). “Database Architecture Optimized for the New Bottleneck: Memory Access”. In: *Proceedings of the 25th International Conference on Very Large Data Bases*, pp. 54–65.
- (2009). “Database Architecture Evolution: Mammals Flourished long before Dinosaurs became Extinct”. In: *PVLDB* 2.2, pp. 1648–1653.
- Bozkaya, Tolga and Z. Meral Özsoyoglu (1999). “Indexing Large Metric Spaces for Similarity Search Queries”. In: *ACM Transactions on Database Systems* 24.3, pp. 361–404.
- Broneske, David, Sebastian Breß, and Gunter Saake (2014). “Database Scan Variants on Modern CPUs: A Performance Study”. In: *Proceedings of the 2nd International Workshop on In Memory Data Management and Analytics*, pp. 1–15.
- Broneske, David, Veit Köppen, Gunter Saake, and Martin Schäler (2017a). “Accelerating Multi-Column Selection Predicates in Main-Memory - The Elf Approach”. In: *Proceedings of the 33rd IEEE International Conference on Data Engineering*, pp. 647–658.
- Broneske, David and Martin Schäler (2017b). “Single Instruction Multiple Data - Not Everything is a Nail for this Hammer”. In: *Failed Aspirations in Database Systems*.
- Bryant, Randal E, O’Hallaron David Richard, and O’Hallaron David Richard (2003). *Computer systems: a programmer’s perspective*. Vol. 2. Prentice Hall Upper Saddle River.
- Bulpin, James R and Ian A Pratt (2004). “Multiprogramming performance of the Pentium 4 with Hyper-Threading”. In: *Second Annual Workshop on Duplicating, Deconstruction and Debunking*. Citeseer, p. 53.
- Buluç, Aydin and Kamesh Madduri (2011). “Parallel breadth-first search on distributed memory systems”. In: *Conference on High Performance Computing Networking, Storage and Analysis*, 65:1–65:12.
- Calder, Brad, Chandra Krintz, Simmi John, and Todd M. Austin (1998). “Cache-Conscious Data Placement”. In: *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 139–149.
- Carlson, Josiah L. (2013). *Redis in Action*. Greenwich, CT, USA: Manning Publications Co. ISBN: 1617290858, 9781617290855.
- Chakrabarti, Kaushik, Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim (2001). “Approximate query processing using wavelets”. In: *VLDB Journal* 10.2-3, pp. 199–223.

- Chan, Chee Yong and Yannis E. Ioannidis (1998). “Bitmap Index Design and Evaluation”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 355–366.
- Chang, Jo-Mei and King-Sun Fu (1981). “Extended Kd tree database organization: A dynamic multiattribute clustering method”. In: *IEEE Transactions on Software Engineering* 3, pp. 284–290.
- Chaudhuri, Surajit and Umeshwar Dayal (1997). “An Overview of Data Warehousing and OLAP Technology”. In: *SIGMOD Record* 26.1, pp. 65–74.
- Chen, Jack, Samir Jindel, Robert Walzer, Rajkumar Sen, Nika Jimsheleishvili, and Michael Andrews (2016). “The MemSQL Query Optimizer: A modern optimizer for real-time analytics in a distributed database”. In: *PVLDB* 9.13, pp. 1401–1412.
- Choi, Byn, Rakesh Komuravelli, Victor Lu, Hyojin Sung, Robert L. Bocchino Jr., Sarita V. Adve, and John C. Hart (2010). “Parallel SAH k-D tree construction”. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on High Performance Graphics*, pp. 77–86.
- Choudhary, Pratik, John Shin, Yongyin Wang, Mark L. Evans, Peter J. Hammond, David Kerr, James A.M. Shaw, John C. Pickup, and Stephanie A. Amiel (2011). “Insulin Pump Therapy With Automated Insulin Suspension in Response to Hypoglycemia”. In: *Diabetes Care* 34.9, pp. 2023–2025.
- Ciaccia, Paolo, Marco Patella, and Pavel Zezula (1997). “M-tree: An Efficient Access Method for Similarity Search in Metric Spaces”. In: *Proceedings of the 23rd International Conference on Very Large Data Bases*, pp. 426–435.
- Codd, E. F. (1970). “A Relational Model of Data for Large Shared Data Banks”. In: *Communications of the ACM* 13.6, pp. 377–387.
- Comer, Douglas (1979). “The Ubiquitous B-Tree”. In: *ACM Computing Surveys* 11.2, pp. 121–137.
- Council, National Research (2011). *The Future of Computing Performance: Game Over or Next Level?* National Academies Press. ISBN: 978-0-309-15951-7, 978-0-309-21164-2.
- Danecek, Petr, Adam Auton, Gonçalo R. Abecasis, Cornelis A. Albers, Eric Banks, Mark A. DePristo, Robert E. Handsaker, Gerton Lunter, Gabor T. Marth, Stephen T. Sherry, Gilean McVean, and Richard Durbin (2011). “The variant call format and VCFtools”. In: *Bioinformatics* 27.15, pp. 2156–2158.
- Das, Dinesh, Jiaqi Yan, Mohamed Zaït, Satyanarayana R. Valluri, Nirav Vyas, Ramarajan Krishnamachari, Prashant Gaharwar, Jesse Kamp, and Niloy Mukherjee (2015). “Query Optimization in Oracle 12c Database In-Memory”. In: *PVLDB* 8.12, pp. 1770–1781.
- DeWitt, David J. and Jim Gray (1992). “Parallel Database Systems: The Future of High Performance Database Systems”. In: *Communications of the ACM* 35.6, pp. 85–98.
- Diaconu, Cristian, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling (2013). “Hekaton: SQL server’s memory-optimized OLTP engine”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1243–1254.
- Drepper, Ulrich (2007). “What every programmer should know about memory”. In: *Technical Report*.

