Slalom: Coasting Through Raw Data via Adaptive Partitioning and Indexing

Matthaios Olma[†] Manos Karpathiotakis[†] Ioannis Alagiannis[‡] Manos Athanassoulis[⋆]
Anastasia Ailamaki[†]

[†]EPFL
{firstname.lastname}@epfl.ch

[‡]Microsoft
ioalagia@microsoft.com

[⋆]Harvard University
manos@seas.harvard.edu

ABSTRACT

The constant flux of data and queries alike has been pushing the boundaries of data analysis systems. The increasing size of raw data files has made data loading an expensive operation that delays the data-to-insight time. Hence, recent in-situ query processing systems operate directly over raw data, alleviating the loading cost. At the same time, analytical workloads have increasing number of queries. Typically, each query focuses on a constantly shifting – yet small – range. Minimizing the workload latency, now, requires the benefits of indexing in in-situ query processing.

In this paper, we present Slalom, an in-situ query engine that accommodates workload shifts by monitoring user access patterns. Slalom makes on-the-fly partitioning and indexing decisions, based on information collected by lightweight monitoring. Slalom has two key components: (i) an online partitioning and indexing scheme, and (ii) a partitioning and indexing tuner tailored for in-situ query engines. When compared to the state of the art, Slalom offers performance benefits by taking into account user query patterns to (a) *logically* partition raw data files and (b) build for each partition lightweight *partition-specific* indexes. Due to its lightweight and adaptive nature, Slalom achieves efficient accesses to raw data with minimal memory consumption. Our experimentation with both micro-benchmarks and real-life workloads shows that Slalom outperforms state-of-the-art in-situ engines ($3-10\times$), and achieves comparable query response times with fully indexed DBMS, offering much lower ($\sim 3\times$) cumulative query execution times for query workloads with increasing size and unpredictable access patterns.

1. INTRODUCTION

Nowadays, an increasing number of applications generate and collect massive amounts of data at a rapid pace. New research fields and applications (e.g., network monitoring, sensor data management, clinical studies, etc.) emerge and require broader data analysis functionality to rapidly gain deeper insights from the available data. In practice, analyzing such datasets becomes a costly task due to the data explosion of the last decade.

Big Data, Small Queries. The trend of exponential data growth due to intense data generation and data collection is expected to persist, however, recent studies of the data analysis workloads show that typically only a small subset of the data is relevant and ultimately used by analytical and/or exploratory workloads [1, 18]. In addition, modern businesses and scientific applications require interactive data access, which is characterized by *no or little a priori workload knowledge* and constant *workload shifting* both in terms of projected attributes and selected ranges of the dataset.

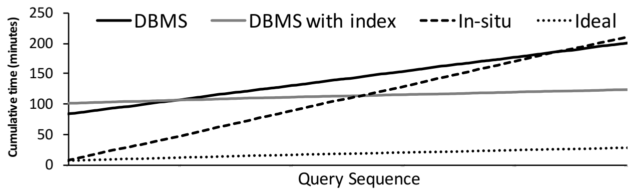


Figure 1: Ideally, in-situ data analysis should be able to retrieve only the relevant data for each query after the initial table scan (ideal - dotted line). In practice today, in-situ query processing avoids the costly phase of data loading (dashed line), however, as the number of the queries increases, the initial investment for full index on a DBMS pays off (the dashed line meets the grey line).

The Cost of Loading, Indexing, and Tuning. Traditional data management systems (DBMS) require the costly steps of *data loading*, *physical design decisions*, and then *index building* in order to offer interactive access over large datasets. Given the data sizes involved, any transformation, copying, and preparation steps over the data introduce substantial delays before the data can be queried, and provide useful insights [2, 5, 34]. The lack of a priori knowledge of the workload makes the physical design decisions virtually impossible because cost-based advisors rely heavily on past or sample workload knowledge [3, 17, 22, 29, 58]. The workload shifts observed in the interactive setting of exploratory workloads can nullify investments towards indexing and other auxiliary data structures (e.g., views), since frequently, they depend on the actual data values and the knowledge generated by the ongoing analysis.

Querying Raw Data Files Is Not Enough. Recent efforts opt to query directly raw files [2, 5, 13, 19, 30, 40] to reduce the data-to-insight cost. These *in-situ* systems avoid the costly initial data loading step, and allow the execution of declarative queries over external files without duplicating or "locking" data in a proprietary database format. Further, they concentrate on reducing costs associated with raw data accesses (e.g., parsing and converting data fields) [5, 19, 40]. Finally, although recent scientific data management approaches index raw data files using file-embedded indexes, they do it in a workload-oblivious manner, or requiring full a priori workload knowledge [13, 57]. Hence, they bring back in the raw data querying paradigm the cost of full index building, negating part of the benefits of avoiding data loading.

Proceedings of the VLDB Endowment, Vol. 10, No. 10
Copyright 2017 VLDB Endowment 2150-8097/17/06.

Figure 1 shows what the ideal in-situ query performance should be (dotted line). After the unavoidable first table scan, ideally, in-situ queries need to access only data relevant to the currently executed query. The figure also visualizes the benefits of state-of-the-art in-situ query processing when compared with a full DBMS. The y-axis shows the cumulative query latency, for an increasing number of queries with fixed selectivity on the x-axis. By avoiding the costly data loading phase the in-situ query execution system (dashed line) can start answering queries very quickly. On the other hand, when a DBMS makes an additional investment on full DBMS indexing (solid grey line), it initially increases significantly the data-to-query latency, however, it pays off as the number of queries issued over the same (raw) dataset increases. Eventually, the cumulative query latency for an in-situ approach becomes larger than the latency of a DBMS equipped with indexing. When operating over raw data, *ideally*, we want after the initial – unavoidable – table scan to collect enough metadata to allow future queries to access only the useful part of the dataset.

Adaptive Partitioning and Fine-Grained Indexing. We use the first table scan to generate partitioning and lightweight indexing hints, which are further refined by the data accesses of (only a few) subsequent queries. During this refinement process, the dataset is partially indexed in a dynamic fashion adapting to three key workload characteristics: (i) data distribution, (ii) query type (e.g., point query, range query), and (iii) projected attributes. Workload shifts lead to varying selected value ranges, selectivity, which areas of the dataset are relevant for a query, and projected attributes.

This paper proposes an online partitioning and indexing tuner for in-situ query processing which, when plugged into a raw data query engine, offers *fast queries over raw data files*. The tuner reduces data access cost by: (i) *logically partitioning* a raw dataset to virtually break it into more manageable chunks without physical restructuring, and (ii) choosing *appropriate indexing strategies over each logical partition* to provide efficient data access. The tuner adapts the partitioning and indexing scheme as a side-effect of executing the query workload. It continuously collects information regarding the values and access frequency of queried attributes at runtime. Based on this information, it uses a randomized online algorithm to define logical partitions. For each logical partition, it estimates the cost-benefit of building partition-local index structures considering both approximate (membership) indexing (i.e., Bloom filters and zonemaps) and full indexing (i.e., bitmaps and B$^+$-Trees). By allowing fine-grained indexing decisions our proposal defers the decision of the index shape to the level of each partition rather than the overall relation. This has two positive side-effects. First, there is no costly indexing investment that might be unnecessary. Second, any indexing effort is tailored to the needs of the data accesses on the corresponding range of the dataset.

Efficient In-Situ Query Processing With *Slalom*. We integrate our online partitioning and indexing tuner to an in-situ query processing prototype system, *Slalom*, which combines the tuner with a state-of-the-art raw data query executor. Slalom is further augmented with index structures and uses the tuner to decide how to partition and which index or indexes to build for each partition. In particular, Slalom logically splits raw data into partitions and selects which fine-grained, per-partition index to build based on how "hot" (i.e., frequently accessed) each partition is, and what types of queries target each partition. Slalom also populates binary caches (of data converted from raw to binary) to further boost performance. Slalom adapts to workload shifts by adjusting the current partitioning and indexing scheme using a randomized cost-based decision algorithm. Overall, the logical partitions and the indexes that Slalom builds over each partition provide performance enhancements without requiring expensive full data indexing nor data file re-organization, all while adapting to workload changes.

Contributions. This paper makes the following contributions:

- We present a logical partitioning scheme of raw data files that enables fine-grained indexing decisions at the level of each partition. As a result, lightweight per-partition indexing provides near-optimal raw data access.
- The lightweight partitioning allows our approach to maintain the benefits of past in-situ approaches. In addition, the granular way of indexing (i) brings the benefit of indexing to in-situ query processing, (ii) having low index building cost, and (iii) small memory footprint. These benefits are further pronounced as the partitioning and indexing decisions are refined on-the-fly using an online randomized algorithm.
- We integrate our partitioning and indexing tuner into our prototype state-of-the-art in-situ query engine *Slalom*. We use synthetic and real-life workloads to compare the query latency of Slalom, a traditional DBMS, a state-of-the-art in-situ query processing, and adaptive indexing (cracking). Our experiments show that, even when excluding the data loading cost, Slalom offers the fastest cumulative query latency. In particular, Slalom (a) outperforms state-of-the-art disk-based approaches by one order of magnitude, (b) state-of-the-art in-memory approaches by 3.7× (with 2.45× smaller memory footprint), and (c) adaptive indexing by 19% (with 1.93× smaller memory footprint).

To our knowledge, this is the first paper that proposes the use of a randomized online algorithm to select which workload-tailored, index structures should be built per partition of the data file. This approach offers constant, and more crucially minimal, decision time, while at the same time delivering optimal competitive ratio against the optimal offline algorithm.

