Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe?

Michael S. Kester Manos Athanassoulis Stratos Idreos

Harvard University
{kester, manos, stratos}@seas.harvard.edu

ABSTRACT

The advent of columnar data analytics engines fueled a series of optimizations on the scan operator. New designs include column-group storage, vectorized execution, shared scans, working directly over compressed data, and operating using SIMD and multi-core execution. Larger main memories and deeper cache hierarchies increase the efficiency of modern scans, prompting a revisit of the question of access path selection.

In this paper, we compare modern sequential scans and secondary index scans. Through detailed analytical modeling and experimentation we show that while scans have become useful in more cases than before, both access paths are still useful, and so, access path selection (APS) is still required to achieve the best performance when considering variable workloads. We show how to perform access path selection. In particular, contrary to the way traditional systems choose between scans and secondary indexes, we find that in addition to the query selectivity, the underlying hardware, and the system design, modern optimizers also need to take into account *query concurrency*. We further discuss the implications of integrating access path selection in a modern analytical data system. We demonstrate, both theoretically and experimentally, that using the proposed model a system can quickly perform access path selection, outperforming solutions that rely on a single access path or traditional access path models. We outline a light-weight mechanism to integrate APS into main-memory analytical systems that does not interfere with low latency queries. We also use the APS model to explain how the division between sequential scan and secondary index scan has historically changed due to hardware and workload changes, which allows for future projections based on hardware advancements.

1. INTRODUCTION

Access Path Selection. Access path selection for filtering operators has always been a key component of database systems [70]. When a query is predicated on a clustered index, using the index is the obvious choice. Similarly, when a query is predicated on an attribute with no index, there is only one access path available, a sequential scan. The more difficult and more common case is when a query is predicated on a column with a secondary index. In this case, a secondary index scan may or may not work better than a full sequential scan [70]; for such cases, the data system needs to perform access path selection to choose the best access method, relying on run-time characteristics during optimization. Having the choice between a secondary index scan and a sequential scan of the base data allows systems to choose the best access path by taking into account run-time characteristics during optimization. A sequential scan over the base data reads every tuple, while a secondary index scan only needs to access an auxiliary copy of the data which is smaller and has more structure. Secondary indexes, typically in the form of B$^+$-Trees [30, 70], have been extensively used in row-oriented systems to access target data, with neither the overhead of reading the unneeded values nor their associated neighboring attributes. The decision of access path selection is typically based on a selectivity threshold, along with the underlying hardware properties, system design parameters, and data layout [29, 32]; once the system is tuned it is a fixed point used for all queries.

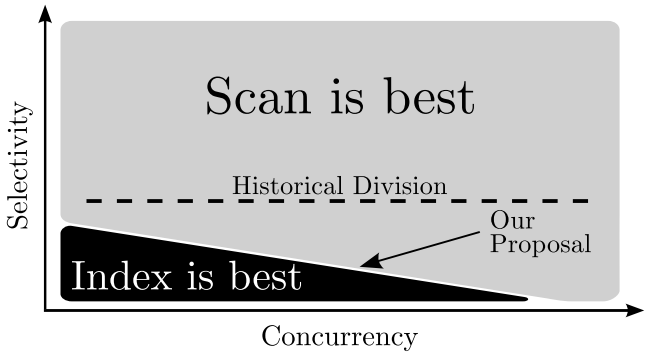

Figure 1: Access path selection in modern data analytics systems should include query concurrency in addition to selectivity, hardware characteristics, and data layout. There is no fixed selectivity threshold; rather, there is a sloped division that depends on the # of concurrent queries and their total selectivity.

Access Paths In Modern Analytical Systems. Over the past two decades, database systems dramatically departed from the design of traditional row-major systems in favor of columnar (and more recently column-group) storage and highly tuned access paths for main-memory [1]. In such analytical systems, a plethora of optimizations and techniques make scans extremely fast and put increasing pressure on secondary indexing. First, *column* or *column-group* storage allows a system to process only the attributes needed to answer a given query, avoiding unneeded reads in much the same way a secondary index does [4, 19, 25, 33, 43, 56, 77]. Second, in contrast to "tuple-at-a-time" processing, *vectorized execution* passes a block of tuples to each operator. This increases efficiency

SIGMOD '17, May 14–19, 2017, Chicago, IL, USA.