Bibliography

- Faerber, Franz, Alfons Kemper, Per-Åke Larson, Justin J. Levandoski, Thomas Neumann, and Andrew Pavlo (2017). “Main Memory Database Systems”. In: *Foundations and Trends in Databases* 8.1-2, pp. 1–130.
- Faloutsos, Christos (1985). “Access Methods for Text”. In: *ACM Computing Surveys* 17.1, pp. 49–74.
- Faloutsos, Christos and Ibrahim Kamel (1994). “Beyond Uniformity and Independence: Analysis of R-trees Using the Concept of Fractal Dimension”. In: *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 4–13.
- Finkel, Raphael A. and Jon Louis Bentley (1974). “Quad Trees: A Data Structure for Retrieval on Composite Keys”. In: *Acta Informatica* 4, pp. 1–9.
- Flynn, Michael J. (1972). “Some Computer Organizations and Their Effectiveness”. In: *IEEE Transactions on Computers* 21.9, pp. 948–960.
- Fomitchev, Mikhail and Eric Ruppert (2004). “Lock-free linked lists and skip lists”. In: *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing*, pp. 50–59.
- Gaede, Volker and Oliver Günther (1998). “Multidimensional Access Methods”. In: *ACM Computing Surveys* 30.2, pp. 170–231.
- Garcia-Molina, Hector, Jeffrey D. Ullman, and Jennifer Widom (2000). *Database System Implementation*. Prentice-Hall. ISBN: 0-13-040264-8.
- Germann, Ulrich, Eric Joanis, and Samuel Larkin (2009). “Tightly packed tries: How to fit large models into memory, and make them load fast, too”. In: *Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing*. Association for Computational Linguistics, pp. 31–39.
- Graefe, Goetz (1990). “Encapsulation of Parallelism in the Volcano Query Processing System”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 102–111.
- (1994). “Volcano - An Extensible and Parallel Query Evaluation System”. In: *IEEE Transactions on Knowledge and Data Engineering* 6.1, pp. 120–135.
- Gubbi, Jayavardhana, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami (2013). “Internet of Things (IoT): A vision, architectural elements, and future directions”. In: *Future Generation Computer Systems* 29.7, pp. 1645–1660.
- Guttman, Antonin (1984). “R-Trees: A Dynamic Index Structure for Spatial Searching”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 47–57.
- Hakenberg, Jörg, Wei-Yi Cheng, Philippe E. Thomas, Ying-Chih Wang, Andrew V. Uzilov, and Rong Chen (2016). “Integrating 400 million variants from 80,000 human samples with extensive annotations: towards a knowledge base to analyze disease cohorts”. In: *BMC Bioinformatics* 17, p. 24.
- Hellerstein, Joseph M., Jeffrey F. Naughton, and Avi Pfeffer (1995). “Generalized Search Trees for Database Systems”. In: *Proceedings of the 21th International Conference on Very Large Data Bases*, pp. 562–573.
- Herlihy, Maurice, Yossi Lev, Victor Luchangco, and Nir Shavit (2006). “A provably correct scalable concurrent skip list”. In: *Conference On Principles of Distributed Systems*. Citeseer.