2. RELATED WORK

Queries over Raw Data. Data loading is a large fraction of overall workload execution time in both the DBMS and Hadoop ecosystems [30]. NoDB [5] treats raw data files as native storage of the DBMS, and introduces auxiliary data structures (positional maps and caches) to reduce the expensive parsing and tokenization costs of raw data access. ViDa [38, 39, 40] introduces code-generated access paths and data pipeline to adapt the query engine to the underlying data formats and layouts, and to the incoming queries. Data Vaults [34, 36] and SDS/Q [13] perform analysis over scientific array-based file formats. SCANRAW [19] uses parallelism to mask the increased CPU processing costs associated with raw data accesses during in-situ data processing. In-situ DBMS approaches either rely on accessing the data via full table scans or require a priori workload knowledge and enough idle time to create the proper indexes. The mechanisms of Slalom are orthogonal to these systems, and can augment such systems by enabling data skipping and indexed accesses while constantly adapting its indexing and partitioning schemes to queries.

Hadoop-based systems such as Hive [55] can access raw data stored in HDFS. While such frameworks internally translate queries to MapReduce jobs, other systems follow a more traditional MPP architecture to offer SQL-on-Hadoop functionality [41, 43]. Hybrid approaches such as invisible loading [2] and Polybase [21] propose co-existence of a DBMS and a Hadoop cluster, transferring data between the two when needed. SQL Server PDW [24] and AsterixDB [6] propose indexes for data stored in HDFS and for external data in general. Similar to the case of DBMS-based approaches, the techniques of Slalom can also be applied in a Hadoop-based environment. In the case of PDW and AsterixDB, which build secondary indexes over HDFS files, the techniques used by Slalom can improve system scalability by reducing the size of the index and building memory efficient indexes per file-partition.

On the other side of raw data querying, Instant Loading [45] parallelizes the loading process for main-memory DBMS, offering bulk loading at near-memory-bandwidth speed. Similarly to Instant Loading, Slalom uses data parsing with hardware support for efficient raw data access. Instead of loading all data, however, Slalom exploits workload locality to adaptively create a fine-grained indexing scheme over raw data and gradually reduce I/O and access costs, all while operating under a modest memory budget.

Database Partitioning. A table can be physically subdivided into smaller disjoint sets of tuples (partitions), allowing tables to be stored, managed and accessed at a finer level of granularity [42].

Offline partitioning approaches [4, 27, 46, 58] present physical design tools that automatically select the proper partition configuration for a given workload to improve performance. Online partitioning [35] monitors and periodically adapts the database partitions to fit the observed workload. Furtado et al. [23] combine physical and virtual partitioning to fragment and dynamically tune partition sizes for flexibility in intra-query parallelism. Shinobi [56] clusters hot data in horizontal partitions which it then indexes, while Sun et al. [54] use a bottom-up clustering framework to offer an approximate solution for the partition identification problem.

Physical re-organization, however, is not suitable for data file repositories due to its high cost and the immutable nature of the files. Slalom presents a non-intrusive, flexible partitioning scheme that creates logical horizontal partitions by exploiting data skew. Additionally, Slalom continuously refines its partitions during query processing without requiring a priori workload knowledge.

Database Indexing. There is a vast collection of index structures with different capabilities, performance, and initialization/maintenance overheads [10, 11]. This paper uses representative index structures from the two categories (i) value-position and (ii) value-existence indexes, that offer good indexing for point and range queries. Value-position indexes include the B$^+$ Tree and hash indexes and their variations [9]. Common value-existence indexes are Bloom filters [14], Bitmap indexes [12, 52] , and Zonemaps [44]. They are lightweight and can provide the information whether a value is present in a given dataset. Value-existence indexes are frequently used in scientific workloads [20, 53, 57]. Slalom builds main-memory auxiliary structures (i) rapidly, (ii) with small footprint, and (iii) without a priori workload knowledge. That way it enables low data-to-insight latency, and does not penalize long running workloads, that indexing is typically useful.

Online Indexing. Physical design decisions made before workload execution can also be periodically re-evaluated. COLT [50] continuously monitors the workload and periodically creates new indexes and/or drops unused ones. COLT adds overhead on query execution because it obtains cost estimations from the optimizer at runtime. A “lighter” approach requiring fewer calls to the optimizer has also been proposed [16]. Slalom also focuses on the problem of selecting an effective set of indexes but Slalom builds indexes on partition granularity. Slalom also populates indexes during query execution in a pipelined fashion instead of triggering a standalone index building phase. Slalom aims to minimize the cost of index construction decisions and the complexity of the costing algorithm.

Adaptive Indexing. In order to avoid the full cost of indexing before workload execution, Adaptive Indexing incrementally refines indexes during query processing. In the context of in-memory column-stores Database Cracking approaches [25, 31, 32, 33, 48] create a duplicate of the indexed column and incrementally sorts it according to the incoming workload, thus reducing memory access. HAIL [49] proposes an adaptive indexing approach for MapReduce systems. ARF [7] is an adaptive value-existence index similar to Bloom filters, yet useful for range queries. Similarly to adaptive indexing,Slalom does not index data upfront and builds indexes during query processing and continuously adapts to the workload characteristics. However, contrary to adaptive indexing that duplicates the whole indexed attribute upfront, Slalom’s gradual index building allows its indexes to have small memory footprint by indexing both the targeted value ranges, and the targeted attributes.

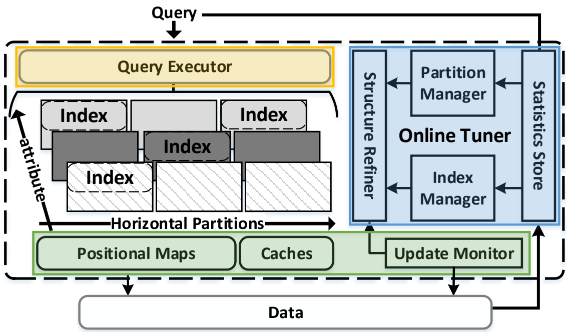


Figure 2: The architecture of Slalom.

3. THE SLALOM SYSTEM

Slalom uses adaptive partitioning and indexing to provide inexpensive index support for *in-situ* query processing while adapting to workload changes. Slalom accelerates query processing by skipping data and minimizes data access cost when this access is unavoidable. At the same time, it operates directly on the original data files without need for physical restructuring (i.e., copying, sorting).

Slalom incorporates state-of-the-art in-situ querying techniques and enhances them with logical partitioning and fine-grained indexing, thereby reducing the amounts of accessed data. To remain effective despite workload shifts, Slalom introduces an online partitioning and indexing tuner, which calibrates and refines logical partitions and secondary indexes based on data and query statistics. Slalom treats data files as relational tables to facilitate the processing of read-only and append-like workloads. The rest of this section focuses on the architecture and implementation of Slalom.

3.1 Architecture

Figure 2 presents the architecture of Slalom. Slalom combines an online partitioning and indexing tuner with a query executor featuring in-situ querying techniques. The core components of the tuner are the *Partition Manager*, which is responsible for creating logical partitions over the data files, and the *Index Manager*, which is responsible for creating and maintaining indexes over partitions. The tuner collects statistics regarding the data and query access patterns and stores them in the *Statistics Store*. Based on those statistics, the *Structure Refiner* evaluates the potential benefits of alternative configurations of partitions and indexes. Furthermore, Slalom uses *in-situ* querying techniques to access data. Specifically, Slalom uses auxiliary structures (i.e., positional maps and caches) which minimize raw data access cost. During query processing, the *Query Executor* utilizes the available data access paths and orchestrates the execution of the other components. Finally, the *Update Monitor* examines whether a data file has been modified and accordingly adjusts the data structures of Slalom.

Table 1: Statistics collected by Slalom per data file during query processing and used to decide (i) which logical partitions to create, and (ii) select the appropriate matching indexes.

Data (partition i)	Data (global)	Queries (partition i)
m_i: mean value	$Size_{page}$: page size	$C_{i_{build}}$: index build cost
min_i: min value	$Size_{file}$: file size	$C_{i_{fullscan}}$: full scan cost
max_i: max value		LA_i: #q since last access
dev_i: std. deviation		AF_i: part. access freq.
DV_i: #distinct values		sel_i: avg. sel. (0.0-1.0)

Slalom Scope. The techniques of Slalom are applicable to any tabular dataset. Specifically, the scan operator of Slalom can use a different specialized parser for each underlying data format. This work concentrates on queries over delimiter-separated textual CSV files, because CSV is the most popular structured textual file format. Still, the yellow- and blue-coded components of Figure 2 are applicable over binary files, which are the typical backend of databases and scientific applications.

Reducing Data Access Cost. Slalom launches queries directly over the original raw data files, without altering or duplicating the files by ingesting them in a DBMS, in order to avoid the initialization cost induced by loading and to allow for instant data access. Similarly to state-of-the-art in-situ query processing approaches [5, 19] Slalom mitigates the overheads of parsing and tokenizing textual data with positional maps (PM) [5] and partial data caching.

PMs are populated at query runtime and maintain *structural* information about the underlying textual file; they keep the positions of file attributes. This information is used during query processing to "jump" to the exact position of an attribute or as close as possible to an attribute, significantly reducing the cost of tokenizing and parsing when a tuple is accessed. Furthermore, Slalom builds binary caches of fields that are already converted to binary format to reduce parsing and data type conversion costs of future accesses.

Statistics Store. Slalom collects statistics during query execution and utilizes them to (i) detect workload shifts and (ii) enable the tuner to evaluate partitioning and index configurations. Table 1 summarizes the statistics about *Data* and *Queries* that Slalom gathers per data file. *Data statistics* are updated after every partitioning action and include the per-partition standard deviation (dev_i) of values, mean (m_i), max (max_i) and min (min_i) values. Additionally, Slalom keeps as global statistics the physical page size ($Size_{page}$) and file size ($Size_{file}$). Regarding *Query statistics*, Slalom maintains the number of queries since the last access (LA_i), the percentage of queries accessing each partition (access frequency AF_i), and the average query selectivity (sel_i). Finally, the full scan cost over a partition ($C_{i_{fullscan}}$) and the indexing cost for a partition ($C_{i_{build}}$) is calculated by considering the operator's data accesses.