DOI: http://dx.doi.org/10.1145/3035918.3064049

by processing each block in a tight loop, reducing interpretation logic overhead [16]. Third, when the system is processing more than one query across the same attribute(s), *scans can be shared* [7, 18, 27, 28, 34, 62, 63, 88]. For each block of the relation, multiple queries can be processed resulting in advantageous disk, memory, or cache utilization compared to a "query-at-a-time" approach. Shared scans are an example of wider trend toward cache friendly access methods [65, 87]. Fourth, modern analytical systems use *compression* more effectively than row-stores by compressing individual columns independently and working directly over compressed data which reduces the cost of moving data through the memory hierarchy [2, 12, 14, 15, 42, 77]. Fifth, holding each attribute contiguously in a *dense array* allows tight `for` loop evaluation [51, 55, 86] and lends itself well to *single instruction multiple data* (SIMD) processing [60, 81, 84]. Where SIMD allows multiple sequential tuples to be processed in parallel, multiple cores allow disparate parts of the relation to be processed in parallel [63, 80].

Many of the techniques that make scans for modern analytical data systems efficient have been explored in past research (e.g., compression [31], columnar data organization [44], main-memory systems [24], scan sharing [89]). However, achieving the full potential required a completely new system design which incorporated the past research from the start and grew into a completely new framework [1, 37, 49, 77, 85]. This research has been largely transferred to most of the industry (e.g., HyPer [59], DB2 [64] and dashDB [13], SAP HANA [25], Oracle's in-memory database [48], Vertica [49], Vectorwise [85], MemSQL [75], and SQL Server [50]). We use the term *modern analytical systems* instead of *column-stores* as these systems use many more techniques than just storing data one column at a time.

Access Path Decisions in Main-Memory. All the above, together with the fact that (hot) data can be memory resident in many cases due to large memories, bring into question the need for secondary indexes. Essentially, what is perceived as the main benefit of a secondary index in a traditional row-store, minimizing data movement from disk, is no longer on the critical path both due to data layouts and to large memories. Hence, many modern analytical systems (at least in their early phases of development) opt not to include secondary indexing at all and instead focus on maximizing scan performance using a single access path (together with clustered indexes and data skipping techniques like zone maps [58]). In addition, given that response times for in-memory processing are typically low, there is limited time to devote to access path selection; optimization time has become the new bottleneck [21, 73].

Open Questions. We answer two questions in this paper: (1) Is there still a case for secondary indexes in modern analytical systems? And if so, (2) how should we perform access path selection given the advancements in system design? What makes the first question interesting (other than the fact that many modern systems opt out of secondary indexes) is the continued optimization of scans [54, 55, 60]. Features like shared scans are increasingly common and concurrency for read queries is increasing exponentially [69]. When multiple queries can be answered with a single sequential scan, is there still a need for secondary indexing? Given the sub-second performance of many queries, is there even time to make a meaningful decision about which access path to use?

Access Path Selection in Modern Analytical Systems. We show both analytically and experimentally that while scans have indeed become useful in more situations than before, secondary indexes are still useful for queries with low selectivity.[1] We demonstrate that in order to perform access path selection in modern systems, in addition to the traditional way of using query selectivity and the environment characteristics, query concurrency also needs to be incorporated to optimizers. The number of concurrent queries at any given time is critical as it affects the number of queries that may share a column-scan, or a secondary index tree traversal.

We show that there should be no fixed selectivity threshold for access path selection as there has been for the last 35 years [6, 21, 70]; rather, there is a moving pivot point that primarily depends on the number of concurrent queries and their total selectivity. Figure 1 shows the basic concept in our access path selection approach and how it differs from the historical method. Traditionally, a system is tuned to the hardware and then a fixed selectivity point is determined. Any queries that are expected to return a larger percent of the relation than this point use a sequential scan. The change we propose for modern analytical systems is that there should be no fixed selectivity point for access path selection. The break-even point is variable, it decreases in terms of selectivity as more queries are run concurrently. We also find that in some cases there is a theoretical point where a shared scan is always the best option in the far right in Figure 1. The point varies as concurrency goes over a threshold which is dependent on the underlying hardware, data layout, and physical design decisions. However, it is not feasible with current technology to have an arbitrarily high level of concurrency due to other limiting factors (e.g., data accesses associated with result writing, which most notably may cause TLB misses).