- Hinterberger, Hans, Kathrin Anne Meier, and Hans Gilgen (1994). “Spatial Data Reallocation Based on Multidimensional Range Queries - A Contribution to Data Management for the Earth Sciences”. In: *Seventh International Working Conference on Scientific and Statistical Database Management*, pp. 228–239.
- Ho, Ching-Tien, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant (1997). “Range Queries in OLAP Data Cubes”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 73–88.
- Hoel, Erik G. and Hanan Samet (1992). “A Qualitative Comparison Study of Data Structures for Large Line Segment Databases”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 205–214.
- Hofmann, Johannes, Jan Treibig, Georg Hager, and Gerhard Wellein (2014). “Comparing the performance of different x86 SIMD instruction sets for a medical imaging application on modern multi- and manycore chips”. In: *Proceedings of the Workshop on Programming models for SIMD/Vector processing*, pp. 57–64.
- Holloway, Allison L., Vijayshankar Raman, Garret Swart, and David J. DeWitt (2007). “How to barter bits for chronons: compression and bandwidth trade offs for database scans”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 389–400.
- Holze, Marc and Norbert Ritter (2007). “Towards workload shift detection and prediction for autonomic databases”. In: *Proceedings of the First Ph.D. Workshop in CIKM, Sixteenth ACM Conference on Information and Knowledge Management*, pp. 109–116.
- Hwang, Sohyun, Eiru Kim, Insuk Lee, and Edward M Marcotte (2015). “Systematic comparison of variant calling pipelines using gold standard personal exome variants”. In: *Scientific reports* 5, p. 17875.
- International Human Genome Sequencing Consortium (2004). “Finishing the euchromatic sequence of the human genome”. In: *Nature* 431.7011, p. 931.
- Jacob, Bruce, Spencer Ng, and David Wang (2010). *Memory systems: cache, DRAM, disk*. Morgan Kaufmann.
- Jagadish, H. V. (1990). “Spatial Search with Polyhedra”. In: *Proceedings of the 6th IEEE International Conference on Data Engineering*, pp. 311–319.
- Jagadish, H. V., Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, and Torsten Suel (1998). “Optimal Histograms with Quality Guarantees”. In: *Proceedings of the 24th International Conference on Very Large Data Bases*, pp. 275–286.
- Jahn, Marco, Marc Jentsch, Christian R Prause, Ferry Pramudianto, Amro Al-Akkad, and Rene Reiners (2010). “The energy aware smart home”. In: *5th International Conference on Future Information Technology*. IEEE, pp. 1–8.
- Jarke, Matthias and Jürgen Koch (1984). “Query Optimization in Database Systems”. In: *ACM Computing Surveys* 16.2, pp. 111–152.
- Ji, Yuanzhen, Thomas Heinze, and Zbigniew Jerzak (2013). “HUGO: real-time analysis of component interactions in high-tech manufacturing equipment (industry article)”. In: *The 7th ACM International Conference on Distributed Event-Based Systems*, pp. 87–96.
- Kallman, Robert, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and

Bibliography

- Daniel J. Abadi (2008). “H-store: a high-performance, distributed main memory transaction processing system”. In: *PVLDB* 1.2, pp. 1496–1499.
- Kamel, Ibrahim and Christos Faloutsos (1994). “Hilbert R-tree: An Improved R-tree using Fractals”. In: *Proceedings of the 20th International Conference on Very Large Data Bases*, pp. 500–509.
- Kanth, Kothuri Venkata Ravi, Siva Ravada, and Daniel Abugov (2002). “Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 546–557.
- Katayama, Norio and Shin’ichi Satoh (1997). “The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 369–380.
- Khamayseh, Ahmed K and Glen Hansen (2007). “Use of the Spatial kd-Tree in Computational Physics Applications”. In: *Communications in Computational Physics* 2.
- Kim, Changkyu, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey (2010). “FAST: fast architecture sensitive tree search on modern CPUs and GPUs”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 339–350.
- Kim, Kihong, Sang Kyun Cha, and Keunjoo Kwon (2001). “Optimizing Multidimensional Index Trees for Main Memory Access”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 139–150.
- King, Mary-Claire and Allan C Wilson (1975). “Evolution at two levels in humans and chimpanzees”. In: *Science* 188.4184, pp. 107–116.
- Kissinger, Thomas, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner (2012). “KISS-Tree: smart latch-free in-memory indexing on modern architectures”. In: *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, pp. 16–23.
- Kooi, Robert (1980). “The Optimization of Queries in Relational Databases”. PhD thesis. Case Western Reserve University.
- Koudas, Nick, Christos Faloutsos, and Ibrahim Kamel (1996). “Declustering Spatial Databases on a Multi-Computer Architecture”. In: *Proceedings of the 5th International Conference on Extending Database Technology*, pp. 592–614.
- Kreveld, Marc J. van and Mark H. Overmars (1991). “Divided k-d Trees”. In: *Algorithmica* 6.6, pp. 840–858.
- Labrinidis, Alexandros and H. V. Jagadish (2012). “Challenges and Opportunities with Big Data”. In: *PVLDB* 5.12, pp. 2032–2033.
- Lahiri, Tirthankar, Marie-Anne Neimat, and Steve Folkman (2013). “Oracle TimesTen: An In-Memory Database for Enterprise Applications”. In: *IEEE Data Engineering Bulletin* 36.2, pp. 6–13.
- Lang, Harald, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper (2016). “Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 311–326.
- Lawder, Jonathan K. and Peter J. H. King (2001). “Querying Multi-dimensional Data Indexed Using the Hilbert Space-filling Curve”. In: *SIGMOD Record* 30.1, pp. 19–24.