Partition Manager. The Partition Manager recognizes patterns in the dataset and logically divides the file into contiguous non-overlapping chunks to enable fine-grained access and indexing. The Partition Manager specifies a logical partitioning scheme for each attribute in a relation. Each partition is internally represented by its starting and ending byte within the original file. The logical partitioning process starts the first time a query accesses an attribute. The Partition Manager triggers the Structure Refiner to iteratively fine-tune the partitioning scheme with every subsequent query. All partitions progressively reach a state in which there is no benefit from further partitioning. The efficiency of a partitioning scheme depends highly on the data distribution and the query workload. Therefore, the Partition Manager adjusts the partitioning scheme based on value cardinality (further explained in Section 4.1).

Index Manager. The Index Manager estimates the benefit of an index over a partition and suggests the most promising combination of indexes for a given attribute/partition. For every new index configuration, the Index Manager invokes the Structure Refiner to build the selected indexes during the execution of the next query. Every index corresponds to a specific data partition. Depending on the access pattern of an attribute and the query selectivity, a single partition may have multiple indexes. Slalom chooses indexes from two categories based on their capabilities: (i) *value-existence* indexes, which respond whether a value exists in a dataset, and (ii) *value-position* indexes, which return the positions of a value within the file. The online nature of Slalom imposes a significant challenge not only on which indexes to choose but also on when and how to build them with low cost. The Index Manager monitors previous queries to decide which indexes to build and when to build them; timing is based on an online randomized algorithm which considers (i) statistics on the cost of full scan ($C_{i_{fullscan}}$), (ii) statistics on the cost of building an index ($C_{i_{build}}$), and (iii) partition access frequency (AF_i), further explained in Section 4.2.

Update Monitor. The main focus of Slalom is read-only and append workloads. Still, to provide query result consistency, the Update Monitor checks the input files for both appends and in-place updates at real-time. Slalom enables append-like updates without disturbing query execution by dynamically adapting its auxiliary data structures. Specifically, Slalom creates a partition at the end of the file to accommodate the new data, and builds binary caches, PMs and indexes over them during the first post-update query. In-place updates require special care in terms of positional map and index maintenance because they can change the internal file structure. Slalom reacts to in-place updates during the first post-update query by identifying the updated partitions and the positional map, and recreating the other corresponding structures. More information about how Slalom tackles updates can be found in [8].

3.2 Implementation

We implement Slalom from scratch in C++. Slalom's query engine uses tuple-at-a-time execution based on the Volcano iterator model [26]. The rest of the components are implemented as modules of the query engine. Specifically, the *Partitioning* and *Indexing* managers as well as the *Structure Refiner* attach to the Query Executor. Furthermore, the *Statistics Store* runs as a daemon, gathering the data and query statistics and persisting them in a catalog.

Slalom reduces raw data access cost by using vectorized parsers, binary caches, and positional maps (PM). The CSV parser uses SIMD instructions; it consecutively scans a vector of 256 bytes from the input file and applies a mask over it using SIMD execution to identify delimiters. Slalom populates a PM for each CSV file accessed. To reduce memory footprint, the PM stores only delta distances for each tuple and field. Specifically, to denote the beginning of a tuple, the PM stores the offset from the preceding tuple. Furthermore, for each field within a tuple, the PM stores only the offset from the beginning of the tuple. The Partition Manager maintains a mapping between partitions and their corresponding PM portions.

Slalom populates binary caches at a partition granularity. When a query accesses an attribute for the first time, Slalom consults the positional map to identify the attribute's position, and then caches the newly converted values. To improve insertion efficiency, Slalom stores the converted fields of each tuple as a group of columns. If Slalom opts to convert an additional field during a subsequent query, it appends the converted value to the current column group.

Slalom also populates secondary indexes at a partition granularity; for each attribute, the indexes store its position in the file and its position in the binary cache (when applicable). Slalom uses a cache friendly in-memory B$^+$-Tree implementation. It uses nodes

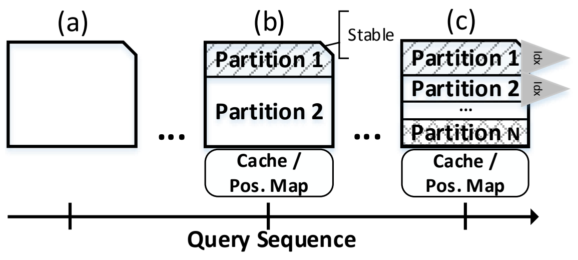


Figure 3: The Slalom execution visualized.

of 256 bytes that are kept 60% full. To minimize the size of inner nodes and make them fit in a processor cache line, the keys in the nodes are stored as deltas. Furthermore, to minimize tree depth, the B^+-Tree stores all appearances of a single value in one record.

The Structure Refiner monitors the construction of all auxiliary structures and is responsible for memory management. Slalom works within a memory area of pre-defined size. The indexes, PMs, and caches are placed in the memory area. However, maintaining caches of the entire file and all possible indexes is infeasible. Thus, the Structure Refiner dynamically decides, on a partition basis, which structure to drop so Slalom can operate under limited resources (details in Section 4.2).

3.3 Query Execution

Figure 3 presents an overview of a query sequence execution over a CSV file. During each query, Slalom analyzes its current state in combination with the workload statistics and updates its auxiliary structures. In the initial state (a), Slalom has no data or query workload information. The first query accesses the data file without any support from auxiliary structures; Slalom thus builds a PM, accesses the data requested, and places them in a cache. During each subsequent query, Slalom collects statistics regarding the data distribution of the accessed attributes and the average query selectivity to decide whether logical partitioning would benefit performance. If a partition has not reached its *stable* state (i.e., further splitting will not provide benefit), Slalom splits the partition into subsets as described in Section 4.1. In state (b), Slalom has already executed some queries and has built a binary cache and a PM on the accessed attributes. Slalom has decided to logically partition the file into two chunks, of which the first (partition 1) is declared to be in a *stable* state. Slalom checks stable partitions for the existence of indexes; if no index exists, Slalom uses the randomized algorithm described in Section 4.2 to decide whether to build one. In state (c), Slalom has executed more queries, and based on the query access pattern it decided index partition 1. In this state, partition 2 of state (b) has been further split into multiple partitions of which partition 2 was declared *stable* and an index was built on it.

4. CONTINUOUS PARTITION AND INDEX TUNING

Slalom provides performance enhancements without requiring expensive full data indexing nor data file re-organization, all while adapting to workload changes. Slalom uses an online partitioning and indexing tuner to minimize the accessed data by (i) logically partitioning the raw dataset, and (ii) choosing appropriate indexing strategies over each partition. To enable online adaptivity, all decisions that the tuner makes must have minimal computational overhead. The tuner employs a Partition Manager which makes all decision considering the partitioning strategy, and an Index Manager which makes all decisions considering indexing. This section presents the design of the Partition and Index Managers as well as the mathematical models they are based on.

4.1 Raw Data Partitioning

The optimal access path may vary across different parts of a dataset. For example, a filtering predicate may be highly selective in one part of a file, and thus benefit from index-based query evaluation, whereas another file part may be better accessed via a sequential scan. As such, any optimization applied on the entire file may be suboptimal for parts of the file. To this end, the Partition Manager of Slalom splits the original data into more manageable subsets; the minimum partition size is a physical disk page. the Partition Manager opts for horizontal logical partitioning because physical partitioning would require manipulating physical storage – a breaking point for many of the use cases that Slalom targets.

Why Logical Partitions. Slalom uses logical partitioning to virtually break a file into more manageable chunks without physical restructuring. The goal of logical partitioning is twofold: (i) enable partition filtering, i.e., try to group relevant data values together so that they can be skipped for some queries, and (ii) allow for more fine-grained index tuning. The efficiency of logical partitioning in terms of partition filtering depends mainly on data distribution and performs best with clustered or sorted data. Still, even in the worst case of uniformly distributed data, although few partitions will be skippable, the partitioning scheme facilitates fine-grained indexing. Instead of populating deep B+ Trees that cover the entire dataset, the B+ Trees of Slalom are smaller and target only "hot" subsets of the dataset. Thus, Slalom can operate under limited memory budget, has a minimal memory footprint, and provides rapid responses.

The Partition Manager performs partitioning as a by-product of query execution and chooses between two partitioning strategies depending on the cardinality of an attribute. For candidate key attributes, where all tuples have distinct values, the Partition Manager uses *query based partitioning*, whereas for other value distributions, it uses *homogeneous partitioning*. Ideally, what the Partition Manager aims for is creating partitions such that: (i) each partition contains uniformly distributed values, and (ii) partitions are pairwise disjoint (e.g., partition 1 has values *(12, 1, 8)* and partition 2 has values *(19, 13, 30)*). Uniformly distributed values within a partition enable efficient index access for all values in a partition and creating disjoint partitions improves partition skipping.

Homogeneous partitioning aims to create partitions with uniformly distributed values and maximize average selectivity within each partition. Increasing query selectivity over the partitions implies that for some queries, some of the newly created partitions will contain a high percentage of the final results, whereas other partitions will contain fewer or zero results and will be skippable. Computing the optimal set of contiguous uniformly distributed partitions has exponential complexity, thus is prohibitive for online execution. Instead, to minimize the overhead of partitioning, the Partition Manager iteratively splits a partition into multiple equi-size partitions. In every iteration, the tuner decides on (i) when to stop splitting and (ii) into how many subsets to split a given partition.

The Partition Manager splits incrementally a partition until it reaches a *stable* state (i.e., a state where the tuner estimates no more gains can be achieved from further splitting). After each partition split, the tuner relies on two conditions to decide whether a partition has reached a stable state. The tuner considers whether (i) the variance of values in the new partition as well as the excess kurtosis [47] of the value distribution have become smaller than the variance and kurtosis in the parent partition, and (ii) the number of distinct values has decreased. Specifically, as variance and excess kurtosis decrease, outliers are removed from the partition and the data distribution of the partition in question becomes more uniform. As the number of distinct values per partition iteratively decreases, the probability of partition disjointness increases. If any of these metrics increases or remains stable by partitioning, then the partition is declared stable. We use the combination of variance and excess kurtosis as a metric for uniformity, because their calculation has a constant complexity and can be performed in an incremental fashion during query execution. An alternative would be using a histogram or chi square estimators [47], but that would require building a histogram as well as an additional pass over the data.