System Integration. We discuss how to integrate this new form of access path selection in modern analytical systems. The optimizer needs to model the hardware (CPU, cache, memory latency and bandwidth) and the workload (selectivity and query concurrency), and to continuously monitor the latter. This allows the system to make dynamic decisions about using a shared scan or a shared secondary index scan for a group of queries predicated on the same data. It should also adjust dynamically depending on the underlying layout of the accessed data (pure column, column-groups for hybrid storage, or even full rows). We combine these techniques in our prototype, FastColumns, to show both that access path selection is beneficial for analytics and that it can be done without hurting the performance of even sub-second queries.

In this paper, we focus on read performance. Modern main-memory optimized systems targeting analytics either do not support updates (only appends) or support updates through a separate write-store (or a form of delta store) that accumulates updates and only periodically merges them into the read-optimized store [35, 47, 49, 77]. Our access path selection analysis targets analytical queries in the read-store.

Contributions. This work offers the following contributions:

- We present an augmented access path cost model that captures the performance of select operators in main-memory optimized analytical data systems that support access sharing (§2).
- We show that access path selection is needed; tuned secondary indexes can be useful even when fast scans are employed (§2).
- We show that in addition to selectivity and hardware characteristics, access path selection needs to *dynamically* take into account query concurrency (§2).
- We integrate our access path selection to the optimizer of a modern analytical prototype and show that even though access path selection is now a more complex operation that must take into account more information, it can be done quickly enough that it remains beneficial, even for sub-second queries.

[1] We use the term "selectivity" in the same way as "selectivity factor" is used in Pat Selinger's paper [70], as a number rather than a property. Hence, we use "high selectivity" to indicate that the result set of a query has many qualifying tuples, and the converse for "low selectivity".

- We demonstrate that access path selection can be done on a variety of workloads outperforming solutions that rely on a single access path or on traditional access path selection models (§4).
- Using the model we show how the decision of when to use an index has changed as the data layout and hardware properties have changed over the last five decades (§5).

2. ACCESS PATH SELECTION

In this section we present a model for access path selection in a main-memory optimized analytical data system. We first provide notation and other preliminaries (§2.1). We continue by modeling sequential scans (§2.2). Similarly to past modeling approaches [16, 56] we take into account the underlying hardware characteristics, as well as a plethora of techniques that boost scan performance like tight `for` loops, using multiple cores, working over compressed data, and crucially (in contrast to previous approaches [6, 21, 70]) we augment the modeling with scan sharing that processes multiple queries concurrently. Then, we model concurrent accesses in a main-memory optimized B$^+$-Tree (§2.3). The model supports data layouts ranging from pure columnar to pure row-oriented (and hybrid layouts in-between), and optionally value-based compression (numeric compression or dictionary compression).

Using this model, we develop an *access path selection* strategy to choose between a sequential scan and a secondary index scan for a given batch of queries (§2.4). We show that in addition to selectivity and hardware characteristics (which are used in traditional access path selection) the number of concurrent queries in the batch, the overall selectivity of the batch, and how each query contributes to that total are also critical.

The modeling presented in this section is an integral part of the design presented in Section 3, where we discuss how to integrate access path selection in analytical data systems.

2.1 Model Preliminaries

Select Operator. Our study is about the performance of `select` operators which typically filter out most of the data in a query. Hence, they are important to estimating the total cost of a query plan. We maintain the standard API for a `select` operator. The input is a column (for pure column-store) or a column-group (for hybrid or pure row-storage). The task of the select operator is to generate a list of all qualifying tuples given a predicate (i.e., range or point query). The output is a collection of rowIDs, which are offsets in a fixed-width and dense column.

Parameters and Notation. The parameters of our access path selection model capture (i) the query workload, (ii) the dataset and physical storage layout, (iii) the underlying hardware, and (iv) the index and scan design. Table 1 shows these parameters as well as the notation used in the model.

Modeling the Query Workload. The query workload is modeled using two parameters, the number of concurrent queries and their selectivity. We model the number of queries, q, and their respective selectivity, s_i, for $i = 1,...,q$. An individual selectivity, s_i, corresponds to the percentage of tuples that qualify in query i based on its predicate. The size of the overall result set of a batch is given by the sum of the q individual selectivities, termed *total selectivity*, $S_{tot} = \sum_{i=1}^{q} s_i$. S_{tot} can be more than 100%; e.g., for three queries each with 40% selectivity, S_{tot} would be 120%.