- Leis, Viktor, Alfons Kemper, and Thomas Neumann (2013). “The adaptive radix tree: ARTful indexing for main-memory databases”. In: *Proceedings of the 29th IEEE International Conference on Data Engineering*, pp. 38–49.
- Lemahieu, Wilfried, Seppe vanden Broucke, and Bart Baesens (2018). *Principles of Database Management: The Practical Guide to Storing, Managing and Analyzing Big and Small Data*. Cambridge University Press.
- Leutenegger, Scott T., J. M. Edgington, and Mario A. López (1997). “STR: A Simple and Efficient Algorithm for R-Tree Packing”. In: *Proceedings of the 13th IEEE International Conference on Data Engineering*, pp. 497–506.
- Levandoski, Justin J., David B. Lomet, and Sudipta Sengupta (2013). “The Bw-Tree: A B-tree for new hardware platforms”. In: *Proceedings of the 29th IEEE International Conference on Data Engineering*, pp. 302–313.
- Li, Heng (2011). “Tabix: fast retrieval of sequence features from generic TAB-delimited files”. In: *Bioinformatics* 27.5, pp. 718–719.
- Li, Xin, Young-Jin Kim, Ramesh Govindan, and Wei Hong (2003). “Multi-dimensional range queries in sensor networks”. In: *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, pp. 63–75.
- Li, Yinan and Jignesh M. Patel (2013). “BitWeaving: fast scans for main memory data processing”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 289–300.
- (2014). “WideTable: An Accelerator for Analytical Data Processing”. In: *PVLDB* 7.10, pp. 907–918.
- Liang, Weifa, Hui Wang, and Maria E. Orlowska (2000). “Range queries in dynamic OLAP data cubes”. In: *Data & Knowledge Engineering* 34.1, pp. 21–38.
- Lievre, Astrid, Jean-Baptiste Bachet, Delphine Le Corre, Valerie Boige, Bruno Landi, Jean-François Emile, Jean-François Côté, Gorana Tomasic, Christophe Penna, Michel Ducreux, et al. (2006). “KRAS mutation status is predictive of response to cetuximab therapy in colorectal cancer”. In: *Cancer research* 66.8, pp. 3992–3995.
- Lin, King-Ip, H. V. Jagadish, and Christos Faloutsos (1994). “The TV-Tree: An Index Structure for High-Dimensional Data”. In: *VLDB Journal* 3.4, pp. 517–542.
- Lipton, Richard J., Jeffrey F. Naughton, and Donovan A. Schneider (1990). “Practical Selectivity Estimation through Adaptive Sampling”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1–11.
- Lomet, David B. and Betty Salzberg (1990). “The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance”. In: *ACM Transactions on Database Systems* 15.4, pp. 625–658.
- Manegold, Stefan, Peter A. Boncz, and Martin L. Kersten (2000). “Optimizing database architecture for the new bottleneck: memory access”. In: *VLDB Journal* 9.3, pp. 231–246.
- (2002). “Generic Database Cost Models for Hierarchical Memory Systems”. In: *Proceedings of the 28th International Conference on Very Large Data Bases*, pp. 191–202.
- Mao, Yandong, Eddie Kohler, and Robert Tappan Morris (2012). “Cache craftiness for fast multicore key-value storage”. In: *Proceedings of the Seventh European Conference on Computer Systems*, pp. 183–196.

Bibliography

- Meagher, Donald (1982). “Geometric modeling using octree encoding”. In: *Computer Graphics and Image Processing* 19.2, pp. 129–147.
- Morrison, Donald R. (1968). “PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric”. In: *Journal of the ACM* 15.4, pp. 514–534.
- Mucci, Philip J, Shirley Browne, Christine Deane, and George Ho (1999). “PAPI: A portable interface to hardware performance counters”. In: *Proceedings of the department of defense HPCMP users group conference*. Vol. 710.
- Müller, Emmanuel, Stephan Günnemann, Ira Assent, and Thomas Seidl (2009). “Evaluating Clustering in Subspace Projections of High Dimensional Data”. In: *PVLDB* 2.1, pp. 1270–1281.
- Munro, J. Ian, Thomas Papadakis, and Robert Sedgewick (1992). “Deterministic Skip Lists”. In: *Proceedings of the Third Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms*, pp. 367–375.
- Nievergelt, Jürg, Hans Hinterberger, and Kenneth C. Sevcik (1984). “The Grid File: An Adaptable, Symmetric Multikey File Structure”. In: *ACM Transactions on Database Systems* 9.1, pp. 38–71.
- Orenstein, Jack A. and T. H. Merrett (1984). “A Class of Data Structures for Associative Searching”. In: *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp. 181–190.
- Pagel, Bernd-Uwe, Hans-Werner Six, Heinrich Toben, and Peter Widmayer (1993). “Towards an Analysis of Range Query Performance in Spatial Data Structures”. In: *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 214–221.
- Plattner, Hasso (2009). “A common database approach for OLTP and OLAP using an in-memory column database”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1–2.
- Plattner, Hasso and Alexander Zeier (2011). *In-memory data management: an inflection point for enterprise applications*. Springer.
- Pohl, Angela, Biagio Cosenza, Mauricio Alvarez Mesa, Chi Ching Chi, and Ben H. H. Juurlink (2016). “An evaluation of current SIMD programming models for C++”. In: *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, 3:1–3:8.
- Polychroniou, Orestis, Arun Raghavan, and Kenneth A. Ross (2015). “Rethinking SIMD Vectorization for In-Memory Databases”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1493–1508.
- Poosala, Viswanath (1997). “Histogram-Based Estimation Techniques in Database Systems”. PhD thesis. University of Wisconsin-Madison.
- Poosala, Viswanath and Yannis E. Ioannidis (1997). “Selectivity Estimation Without the Attribute Value Independence Assumption”. In: *Proceedings of the 23rd International Conference on Very Large Data Bases*, pp. 486–495.
- Procopiuc, Octavian, Pankaj K. Agarwal, Lars Arge, and Jeffrey Scott Vitter (2003). “Bkd-Tree: A Dynamic Scalable kd-Tree”. In: *Proceedings of the 8th International Symposium on Advances in Spatial and Temporal Databases*, pp. 46–65.