The number of sub-partitions to which an existing partition is divided depends on the average selectivity of the past queries accessing the partition and the size of the partition in number of tuples. The goal of the tuner is to maximize selectivity in each new partition. We assume that the rows of the partition that have been part of query results within the partition are randomly distributed. We model the partitioning problem as randomly choosing tuples from the partition with the goal to have at least 50% of the new partitions exhibit higher selectivity than the original partition. The intuition is that by decreasing selectivity in a subset of partitions will enhance partition skipping in the rest. The model follows the hypergeometric distribution, whose CDF requires $O(N \cdot log(N))$ time to be computed [15]. As the sizes of partitions are large in comparison to selectivity, we can, without hurting generality, use the binomial approximation of the hypergeometric distribution. We assume that K out of the N tuples in a given partition qualify as query results ($sel = N/K$). We want to split the partition into m sub-partitions, each of size n, with our goal being that the ratio of qualifying tuples to total number of tuples in each new partition will be at least sel with probability 0.5. Thus, for every split, the Partition Manager uses the following formula to choose the number of partitions:

$$m = \frac{N \cdot (sel + \log_b(1 - sel))}{\log_b\left(\frac{\sqrt{2 \cdot \pi \cdot sel \cdot N}}{2}\right)} \qquad \text{where} \qquad b = \frac{e}{sel \cdot (1 - sel)}$$

Query based partitioning targets candidate keys, or attributes that are *implicitly clustered* (e.g., increasing timestamps). For such attributes, homogeneous partitioning will lead to increasingly small partitions as the number of distinct values and variance will be constantly decreasing with smaller partitions. Thus, the tuner decides upon a static number of partitions to split the file. Specifically, the number of partitions is decided based on the selectivity of the first range query using the same mechanism as in homogeneous partitioning. If the partition size is smaller than the physical disk page size, the tuner creates a partition per disk page. By choosing its partitioning approach based on the data distribution, Slalom improves the probability of data skipping and enables fine-grained indexing.

4.2 Adaptive Indexing in Slalom

The tuner of Slalom employs the Index Manager to couple logical partitions with appropriate indexes and thus decrease the amount of accessed data. The Index Manager uses *value-existence* and *value-position* indexes; it takes advantage of the capabilities of each category in order to reduce execution overhead and memory footprint. To achieve these goals, the Index Manager enables each partition to have multiple value-existence and value-position indexes.

Value-Existence Indexes. Value-existence indexes are the basis of partition-skipping for Slalom; once a partition has been set as stable, the Index Manager builds a value-existence index over it. Value-existence indexes allow Slalom to avoid accessing some partitions. The Index Manager uses Bloom filters, Bitmaps, and zone maps (min-max values) as value-existence indexes. Specifically, the Index Manager uses bitmaps only when indexing boolean attributes, because they require a larger memory budget than Bloom Filters for other data types. The Index Manager also uses zone maps on all partitions because they have small memory overhead and provide sufficient information for value-existence on partitions with small value variation. For all other data types, the Index Manager favors Bloom filters because of their high performance and small memory footprint. Specifically, the memory footprint of a Bloom filter has a constant factor, yet it also depends on the number of distinct values it will store and the required false positive probability. To overcome the inherent false positives that characterize Bloom filters, the Index Manager adjusts the Bloom filter's precision by calculating the number of distinct values to be indexed and the optimal number of bytes required to model them [14].

Value-Position Indexes. The Index Manager builds a value-position index (B$^+$-Tree) over a partition to offer fine-grained access to tuples. As value-position indexes are more expensive to construct compared to value-existence indexes, both in terms of memory and time, it is crucial for the index to pay off the building costs in future query performance. The usefulness and performance of an index depend highly on the type and selectivity of queries and the distribution of values in the dataset. Thus, for workloads of shifting locality, the core challenge is deciding *when* to build an index.

When to Build a Value-Position Index. The Index Manager builds a value position index over a partition if it estimates that there will be enough subsequent queries accessing that partition to pay off the investment (in execution time). As the tuner is unaware of the future workload trends, decisions for building indexes are based on the past query access patterns. To make these decisions, the Index Manager uses an online randomized algorithm which considers the cost of indexing the partition (Ci_{build}), the cost of full partition scan ($Ci_{fullscan}$), and the access frequency on the partition (AF_i). These values depend on the data type and the size of the partition, so they are updated accordingly in case of a partition split or an append to the file. The tuner stores the average cost of an access to a file tuple as well as the average cost of an insertion to every index for all data types, and uses these metrics to calculate the cost of accessing and building an index over a partition. In addition, the tuner calculates the cost of an index scan ($Ci_{indexscan}$) based on the cost of a full partition scan and the average selectivity. For each future access to the partition, the Index Manager uses these statistics to generate online a probability estimate which calculates whether the index will reduce execution time for the rest of the workload. Given this probability, the Index Manager decides whether to build the index.

The Index Manager calculates the index building probability using a randomized algorithm based on the randomized solution of the snoopy caching problem [37]. In the snoopy caching problem, two or more caches share the same memory space which is partitioned into blocks. Each cache writes and reads from the same memory space. When a cache writes to a block, caches that share the block spend 1 bus cycle to get updated. These caches can invalidate the block to avoid the cost of updating. When a cache decides to invalidate a block which ends up required shortly after, there is a penalty of p cycles. The optimization problem lies in finding when a cache should invalidate and when to update the block. The solution to the index building problem in this work involves a similar decision. The indexing mechanism of the tuner of Slalom decides whether to pay an additional cost per query ("updating a block") or invest in building an index, hoping that the investment will be covered by future requests ("invalidating a block").

The performance measure of randomized algorithms is the *competitive* ratio (*CR*): the ratio between the expected cost incurred when the online algorithm is used and that of an optimal offline algorithm that we assume has full knowledge of the future. The randomized algorithm of the tuner guarantees optimal CR ($\frac{e}{e-1}$). The tuner uses a randomized algorithm in order to avoid the high complexity of what-if analysis [50] and to improve the competitive ratio offered by the deterministic solutions [16].

Cost Model. Assume query workload W. At a given query of the workload, a partition is in one of two states: it either has an index or not. The state is characterized by the pair (C_{build}, C_{use}) where C_{build} is the cost to enter the state (e.g., build the index) and C_{use} the cost to use the state (e.g., use the index). Initially the system is in state with no index (i.e., full scan) $(C_{build,fs}, C_{use,fs})$ where $C_{build,fs} = 0$. In the second state $(C_{build,idx}, C_{use,idx})$, the system has an index. We assume that the relation between the costs for the two states is $C_{build,idx} > C_{build,fs}$ and $C_{use,idx} < C_{use,fs}$ and $C_{build,idx} > C_{use,fs}$.

Given a partition i, the index building cost over that partition ($C_{i_{build}}$), the full partition scan cost ($C_{i_{fullscan}}$), the index partition scan cost ($C_{i_{indexscan}}$) and a sequence of queries $Q : [q_1, \ldots, q_T]$ accessing the partition. Assume that q_T is the last query that accesses the partition (*and is not known*). At the arrival time of $q_k, k < T$, we want to decide whether the Index Manager should build the index or perform full scan over the partition to answer the query.

To make the decision we need a probability estimate p_i for building the index at moment i based on the costs of building the index or not. In order to calculate p_i we initially define the overall expected execution cost of the randomized algorithm that depends on the probability p_i. The expected cost E comprises three parts:

i. the cost of building the index, which corresponds to the case where the building of the index will take place at time i. Index construction takes place as a by-product of query execution and includes the cost of the current query.

ii. the cost of using the index, which corresponds to the case where the index has already been built.

iii. the cost of queries doing full partition scan, which corresponds to the case for which the index will not be built.

$$E = \sum_{i=1}^{T} \left(p_i \cdot C_{build,idx} + \sum_{j=1}^{i-1} p_j \cdot C_{use,idx} + (1 - \sum_{j=1}^{i-1} p_j) \cdot C_{use,fs} \right)$$

The optimal offline algorithm which has full knowledge of the future, including q_T when the partition stops being accessed, considers that if the number of queries executed is insufficient to cover the expense of building the index, then the optimal approach is to execute only full partition scans. On the other hand, if the number of queries executed is sufficient for the execution time savings to cover the cost of index building, the algorithm invests into building the index at the time of the first query. Specifically, the expected cost formula of the optimal offline algorithm is the following:

$$\begin{cases} T \cdot C_{use,fs}, & \text{if } T \cdot C_{use,fs} \leq C_{build,idx} + T \cdot C_{use,idx} \\ C_{build,idx} + T \cdot C_{use,idx}, & \text{otherwise} \end{cases}$$

The randomized algorithm will be in the best case as efficient as the optimal. Thus, the tuner chooses p_i such that it minimizes a:

$$E \leq (1+a) \cdot T \cdot C_{use,fs}$$
$$E \leq (1+a) \cdot (C_{build,idx} + T \cdot C_{use,idx})$$

By exchanging the inequalities to equalities and solving the linear system for minimizing a we get p_i. Based on the probability p_i the tuner decides whether to build the index.

Eviction Policy. The tuner works within a predefined memory budget to minimize memory overhead. If the memory budget is fully consumed and the Index Manager attempts to build a new index, then it defers index construction for the next query and searches indexes to drop to make the necessary space available. The Index Manager keeps all value-existence indexes once built, because their size is minimal and they are the basis of partition skipping. Furthermore, the Index Manager prioritizes binary caches over indexes, because (i) using a cache improves the performance of all queries accessing a partition, and (ii) accessing the raw data file is typically more expensive than rebuilding an index for large partitions. Deciding which indexes from which partitions to drop is based on index size ($Size_{index_i}$), number of queries since last access (LA_i), and average selectivity (sel_i) in a partition. To compute the set of indexes to drop, the Index Manager uses a greedy algorithm which gathers the least accessed indexes with cumulative size ($\sum_i Size_{index_i}$) equal to the size of the new index.