- Pugh, William (1990). “Skip Lists: A Probabilistic Alternative to Balanced Trees”. In: *Communications of the ACM* 33.6, pp. 668–676.
- Qi, Jianzhong, Yufei Tao, Yanchuan Chang, and Rui Zhang (2018). “Theoretically Optimal and Empirically Efficient R-trees with Strong Parallelizability”. In: *PVLDB* 11.5, pp. 621–634.
- Qiao, Lin, Vijayshankar Raman, Frederick Reiss, Peter J. Haas, and Guy M. Lohman (2008). “Main-memory scan sharing for multi-core CPUs”. In: *PVLDB* 1.1, pp. 610–621.
- Raman, Vijayshankar, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang (2013). “DB2 with BLU Acceleration: So Much More than Just a Column Store”. In: *PVLDB* 6.11, pp. 1080–1091.
- Ramsak, Frank, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt, and Rudolf Bayer (2000). “Integrating the UB-Tree into a Database System Kernel”. In: *Proceedings of the 26th International Conference on Very Large Data Bases*, pp. 263–272.
- Rao, Jun and Kenneth A. Ross (1999). “Cache Conscious Indexing for Decision-Support in Main Memory”. In: *Proceedings of the 25th International Conference on Very Large Data Bases*, pp. 78–89.
- (2000). “Making B⁺-Trees Cache Conscious in Main Memory”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 475–486.
- Robinson, John T. (1981). “The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indexes”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 10–18.
- Rödiger, Wolf, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann (2015). “High-Speed Query Processing over High-Speed Networks”. In: *PVLDB* 9.4, pp. 228–239.
- Roussopoulos, Nick and Daniel Leifker (1985). “Direct Spatial Search on Pictorial Databases Using Packed R-Trees”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 17–31.
- Rowstron, Antony, Dushyanth Narayanan, Austin Donnelly, Greg O’Shea, and Andrew Douglas (2012). “Nobody ever got fired for using Hadoop on a cluster”. In: *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*. ACM.
- Saecker, Michael and Volker Markl (2012). “Big Data Analytics on Modern Hardware Architectures: A Technology Survey”. In: *Business Intelligence - Second European Summer School, eBISS 2012*, pp. 125–149.
- Sagan, Hans (2012). *Space-filling curves*. Springer Science & Business Media.
- Saini, Subhash, Haoqiang Jin, Robert Hood, David Barker, Piyush Mehrotra, and Rupak Biswas (2011). “The impact of hyper-threading on processor resource utilization in production applications”. In: *18th International Conference on High Performance Computing*. IEEE, pp. 1–10.
- Schlegel, Benjamin, Rainer Gemulla, and Wolfgang Lehner (2009). “k-ary search on modern processors”. In: *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pp. 52–60.