5. EXPERIMENTAL EVALUATION

In this section, we present an analysis of Slalom. We analyze its partitioning and indexing algorithm, and compare it against state-of-the-art systems over both synthetic and real life workloads.

Methodology. We compare Slalom against DBMS-X, a commercial state-of-the-art in-memory DBMS that stores records in a row oriented manner and the open-source DBMS PostgreSQL (version 9.3). We use DBMS-X and PostgreSQL with two different configurations: (i) Fully-loaded tables and (ii) Fully-loaded, indexed tables. We also compare Slalom with the in-situ DBMS PostgresRaw [5]. PostgresRaw is an implementation of NoDB [5] over PostgreSQL; PostgresRaw avoids data loading and executes queries by performing full scans over CSV files. In addition, PostgresRaw builds positional maps on-the-fly to reduce parsing and tokenization costs. Besides positional maps, PostgresRaw uses caching structures to hold previously accessed data in a binary format. Furthermore, to compare Slalom with other adaptive indexing techniques we integrate into Slalom two variations of database cracking: (i) standard cracking [31] and (ii) the MDD1R variant of stochastic cracking [28]. We chose MDD1R as it showed the best overall performance in [51]. We integrated the cracking techniques by disabling the Slalom tuner and setting Cracking as the sole access path. Thus, Slalom and Cracking use the same execution engine and have the same data access overheads.

Slalom's query executor pushes predicate evaluation down to the access path operators for early tuple filtering and results are pipelined to the other operators of a query (e.g., joins). Thus, in our analysis, we focus on scan intensive queries. We use select - project - aggregate queries to minimize the number of tuples returned and avoid any overhead from the result tuple output that might affect the measured times. Unless otherwise stated, the queries are of the following template ($OP : \{<, >, =\}$):

```
SELECT agg(A), agg(B), ..., agg(N) FROM R
WHERE A OP X (AND A OP Y)
```

Experimental Setup. The experiments are conducted in a Sandy Bridge server with a dual socket Intel(R) Xeon(R) CPU E5-2660 (8 cores per socket @ 2.20 Ghz), equipped with 64 KB L1 cache and 256 KB L2 cache per core, 20 MB L3 cache shared, and 128 GB RAM running Red Hat Enterprise Linux 6.5 (Santiago - 64 bit) with kernel version 2.6.32. The server is equipped with a RAID-0 of 7 250GB 7500 RPM SATA disks.

5.1 Adapting to Workload Shifts

Slalom adapts efficiently to workload shifts despite (i) changes in data distribution, (ii) changes in query selectivity, and (iii) changes in query locality - both vertical (i.e., different attributes) and horizontal (i.e., different records). We demonstrate the adaptivity experimentally by executing a dynamic workload with varying selectivities and access patterns over a synthetic dataset.

Methodology. To emulate the worst possible scenario for Slalom, we use a relation of 640 million tuples (59GB), where each tuple comprises 25 unsigned integer attributes with uniformly distributed

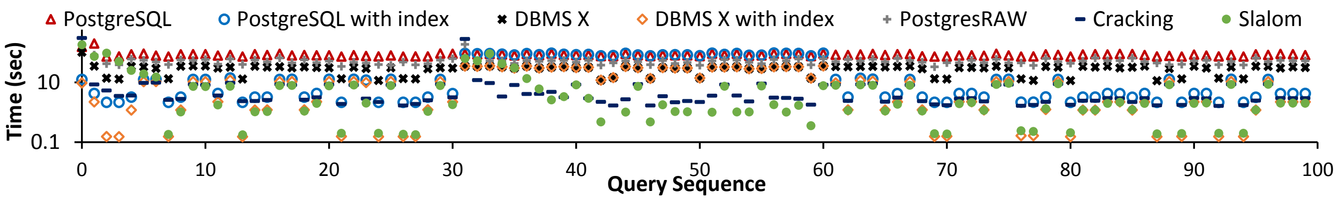


Figure 4: Sequence of 100 queries. Slalom dynamically refines its indexes to reach the performance of an index over loaded data.

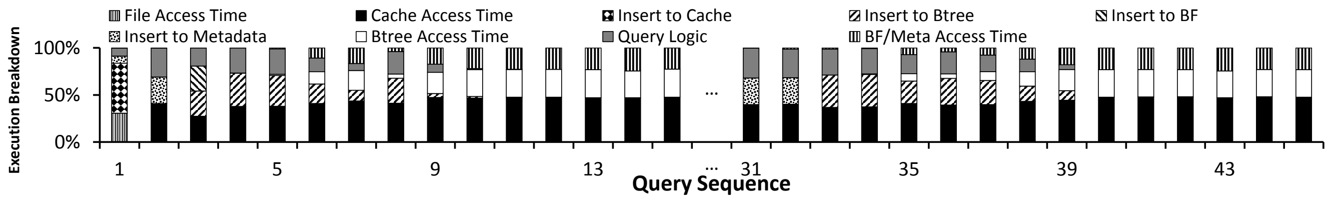


Figure 5: A breakdown of the operations taking place for Slalom during the execution of a subset of the 1000 point query sequence.

values ranging from 0 to 1000. Slalom is unable to find a value clustering in the file because all values are uniformly distributed, thus Slalom applies homogeneous partitioning. Slalom, Cracking, and PostgresRaw operate over the CSV data representation, whereas PostgreSQL and DBMS-X load the raw data prior to querying. In this experiment we limit the index memory budget for Slalom to 5GB and the cache budget to 10GB. All other systems are free to use all available memory. Specifically, for this experiment DBMS-X required 98GB of RAM to load and fully build the index.

We execute a sequence of 1000 point and range select-project-aggregation queries following the template from Section 5. The selection value is randomly selected from the domain of the predicate attribute. Point query selectivity is 0.1% and range query selectivity varies from 0.5% to 5%. To emulate workload shifts and examine system adaptivity, in every 100 queries, queries 1-30 and 61-100 use a predicate on the first attribute of the relation and queries 31-60 use a predicate on the second attribute.

The indexed variations of PostgreSQL and DBMS-X build a clustered index only on the first attribute. It is possible to build indexes on more columns for PostgreSQL and DBMS-X, however it requires additional resources and would increase data-to-query time. In addition, choosing which attributes to index requires a priori knowledge of the query workload, which is unavailable in the dynamic scenarios that Slalom considers. Indicatively, building an secondary index on a column for PostgreSQL for our experiment takes ∼25 minutes. Thus, by the time PostgreSQL finishes indexing, Slalom will have finished executing the workload (Figure 6).

Slalom Convergence. Figure 4 presents the response time of each query of the workload for the different system configurations. For clarity we present the results for the first 100 queries. To emulate the state of DBMS systems immediately after loading, all systems run from a hot state where data is resting in the OS caches. Figure 4 plots only query execution time and does not show data loading or index building for PostgreSQL and DBMS-X.

The runtime for the first query of Slalom is 20× slower than its average query time, because during that query it builds a positional map and a binary cache. In subsequent queries (queries 2-7) Slalom iteratively partitions the dataset and builds B$^+$-Trees. After the initial set of queries (queries 1-6), Slalom has comparable performance to the one of PostgreSQL over fully indexed data. During the 3rd query, multiple partitions stabilize simultaneously, thus Slalom builds many B$^+$-Tree and Bloom Filter indexes, adding considerable overhead. When Slalom converges to its final state, its performance is comparable to indexed DBMS-X. When the queried attribute changes (query 31), Slalom starts partitioning and building indexes on the new attribute. After query 60, when the workload filters data based on the first attribute again, since the partitioning has already stabilized, Slalom re-uses the pre-existing indexes.

PostgreSQL with no indexes demonstrates a stable execution time as it has to scan all data pages of the loaded database regardless of the result size. Due to the queries being very selective, when an index is available for PostgreSQL the response times are ∼9× lower when queries touch the indexed attribute. DBMS-X keeps all data in memory and uses memory-friendly data structures, so it performs on average 3× better than PostgreSQL. The difference in performance varies with query selectivity. In highly selective queries, DBMS-X is more efficient in data access whereas for less selective queries the performance gap is smaller. Furthermore, for very selective queries, indexed DBMS-X is more efficient than Slalom as its single B$^+$-Tree traverses very few results nodes.

During query 1, PostgresRaw builds auxiliary structures (cache, positional map) and takes 3× more time (180 sec) than its average query run time. PostgresRaw becomes faster than the unindexed PostgreSQL variation because its scan operators use vector-based (SIMD) instructions and exploit compact caching structures.

Similarly, during query 1, Cracking builds a binary cache and populates the cracker column it uses for incremental indexing. The runtime of its first query is 4× slower than the average query time for PostgreSQL without indexes. When it touches a different attribute (query 31) it also populates a cracker column for the second attribute. Despite the high initialization cost, Cracking converges efficiently, and reaches its final response time after the 4th query. The randomness in the workload benefits Cracking as it splits the domain into increasingly smaller pieces. After converging, Cracking performance is comparable to the PostgreSQL with index. Slalom requires more queries to converge than Cracking. However, after it converges, Slalom is ∼2× faster than Cracking. This difference stems from Cracking execution overheads. Cracking sorts the resulting tuples based on their memory location and enforces sequential memory access. This sorting operation adds an overhead, especially for less selective queries.

Execution Breakdown. Slalom aims to build efficient access paths with minimal overhead. Figure 5 presents the breakdown of query execution for the same experiment as before. For clarity, we present only queries Q1-15 and Q31-45 as Q16-30 show the same pattern as Q11-15. Queries Q1-15 have a predicate on the first attribute and queries Q31-45 have a predicate on the second attribute.

During the first query, Slalom scans through the original file and creates the cache. During Q2 and Q3 Slalom is actively partitioning the file and collects data statistics (i.e., distinct value counts) per partition; Slalom bases the further partitioning and indexing decisions on these statistics. Statistics gathering cost is represented

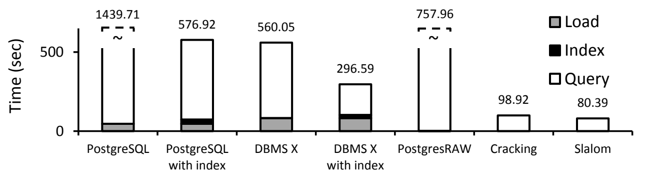


Figure 6: Sequence of 1000 queries. Slalom does not incur loading cost and dynamically builds indexes.

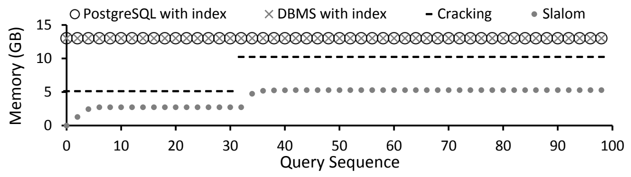


Figure 7: Memory consumption of Slalom vs. a single fully-built B+ Tree for PostgreSQL and DBMS-X. Slalom uses less memory because its indexes only target specific areas of a raw file.

in Figure 5 as "Insert to Metadata". During queries Q2 and Q3, as the partitioning scheme stabilizes, Slalom builds Bloom filters and B$^+$-Trees. Q3 is the last query executed using a full partition scan, and since it also incurs the cost of index construction there is a local peak in execution time. During Q4 through Q8, Slalom increasingly improves performance by building new indexes. After Q31, the queries use the second attribute of the relation in the predicate, thus Slalom repeats the process of partitioning and index construction. In total, even after workload shifts, Slalom converges into using index-based access paths over converted binary data.

Full Workload: From Raw Data to Results. Figure 6 presents the full workload of 1000 queries, this time starting with cold OS caches and no loaded data to include the cost of the first access to raw data files for all systems. We plot the aggregate execution time for all approaches described earlier, including the loading and indexing costs for PostgreSQL and DBMS-X.

PostgresRaw, Slalom and Cracking incur no loading and indexing cost, and start answering queries before the other DBMS load data and before the indexed approaches finish index building. Unindexed PostgreSQL incurs data loading cost as well as a total query aggregate greater than PostgresRaw. Indexed PostgreSQL incurs both indexing and data loading cost, and due to some queries touching a non-indexed attribute, its aggregate query time is greater than the one of Slalom. Unindexed DBMS-X incurs loading cost; however, thanks to its main memory-friendly data structures and execution engine, it is faster than the disk-based engine of PostgreSQL.

After adaptively building the necessary indexes, Slalom has comparable performance with a conventional DBMS which uses indexes. Cracking converges quickly and adapts to the workload efficiently. However, creating the cracker columns incurs a significant cost. Overall, Cracking and Slalom offer comparable raw-data-to-results response time for this workload while, Slalom requires 0.5× memory. We compare in detail Cracking and Slalom in Section 5.3.

Memory Consumption. Figure 7 plots the memory consumption of (i) the fully built indexes used for DBMS-X and PostgreSQL, (ii) the cracker columns for Cracking, and (iii) the indexes of Slalom. Figure 7 excludes the size of the caches used by Slalom and Cracking or the space required by DBMS-X after loading. The traditional DBMS require significantly more space for their indexes. Orthogonally to the index memory budget, DBMS-X required 98GB of memory in total, whereas the cache of Slalom required 9.7GB. Cracking builds its cracker columns immediately when accessing a new attribute. The cracker column requires storing the original column values as well as pointers to the data, thus it has a large

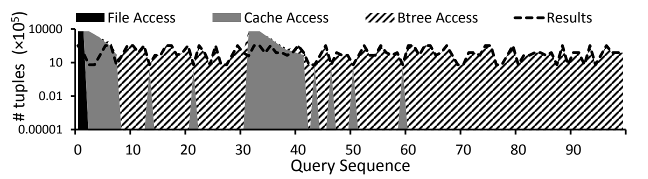


Figure 8: Sequence 100 queries. Number of accessed tuples using different access paths. Slalom uses indexes and data skipping to reduce data access.

memory footprint even for low value cardinality. Regarding the indexes of Slalom, when the focus shifts to another filtering attribute (Q31), Slalom increases its memory consumption, as during Q31-34 it creates logical partitions and builds Bloom filters and B$^+$-Tree indexes on the newly accessed attribute. By building and keeping only the necessary indexes for a query sequence, Slalom strikes a balance between query performance and memory utilization.

Minimizing Data Access. The performance gains of Slalom are a combination of data skipping based on partitioning, value-existence indexes, and value-position indexes, all of which minimize the number of tuples Slalom has to access. Figure 8 presents the number of tuples that Slalom accesses for each query in this experiment. We observe that as the partitioning and indexing schemes of Slalom converge, the number of excess tuples accessed is reduced. Since the attribute participating in the filtering predicate of queries Q31-60 has been cached, Slalom accesses the raw data file only during the first query. Slalom serves the rest of the queries utilizing only the binary cache and indexes. For the majority of queries, Slalom responds using an index scan. However there are queries where it responds using a combination of partition scan and index scan.

Figure 9 presents how the minimized data access translates to reduced response time and the efficiency of data skipping and indexing for different data distribution and different query types. Specifically, it presents the effect of Zone Maps, Bloom filters and B$^+$-Trees on query performance for point queries and range queries with 5% selectivity over Uniform and Clustered datasets. The Clustered dataset contains mutually disjointed partitions (i.e., subsets of the file contain values which do not appear in the rest of the file). The workload used is the same used for Figure 4. Zone maps are used for both range and point queries and are most effective when used over clustered data. Specifically, they offer a ∼9× better performance than full cache scan. Bloom filters are useful only for point queries. As the datasets have values in the domain [1,1000], point queries have low selectivity making Bloom filters ineffective. Finally, B$^+$-Trees improve performance for both range and point queries. The effect of B$^+$-Tree is seen mostly for uniform data where partition skipping is less effective. Slalom stores all indexes in-memory, thus by skipping a partition, Slalom avoids full access of the partition and reduces memory access or disk I/O if the partition is cached or not respectively.

Summary. We compare Slalom against (i) a state-of-the-art in-situ querying approach, (ii) a state-of-the-art adaptive indexing technique, (iii) a traditional DBMS, and (iv) a state-of-the-art in-memory DBMS. Slalom gracefully adapts to workload shifts using an adaptive algorithm with negligible execution overhead. Slalom offers performance comparable with a DBMS which uses indexes, while also being more conservative in memory space utilization.

5.2 Working Under Memory Constraints

As described in Section 4.2, Slalom efficiently uses the available memory budget to keep the most beneficial auxiliary structures. We show this experimentally by executing the same workload under various memory utilization constraints. We run the 20 first queries

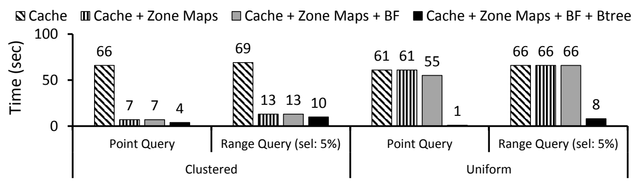


Figure 9: The effect of different indexes on point and range queries over uniform and clustered datasets.

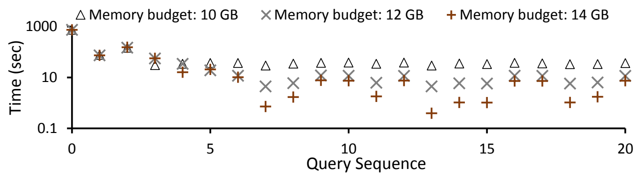


Figure 10: Slalom performance using different memory budgets. Slalom performance varies with alloted memory.

– a mix of point and range queries. We consider three memory budget configurations with 10GB, 12GB and 14GB of available memory, respectively. The budget includes both indexes and caches.

Figure 10 presents the query execution times for the workload given the three different memory budgets. The three memory configurations build a binary cache and create the same logical partitioning. Slalom requires 13.5GB in total for this experiment; given an 14GB memory budget, it can build all necessary indexes, leading to the best performance for the workload. For the 10GB and 12GB memory budgets, there is insufficient space to build all necessary indexes, thus these configurations experience a performance drop. We observe that the configurations with 10GB and 12GB memory budgets outperform the configuration with 14GB of memory budget for individual queries (i.e., Q3 and Q5). The reason is that the memory-limited configurations build fewer B$^+$-Trees during these queries than the configuration with 14GB of available memory. However, future queries can benefit from the additional B$^+$-Trees, amortizing the extra overhead over a sequence of queries.

Figure 11 presents the breakdown of memory allocation for the same query sequence when Slalom is given a 12GB memory budget. We consider the space required for storing caches, B$^+$-Trees and Bloom filters. The footprint of the statistics and metadata Slalom collects for the cost model and zone maps is negligible, thus we exclude them from the breakdown. Slalom initially builds the binary cache, and logically partitions the data until some partitions become stable (Q1, Q2). At queries Q3, Q4 and Q5 Slalom starts building B$^+$-Trees, and it converges to a stable state at query Q7 where all required indexes are built. Thus, from Q7-Q10 Slalom stabilizes performance. Overall, this experiment shows that Slalom can operate under limited memory budget gracefully managing the available resources to improve query execution performance.

5.3 Adaptivity Efficiency

Slalom adapts to query workloads as efficiently as state-of-the-art adaptive indexing techniques while working with less memory. Furthermore, it exploits any potential data clustering to further improve its performance. We demonstrate this by executing a variety of workloads. We use datasets of 480M tuples (55GB on disk); each tuple comprises 25 unsigned integer attributes whose values belong to the domain $[1, 10000]$. Queries in all workloads have equal selectivity to alleviate the noise from data access; all queries have 0.1% selectivity, i.e., select 10 consecutive values.

Methodology. Motivated by related work [51], we compare Slalom against Cracking and Stochastic Cracking in three cases.