Bibliography

- Schlegel, Benjamin, Thomas Willhalm, and Wolfgang Lehner (2011). “Fast Sorted-Set Intersection using SIMD Instructions”. In: *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, pp. 1–8.
- Schnitzer, Bernd and Scott T. Leutenegger (1999). “Master-Client R-Trees: A New Parallel R-Tree Architecture”. In: *Proceedings of the 11th International Conference on Scientific and Statistical Database Management*, pp. 68–77.
- Seeger, Bernhard and Hans-Peter Kriegel (1988). “Techniques for Design and Implementation of Efficient Spatial Access Methods”. In: *Proceedings of the 14th International Conference on Very Large Data Bases*, pp. 360–371.
- (1990). “The Buddy-Tree: An Efficient and Robust Access Method for Spatial Data Base Systems”. In: *Proceedings of the 16th International Conference on Very Large Data Bases*, pp. 590–601.
- Selinger, Patricia G., Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price (1979). “Access Path Selection in a Relational Database Management System”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 23–34.
- Sellis, Timos K., Nick Roussopoulos, and Christos Faloutsos (1987). “The R+-Tree: A Dynamic Index for Multi-Dimensional Objects”. In: *Proceedings of the 13th International Conference on Very Large Data Bases*, pp. 507–518.
- Smith, Alan Jay (1982). “Cache Memories”. In: *ACM Computing Surveys* 14.3, pp. 473–530.
- Sprenger, Stefan, Patrick Schäfer, and Ulf Leser (2018a). “BB-Tree: A practical and efficient main-memory index structure for multidimensional workloads”. In: *Submitted for publication*.
- (2018b). “Multidimensional Range Queries on Modern Hardware”. In: *Proceedings of the 30th International Conference on Scientific and Statistical Database Management*.
- Sprenger, Stefan, Steffen Zeuch, and Ulf Leser (2016). “Cache-Sensitive Skip List: Efficient Range Queries on Modern CPUs”. In: *7th International Workshop on Accelerating Data Analysis and Data Management Systems Using Modern Processor and Storage Architectures and 4th International Workshop on In-Memory Data Management and Analytics*, pp. 1–17.
- (2018c). “Exploiting Automatic Vectorization to Employ SPMD on SIMD Registers”. In: *34th IEEE International Conference on Data Engineering Workshops*, pp. 90–95.
- Sproull, Robert F. (1991). “Refinements to Nearest-Neighbor Searching in k-Dimensional Trees”. In: *Algorithmica* 6.4, pp. 579–589.
- Stonebraker, Michael, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O’Neil, Patrick E. O’Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik (2005). “C-Store: A Column-oriented DBMS”. In: *Proceedings of the 31st International Conference on Very Large Data Bases*, pp. 553–564.
- Stonebraker, Michael and Ariel Weisberg (2013). “The VoltDB Main Memory DBMS”. In: *IEEE Data Engineering Bulletin* 36.2, pp. 21–27.
- Taniar, David, Clement HC Leung, Wenny Rahayu, and Sushant Goel (2008). *High-performance parallel database processing and grid databases*. Vol. 67. John Wiley & Sons.
- Teubner, Jens and Louis Woods (2013). *Data Processing on FPGAs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers.

- The 1000 Genomes Project Consortium (2012). “An integrated map of genetic variation from 1,092 human genomes”. In: *Nature* 491.7422, p. 56.
- (2015). “A global reference for human genetic variation”. In: *Nature* 526.7571, pp. 68–74.
- Thorvaldsdóttir, Helga, James T. Robinson, and Jill P. Mesirov (2013). “Integrative Genomics Viewer (IGV): high-performance genomics data visualization and exploration”. In: *Briefings in Bioinformatics* 14.2, pp. 178–192.
- Tian, Xinmin, Yen-Kuang Chen, Milind Girkar, Steven Ge, Rainer Lienhart, and Sanjiv Shah (2003). “Exploring the Use of Hyper-Threading Technology for Multimedia Applications with Intel® OpenMP* Compiler”. In: *Proceedings of 17th International Parallel and Distributed Processing Symposium*, p. 36.
- Tsirogiannis, Dimitris, Sudipto Guha, and Nick Koudas (2009). “Improving the Performance of List Intersection”. In: *PVLDB* 2.1, pp. 838–849.
- Uhlmann, Jeffrey K. (1991). “Satisfying General Proximity/Similarity Queries with Metric Trees”. In: *Information Processing Letters* 40.4, pp. 175–179.
- Ullman, Jeffrey D. (1988). *Principles of Database and Knowledge-Base Systems, Volume I*. Vol. 14. Principles of computer science series. Computer Science Press. ISBN: 0-7167-8069-0.
- Valois, John D. (1995). “Lock-Free Linked Lists Using Compare-and-Swap”. In: *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pp. 214–222.
- Wang, Sheng, David Maier, and Beng Chin Ooi (2014). “Lightweight Indexing of Observational Data in Log-Structured Storage”. In: *PVLDB* 7.7, pp. 529–540.
- (2016). “Fast and Adaptive Indexing of Multi-Dimensional Observational Data”. In: *PVLDB* 9.14, pp. 1683–1694.
- Weber, Roger, Klemens Böhm, and Hans-Jörg Schek (2000). “Interactive-Time Similarity Search for Large Image Collections Using Parallel VA-Files”. In: *Proceedings of the 4th European Conference on Research and Advanced Technology for Digital Libraries*, pp. 83–92.
- Weber, Roger, Hans-Jörg Schek, and Stephen Blott (1998). “A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces”. In: *Proceedings of the 24th International Conference on Very Large Data Bases*, pp. 194–205.
- White, David A. and Ramesh Jain (1996). “Similarity Indexing with the SS-tree”. In: *Proceedings of the 12th IEEE International Conference on Data Engineering*, pp. 516–523.
- Willhalm, Thomas, Ismail Oukid, Ingo Müller, and Franz Faerber (2013). “Vectorizing Database Column Scans with Complex Predicates”. In: *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, pp. 1–12.
- Willhalm, Thomas, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner (2009). “SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units”. In: *PVLDB* 2.1, pp. 385–394.
- Xie, Zhongle, Qingchao Cai, H. V. Jagadish, Beng Chin Ooi, and Weng-Fai Wong (2016). “PI : a Parallel in-memory skip list based Index”. In: *CoRR* abs/1601.00159. URL: <http://arxiv.org/abs/1601.00159>.
- Yianilos, Peter N. (1993). “Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces”. In: *Proceedings of the Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms*, pp. 311–321.