Random workload over Uniform dataset. We execute a sequence

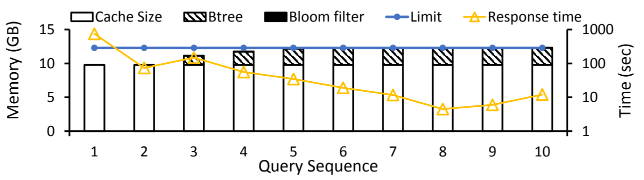


Figure 11: Slalom memory allocation (12 GB memory budget).

of range queries which access random ranges throughout the domain to emulate the best case scenario for cracking. As subsequent queries filter on random values and the data is uniformly distributed in the file, Cracking converges and minimizes data access.

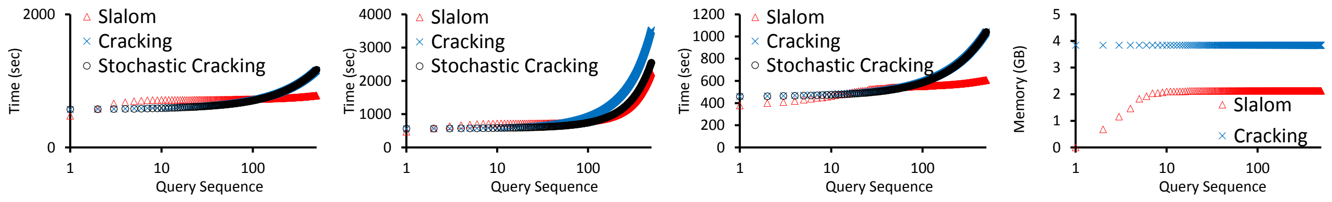
"Zoom In Alternate" over Uniform dataset. To emulate the effect of patterned accesses we execute a sequence of queries that access either part of the domain in alternate i.e., 1st query: [1,10], 2nd query: [9991,10000], 3rd query: [11,20], etc. This access pattern is one of the scenarios where the original cracking algorithm underperforms [28]. Splits are only query-driven, and every query splits data into a small piece and the rest of the file. Thus, the improvements in performance with subsequent queries are minimal. Stochastic cracking alleviates the effect of patterned accesses by splitting in more pieces apart from the ones based on queries.

Random workload over Clustered dataset. This setup examines how adaptive indexing techniques perform on datasets where certain data values are clustered together e.g., data clustered on timestamp or sorted data. The clustered dataset we use in the experiment contains mutually disjoint partitions, i.e., subsets of the file contain specific values which do not appear in the rest of the file.

Figure 12a demonstrates the cumulative execution time for Cracking, Stochastic Cracking and Slalom for the random workload over uniform data. All approaches start from a cold state, thus during the first query they parse the raw data file and build a binary cache. Stochastic Cracking and Cracking incur an additional cost of cracker column initialization during the first query, but reduce execution time with every subsequent query. During the first three queries, Slalom creates its partitions; during the following 6 queries, Slalom builds the required indexes, and finally converges to a stable state at query 10. Due to its fine-grained indexing and local memory accesses, Slalom provides $\sim 8\times$ lower response time than cracking and their cumulative execution time is equalized during query 113. Furthermore, Figure 12d demonstrates the memory consumption of the cracking approaches and Slalom for the same experiment. The cracking approaches have the same memory footprint; they both duplicate the full indexed column along with pointers to the original data. On the other hand, the cache-conscious B$^+$-Trees of Slalom stores only the distinct values along with the positions of each value, thus reducing the memory footprint. In addition, Slalom allocates space for its indexes gradually, allowing it to offer efficient query execution even with limited resources.

Figure 12b shows the cumulative execution time for Cracking, Stochastic Cracking, and Slalom for the "Zoom In Alternate" workload over uniform data. Cracking needs more queries to converge to its final state as it is cracking only based on query-driven values. Stochastic cracking converges faster because it cracks based on more values except the ones found in queries. Slalom uses a combination of data and query driven optimizations. Slalom requires an increased investment during the initial queries to create its partitioning scheme and index the partitions, but ends up providing $7\times$ lower response time, and equalizes cumulative execution time with Cracking at query 53 and Stochastic Cracking at query 128.

Figure 12c presents the cumulative execution time of Cracking, Stochastic Cracking and Slalom for the random workload over implicitly clustered data. In this situation, Slalom exploits the clustering of the underlying data early on (from the second query) and



(a) Random/Uniform data **(b)** Zoom In Alt./Uniform data **(c)** Random/Clustered data **(d)** Memory on Random/Uniform data

Figure 12: Cracking techniques converge more efficiently but Slalom takes advantage of data distribution.

Table 2: Cost of each phase of a smart-meter workload.

System	Loading	Index Build	Queries	Total
Slalom	0 sec	0 sec	4301 sec	4301 sec
Cracking	0 sec	0 sec	6370 sec	6370 sec
PostgresRaw	0 sec	0 sec	10077 sec	10077 sec
PostgresSQL (with index)	2559 sec	1449 sec	9058 sec	13066 sec
PostgreSQL (no index)	2559 sec	0 sec	15379 sec	17938 sec
DBMS-X (with index)	6540 sec	1207 sec	3881 sec	11628 sec
DBMS-X (no index)	6540 sec	0 sec	5243 sec	11783 sec

skips the majority of data. For the accessed partitions, Slalom builds indexes to further reduce access time. Similarly to Figure 12a, the Cracking approaches crack only based on the queries and are agnostic to the physical organization of the dataset.

Summary. Slalom converges comparably to the best Cracking variation when querying uniform data over both random and "Zoom In Alternate" workloads. Furthermore, when Slalom operates over clustered data, it exploits the physical data organization and provides minimal data-to-query time. Finally, as Slalom builds indexes gradually and judiciously, it requires less memory than the cracking approaches, and it can operate under a strict memory budget.

5.4 Slalom Over Real Data

In this experiment, we demonstrate how Slalom serves a real-life workload. We use a smart home dataset (SHD) taken from an electricity monitoring company. The dataset contains timestamped information about sensor measurements such as energy consumption and temperature, as well as a sensor id for geographical tracking. The timestamps are in increasing order. The total size of the dataset is 55 GB in CSV format. We run a typical workload of an SHD analytics application. Initially, we ask a sequence of range queries with variable selectivity, filtering data based on the timestamp attribute (Q1-29). Subsequently, we ask a sequence of range queries which filter data based on energy consumption measurements to identify a possible failure in the system (Q30-59). We then ask iterations of queries that filter results based on the timestamp attribute (Q60-79, Q92-94), the energy consumption (Q80-84, Q95-100), and the sensor id (Q85-91) respectively. Selectivity varies from 0.1% to 30%. Queries focusing on energy consumption are the least selective.

Figure 13 shows the response time of the different approaches for the SHD workload. All systems run from a hot state, with data resting in the OS caches. The indexed versions of PostgreSQL and DBMS-X build a B$^+$-Tree on the timestamp attribute. The figure plots only query execution time and does not show the time for loading or indexing for PostgreSQL and DBMS-X. For other other systems, where building auxiliary structures takes place during query execution, execution time contains the total cost.

PostgreSQL and DBMS-X without indexes perform full table scans for each query. Q30-60 are more expensive because they are not selective. For queries filtering on the timestamp, indexed PostgreSQL exhibits 10× better performance than PostgreSQL full table scan. Similarly, indexed DBMS-X exhibits 17× better performance compared to DBMS-X full table scan. As the queries using the index become more selective, response time is reduced. For the queries that do not filter data based on the indexed field, the optimizer of DBMS-X chooses to use the index despite the predicate involving a different attribute. This choice leads to response time slower than the DBMS-X full scan.

PostgresRaw is slightly faster than PostgreSQL without indexes. The runtime of the first query that builds the auxiliary structures (cache, positional map) is 8× slower (374 sec) than the average query runtime. For the rest of the queries PostgresRaw behaves similar to PostgreSQL and performs a full table scan for each query.

After the first query, Slalom identifies that the values of the timestamp attribute are unique. Thus, it chooses to statically partition the data following the cost model for query-based partitioning (Section 4.1) and creates 1080 partitions. Slalom creates the logical partitions during the second query and calculates statistics for each partition. Thus, the performance of Slalom is similar to that of PostgresRaw for the first two queries. During the third query, Slalom takes advantage of the implicit clustering of the file to skip the majority of the partitions, and decides whether to build an index for each of the partitions. After Q5, when Slalom has stabilized partitions and already built a number of indexes over them, the performance is better than that of the indexed PostgreSQL variation.

Queries Q2-Q30 represent a best-case scenario for DBMS-X: Data resides in memory and its single index can be used, therefore DBMS-X is faster than Slalom. After Q29, when queries filter on a different attribute, the performance of Slalom becomes equal to that of PostgresRaw until Slalom builds indexes. Because the energy consumption attribute has multiple appearances of the same value, Slalom decided to use homogeneous partitioning. Q30 to Q59 are not selective, thus execution times increase for all systems.

Table 2 shows the costs for loading and indexing as well as the aggregate query costs for the same query workload of 100 queries, for all the systems. Due to the queries being non-selective, the indexed and non-indexed approaches of DBMS-X have similar performance, thus in total Slalom exploits its adaptive approach to offer competitive performance to the fully indexed competitors.

Summary. Slalom serves a real-world workload which involves fluctuations in the areas of interest, and queries of great variety in selectivity. Slalom serves the workload efficiently due to its low memory consumption and its adaptivity mechanisms, which gradually lower query response times despite workload shifts.

6. CONCLUSION

In-situ data analysis over large and, crucially, growing data sets faces performance challenges as more queries are issued. State-of-the-art in-situ query execution reduces the data-to-insight time, however, as the number of issued queries is increasing and, more frequently, queries are changing access patterns (having variable selectivity, projectivity and are of interest in the dataset), in-situ query execution cumulative latency increases.

To address this, we bring the benefits of indexing to in-situ query processing. We present *Slalom*, a system that combines an in-situ query executor with an online partitioning and indexing tuner.

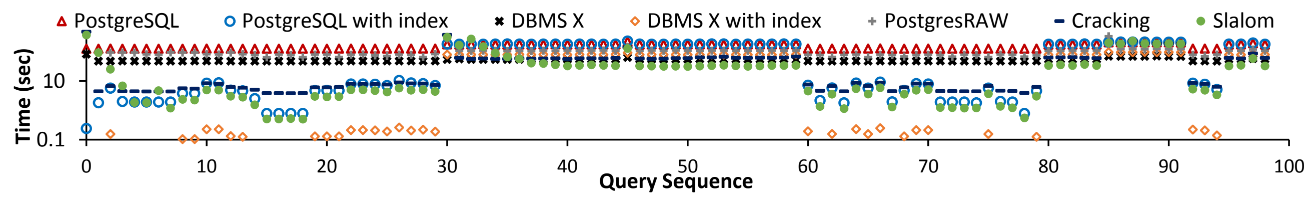


Figure 13: Sequence of SHD analytics workload. Slalom offers consistently comparable performance to in-memory DBMS.

Slalom takes into account user query patterns to reduce query time over raw data by partitioning raw data files *logically* and building for each partition lightweight *partition-specific* indexes when needed. The tuner further adapts its decisions on-the-fly to follow any workload changes and maintains a balance between the potential performance gains, the effort needed to construct an index, and the overall memory consumption of the indexes built.

Acknowledgments. We would like to thank the reviewers for their valuable comments. We also thank Christos Kalaitzis and Pavlos Nikolopoulos for their input on the cost model. This work is partially funded by the EU FP7 programme (ERC-2013-CoG), Grant No 617508 (ViDa), the EU FP7 Collaborative project Grant No 317858 (BigFoot) and EU Horizon 2020 research and innovation programme Grant No 650003 (Human Brain project).

7. REFERENCES

[1] C. L. Abad, N. Roberts, Y. Lu, and R. H. Campbell. A Storage-centric Analysis of MapReduce Workloads: File Popularity, Temporal Locality and Arrival Patterns. In *IISWC*, pages 100–109, 2012.
[2] A. Abouzied, D. J. Abadi, et al. Invisible Loading: Access-driven Data Transfer from Raw Files into Database Systems. In *EDBT*, pages 1–10, 2013.
[3] S. Agrawal, S. Chaudhuri, et al. Database Tuning Advisor for Microsoft SQL Server 2005. In *PVLDB*, 2004.
[4] S. Agrawal, V. Narasayya, et al. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *SIGMOD*, pages 359–370, 2004.
[5] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, et al. NoDB: Efficient Query Execution on Raw Data Files. In *SIGMOD*, pages 241–252, 2012.
[6] A. A. Alamoudi, R. Grover, et al. External Data Access And Indexing In AsterixDB. In *CIKM*, pages 3–12, 2015.
[7] K. Alexiou, D. Kossmann, and P.-A. Larson. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. In *PVLDB*, volume 6, pages 1714–1725, 2013.
[8] A. Anagnostou, M. Olma, and A. Ailamaki. Alpine: Efficient in-situ data exploration in the presence of updates. In *SIGMOD*, pages 1651–1654, 2017.
[9] M. Athanassoulis and A. Ailamaki. BF-Tree: Approximate Tree Indexing. In *PVLDB*, volume 7, pages 1881–1892, 2014.
[10] M. Athanassoulis and S. Idreos. Design Tradeoffs of Data Access Methods. In *SIGMOD*, 2016.
[11] M. Athanassoulis, M. Kester, et al. Designing Access Methods: The RUM Conjecture. In *EDBT*, 2016.
[12] M. Athanassoulis, Z. Yan, and S. Idreos. UpBit: Scalable In-Memory Updatable Bitmap Indexing. In *SIGMOD*, pages 1319–1332, 2016.
[13] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. Parallel Data Analysis Directly on Scientific File Formats. In *SIGMOD*, pages 385–396, 2014.
[14] B. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *CACM*, 1970.
[15] P. Borwein. On the complexity of calculating factorials. *J. of Algorithms*, 1985.
[16] N. Bruno and S. Chaudhuri. An Online Approach to Physical Design Tuning. In *ICDE*, pages 826–835, 2007.
[17] S. Chaudhuri and V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *PVLDB*, pages 146–155, 1997.
[18] Y. Chen et al. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. *PVLDB*, 2012.
[19] Y. Cheng and F. Rusu. Parallel In-situ Data Processing with Speculative Loading. In *SIGMOD*, pages 1287–1298, 2014.
[20] J. Chou, M. Howison, B. Austin, K. Wu, J. Qiang, E. W. Bethel, et al. Parallel Index and Query for Large Scale Data Analysis. In *SC*, pages 30:1–30:11, 2011.
[21] D. J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, et al. Split Query Processing in Polybase. In *SIGMOD*, pages 1255–1266, 2013.
[22] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. In *ACM TODS*, volume 13, pages 91–128, 1988.
[23] C. Furtado, A. A. B. Lima, E. Pacitti, P. Valduriez, et al. Physical and virtual partitioning in OLAP database clusters. In *SBAC-PAD*, pages 143–150, 2005.
[24] V. R. Gankidi, N. Teletia, et al. Indexing HDFS Data in PDW: Splitting the Data from the Index. In *PVLDB*, volume 7, pages 1520–1528, 2014.
[25] G. Graefe and H. Kuno. Self-selecting, Self-tuning, Incrementally Optimized Indexes. In *EDBT*, pages 371–381, 2010.
[26] G. Graefe and W. J. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *PVLDB*, pages 209–218, 1993.
[27] M. Grund, J. Krüger, H. Plattner, A. Zeier, et al. HYRISE: A Main Memory Hybrid Storage Engine. In *PVLDB*, volume 4, pages 105–116, 2010.
[28] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-memory Column-stores. In *PVLDB*, volume 5, pages 502–513, 2012.
[29] T. Härder. Selecting an Optimal Set of Secondary Indices. In *ECI*, 1976.
[30] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *CIDR*, pages 57–68, 2011.
[31] S. Idreos, M. L. Kersten, et al. Database cracking. In *CIDR*, pages 68–78, 2007.
[32] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing Tuple Reconstruction in Column-stores. In *SIGMOD*, pages 297–308, 2009.
[33] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. In *PVLDB*, pages 585–597, 2011.
[34] M. Ivanova, M. Kersten, et al. Data Vaults: A Symbiosis between Database Technology and Scientific File Repositories. In *SSDBM*, pages 485–494, 2012.
[35] A. Jindal and J. Dittrich. Relax and Let the Database Do the Partitioning Online. In *BIRTE*, pages 65–80, 2012.
[36] Y. Kargn, M. Kersten, S. Manegold, and H. Pirk. The DBMS - your big data sommelier. In *ICDE*, pages 1119–1130, 2015.
[37] A. R. Karlin, M. S. Manasse, L. A. McGeoch, et al. Competitive Randomized Algorithms for Non-uniform Problems. In *SODA*, pages 301–309, 1990.
[38] M. Karpathiotakis, I. Alagiannis, et al. Fast Queries over Heterogeneous Data Through Engine Customization. In *PVLDB*, volume 9, pages 972–983, 2016.
[39] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, et al. Just-In-Time Data Virtualization: Lightweight Data Management with ViDa. In *CIDR*, 2015.
[40] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive Query Processing on RAW Data. In *PVLDB*, volume 7, pages 1119–1130, 2014.
[41] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, A. Choi, J. Erickson, et al. Impala: A modern, open-source sql engine for hadoop. In *CIDR*, 2015.
[42] S. S. Lightstone, T. J. Teorey, et al. *Physical Database Design: The Database Professional's Guide to Exploiting Indexes, Views, Storage, and More*. 2007.
[43] S. Melnik, A. Gubarev, J. J. Long, et al. Dremel: Interactive Analysis of Web-scale Datasets. In *PVLDB*, volume 3, pages 330–339, 2010.
[44] G. Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *PVLDB*, 1998.
[45] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, et al. Instant Loading for Main Memory Databases. In *PVLDB*, volume 6, pages 1702–1713, 2013.
[46] S. Papadomanolakis et al. AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In *SSDBM*, pages 383–, 2004.
[47] K. Pearson. Contributions to the Mathematical Theory of Evolution. II. Skew Variation in Homogeneous Material. 1895.
[48] E. Petraki, S. Idreos, and S. Manegold. Holistic Indexing in Main-memory Column-stores. In *SIGMOD*, pages 1153–1166, 2015.
[49] S. Richter, J.-A. Quiané-Ruiz, et al. Towards Zero-overhead Static and Adaptive Indexing in Hadoop. In *PVLDB*, volume 23, pages 469–494, 2014.
[50] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. COLT: Continuous On-line Tuning. In *SIGMOD*, pages 793–795, 2006.
[51] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The Uncracked Pieces in Database Cracking. In *PVLDB*, volume 7, pages 97–108, 2013.
[52] L. Sidirourgos and M. Kersten. Column Imprints: A Secondary Index Structure. In *SIGMOD*, pages 893–904, 2013.
[53] R. R. Sinha, S. Mitra, and M. Winslett. Bitmap Indexes for Large Scientific Data Sets: A Case Study. In *IPDPS*, pages 68–68, 2006.
[54] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained partitioning for aggressive data skipping. In *SIGMOD*, pages 1115–1126, 2014.
[55] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, et al. Hive: A warehousing solution over a map-reduce framework. In *PVLDB*, volume 2, pages 1626–1629, 2009.
[56] E. Wu and S. Madden. Partitioning techniques for fine-grained indexing. In *ICDE*, pages 1127–1138, 2011.
[57] K. Wu, S. Ahern, E. W. Bethel, et al. FastBit: Interactively Searching Massive Data. *SCIDAC*, 2009.
[58] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. Db2 design advisor: Integrated automatic physical database design. In *PVLDB*, pages 1087–1097, 2004.