Bibliography

- Yoo, Andy, Edmond Chow, Keith W. Henderson, Will McLendon III, Bruce Hendrickson, and Ümit V. Çatalyürek (2005). “A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L”. In: *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing*, p. 25.
- Yu, Jia and Mohamed Sarwat (2016). “Two Birds, One Stone: A Fast, yet Lightweight, Indexing Scheme for Modern Database Systems”. In: *PVLDB* 10.4, pp. 385–396.
- Zäschke, Tilmann, Christoph Zimmerli, and Moira C. Norrie (2014). “The PH-tree: a space-efficient storage structure and multi-dimensional index”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 397–408.
- Zeuch, Steffen, Johann-Christoph Freytag, and Frank Huber (2014). “Adapting Tree Structures for Processing with SIMD Instructions”. In: *Proceedings of the 17th International Conference on Extending Database Technology*, pp. 97–108.
- Zhang, Hao, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang (2015). “In-Memory Big Data Management and Processing: A Survey”. In: *IEEE Transactions on Knowledge and Data Engineering* 27.7, pp. 1920–1948.
- Zhang, Huanchen, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen (2016). “Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1567–1581.
- Zhou, Jingren and Kenneth A. Ross (2002). “Implementing database operations using SIMD instructions”. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 145–156.
- Zhou, Kun, Qiming Hou, Rui Wang, and Baining Guo (2008). “Real-time KD-tree construction on graphics hardware”. In: *ACM Transactions on Visualization and Computer Graphics* 27.5, 126:1–126:11.

List of Figures

2.1	A kd-tree (right) indexing six points from a two-dimensional data set (left). . . .	12
2.2	A PH-tree indexing the bitstrings of three two-dimensional points: (1, 8), (3, 8), (3, 10). The bitstrings are: (0001, 1000), (0011, 1000), (0011, 1010). The figure is taken from [Zäschke et al. 2014].	15
2.3	An R-tree (right) managing six points from a two-dimensional data set (left). . .	19
2.4	Memory hierarchy of Intel Skylake CPUs (Source: https://www.7-cpu.com/cpu/Skylake.html , Last access: August 29, 2018).	24
2.5	Evolution of Intel Xeon server CPUs (Source: https://en.wikipedia.org/wiki/List_of_Intel_Xeon_microprocessors and https://en.wikipedia.org/wiki/Xeon_Phi , Last accesses: February 25, 2019).	27
2.6	A screenshot taken from a genome browser, the integrative genomics viewer [Thorvaldsdóttir et al. 2013].	30
3.1	A balanced skip list that manages nine keys with two fast lanes. Each fast lane skips over two elements ($p = 1/2$).	38
3.2	A CSSL managing 32 keys with two fast lanes ($p = 1/2$).	40
3.3	The linearized fast lane array of a CSSL indexing all four-byte integers in $\{1, \dots, 64\}$ with two levels ($p = 1/2$). The fast lane array is aligned to 64-byte cache lines. .	42
3.4	Range query throughput of the competitors on 16 Million four-byte synthetic integer keys depending on the range size.	49
3.5	Range query throughput of the competitors on 256 Million four-byte synthetic integer keys depending on the range size.	51
3.6	Range query throughput on 13,571,394 four-byte integer keys obtained from real-world genomic variant data depending on the range size.	51
3.7	Lookup throughput on 16 Million four-byte synthetic integer keys.	52
3.8	Throughput of a mixed workload consisting of 500,000 lookups and 500,000 range queries when applied to 16 Million four-byte synthetic integer keys.	54
3.9	Space consumption for 16 Million four-byte integer keys.	55
4.1	Horizontal and vertical partitioning used to divide twenty five-dimensional tuples into five partitions.	60
4.2	Single-threaded execution of an MDRQ on a conventional R*-tree vs. parallel execution of an MDRQ on p instances of an R*-tree, where each instance manages the data of one partition (horizontal partitioning) and is searched with a distinct thread (p threads in total).	65

List of Figures

4.3	Throughput when executing MDRQ with an average selectivity of 0.1% on one Million twenty-dimensional tuples from UNIFORM depending on the used hardware features.	70
4.4	Throughput when executing MDRQ with an average selectivity of 0.4% (five dimensions) to 0.0002% (more than ten dimensions) on one Million tuples from UNIFORM using 24 software threads depending on the dimensionality.	71
4.5	Throughput when executing range queries on one Million five-dimensional tuples from UNIFORM using 24 software threads depending on the query selectivity.	72
4.6	Throughput when executing range queries with an average selectivity of 0.4% on five-dimensional tuples from UNIFORM using 24 software threads depending on the size of the data set.	73
4.7	Throughput when executing range queries with an average selectivity of 0.38% (one cluster) to 27.40% (20 clusters) on one Million five-dimensional tuples from CLUSTERED using 24 software threads depending on the number of clusters.	74
4.8	Throughput when executing range queries with an average selectivity of 11.12% on three-dimensional tuples from POWER using 24 software threads depending on the size of the data set.	75
4.9	Throughput of the contestants when executing the GMRQB on ten Million 19-dimensional tuples from GENOMIC using 24 software threads (the query templates are ordered by average selectivity, from low (left) to high (right)).	76
4.10	Throughput of the contestants when applying the mixed workload from GMRQB to ten Million 19-dimensional tuples from GENOMIC depending on the number of used software threads.	77
4.11	Memory usage of the competitors.	78
4.12	Throughput of the contestants on the desktop machine when applying the GMRQB to ten Million 19-dimensional tuples from GENOMIC using 12 software threads (the query templates are ordered by average selectivity, from low (left) to high (right)).	79
5.1	A BB-Tree of height $h = 2$ with an inner node fan out of $k = 3$ and a BB capacity of $b_{max} = 4$ managing $n = 36$ data objects of dimensionality $m = 3$; buckets 3 to 6 are omitted.	89
5.2	The linearized storage of the inner search tree.	90
5.3	When inserting a new data object (3 8 7) with TID 42 into the BB-Tree from Figure 5.1, the regular BB 2 morphs into a super BB that contains k regular nodes and partitions data objects according to dimension 2.	92
5.4	Parallel execution of an exemplary range query, defined by the lower boundary $[1, 0, 3]$ and the upper boundary $[3, 7, 6]$, using three threads.	97
5.5	Performance of BB-Trees with different BB capacities (B_{max}) when executing range queries with varying selectivities (1%, 10%, and 20%) on ten Million data objects from UNIFORM and CLUSTERED.	102
5.6	Performance of point queries on the different data sets depending on the number of data objects.	103

5.7	Performance of synthetic range queries on the different data sets depending on the number of data objects. Average query selectivities are as follows: UNIFORM: 0.4% ($\sigma = 0.9\%$), CLUSTERED: 19.8% ($\sigma = 19.7\%$), POWER: 12.6% ($\sigma = 13.1\%$), GENOMIC: 0.2% ($\sigma = 0.2\%$).	104
5.8	Performance of the Genomic Multidimensional Range Query Benchmark when executed on ten Million 19-dimensional data objects from GENOMIC. Query templates are ordered by selectivity, from low (left) to high (right).	105
5.9	Performance of range queries on ten Million data objects from UNIFORM (five dimensions) depending on query selectivity. We omitted the kd-tree.	105
5.10	Performance of point and range queries (average selectivity = 1%, $\sigma = 0.7\%$) when applied to ten Million uniformly distributed data objects depending on dimensionality.	107
5.11	Performance of synthetic range queries with a varying selectivity executed on ten Million five-dimensional data objects from UNIFORM depending on the number of distinct values per dimension.	108
5.12	Performance of synthetic range queries with an average selectivity of 0.00002% ($\sigma = 0.0\%$) executed on ten Million 50-dimensional data objects from UNIFORM depending on the number of distinct values per dimension; PH-tree is omitted.	109
5.13	Performance of insert operations on the different data sets depending on the number of data objects.	110
5.14	Performance of delete operations on the different data sets depending on the number of data objects.	111
5.15	Execution times of the mixed workload consisting of inserts, deletes, point and range queries executed in random order (bulk insert is not included). The PH-tree ran out of memory. The VA-file was excluded, because it does not support single-tuple inserts.	112
5.16	Performance of the Mixed Workload from GMRQB with an average selectivity of 1.58% ($\sigma = 3.58\%$) when applied to ten Million data objects from GENOMIC depending on the number of used software threads.	113
5.17	Space consumption of the competitors when storing ten Million data objects from the four data sets used in our evaluation.	114

List of Figures

List of Tables

2.1	The set of genomic variants used in this thesis consisting of ten Million tuples. . .	32
2.2	The query templates of the GMRQB.	34
3.1	Hardware performance counters per range query (10 % range size) on 16 Million four-byte integer keys.	50
3.2	Hardware performance counters per lookup on 16 Million four-byte integer keys.	53
4.1	Data sets used in our experiments.	68
4.2	Space consumption of the competitors.	77
5.1	Frequently used notations and input parameters.	86
5.2	Data sets used in our experiments.	100
5.3	Time spent on navigation of the IST and scanning of the BB when executing range queries of varying selectivity on ten Million five-dimensional objects from UNIFORM.	106
5.4	Performance counters per range query with a selectivity of 1% when applied to ten Million objects from UNIFORM (five dimensions).	106
5.5	(1) Total execution time of the bulk insert and (2) average, minimum and maximum execution time of the remaining queries, both read and write operations, of the mixed workload.	112

List of Tables

Selbständigkeitserklärung

Hiermit erkläre ich, die Dissertation selbstständig und nur unter Verwendung der angegebenen Hilfen und Hilfsmittel angefertigt zu haben. Ich habe mich nicht anderwärts um einen Doktorgrad in dem Promotionsfach beworben und besitze keinen entsprechenden Doktorgrad. Die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät, veröffentlicht im Amtlichen Mitteilungsblatt der Humboldt-Universität zu Berlin Nr. 42 am 11. Juli 2018, habe ich zur Kenntnis genommen.

Berlin, den 29. August 2018

Stefan Sprenger