Modeling Data Layout. Next, we model the physical storage layout by taking into account the shape and size of the input data. Both the sequential scan and the secondary index scan operate on a single column or group of columns. We use N to model the cardinality of the relation in tuples, and ts to describe the width of each tuple

Workload	q	number of queries
	s_i	selectivity of query i
	S_{tot}	total selectivity of the workload
Dataset	N	data size (tuples per column)
	ts	tuple size (bytes per tuple)
Hardware	C_A	L1 cache access (sec)
	C_M	LLC miss: memory access (sec)
	BW_S	scanning bandwidth (GB/s)
	BW_R	result writing bandwidth (GB/s)
	BW_I	leaf traversal bandwidth (GB/s)
	p	The inverse of CPU frequency
	f_p	Factor accounting for pipelining
Scan & Index	rw	result width (bytes per output tuple)
	b	tree fanout
	aw	attribute width (bytes of the indexed column)
	ow	offset width (bytes of the index column offset)

Table 1: *Parameters and notation used to model access methods and to perform access path selection.*

(consisting of a single attribute, a group of attributes, or the entire row). The width of every output value is given by rw. Modeling the index scan uses two additional parameters, the width of the index attribute, aw, and the width of the rowIDs, ow.

Modeling Hardware Characteristics. Another important factor of the model is the underlying hardware. Our focus is on main-memory optimized systems, so data (at least frequently used data) is expected to be memory resident making the primary cost component memory accesses in the different levels of the memory hierarchy. Everything is measured in time units; accesses are either a cache access (hit), C_A, or a last level cache miss resulting in a main memory access, C_M, while data is read sequentially from memory at memory bandwidth speed (in bytes per second). The model differentiates the memory bandwidth speed when scanning sequentially (BW_S), when writing the result in an output buffer (BW_R), and when scanning the contents of the leaves of a tree (BW_I).

Other Scan Enhancements. Single query performance can be enhanced with data skipping techniques like zonemaps. Zonemaps are commonly found in analytical systems as a means to avoid reading a group of values that can not qualify for a given query based on a check of the bounds of the group. For a single query this is modeled by simply reducing the number of values in the relation by the expected number of zones skipped and the number of entries in a zone. One drawback of zonemaps is that as concurrency increases the number of zones that can be skipped is reduced as in order to skip a zone, it must be unneeded by *all* queries in the group.

Compression. Data may be compressed and access paths can work directly over compressed data. We consider order preserving dictionary compression [12, 81] where an access path first probes the dictionary for the low and high values, and then evaluates the predicate on the compressed data directly. We use a single dictionary per column and data compressed to two bytes. When the value domain of the data is very small (256 distinct values or less), then a column could be compressed even further. However, in these cases alternate access paths, such as bitmap indexes, should also be considered.

Tuning B$^+$-Trees. Last but not least, the performance of an in-memory tree-based secondary index is heavily affected by the shape of the tree. The main parameter that defines the shape of the tree is the fanout, b. This affects both memory efficiency and the number of random accesses incurred in a tree index traversal.

2.2 Modeling In-Memory Shared Scans

At the core, a sequential scan is an iteration over an array of input data to find qualifying tuples based on a predicate comparison. This is done in a tight `for` loop where each tuple is evaluated in

succession (Figure 2(a)). In modern data systems, each attribute is stored in dense arrays with fixed-width elements (or as wider arrays that contain adjacent attributes in the case of hybrid storage). Such an iteration over sequential memory results in efficient evaluation limited only by the memory bandwidth.

Data Movement for Scan. In tight `for` loop scans data in the underlying array is accessed sequentially. Given memory bandwidth BW_S, the cost of sequentially scanning the data, TD_S, (reading N tuples of size ts) in seconds is:

$$TD_S = \frac{N \cdot ts}{BW_S} \quad (1)$$

Predicate Evaluation. The CPU cost to evaluate the predicate might affect the scan's performance. A scan needs to evaluate the predicate (a lower and a higher bound) for each value it reads. Assuming that p is the period of the processor's clock, and f_p is a constant factor to account for instruction pipelining, the CPU of predicate evaluation PE is:

$$PE = 2 \cdot f_p \cdot p \cdot N \quad (2)$$

We assume aggressive utilization of SIMD and multi-core to optimize the CPU cost of each sequential scan. Each query (select operator) is assigned to a single hardware thread if the total number of concurrent queries is bigger than the number of available threads. Otherwise, the available hardware threads are spread to the current queries and each query may use more than one.

Depending on the hardware characteristics the CPU cost might be on the critical path, as a result, every scan has a data consumption cost equal to the maximum of TD_S and PE.

When data is compressed, the scan probes the dictionary for the low and high values. The process is repeated more than once if a column is encoded with more than one dictionary. The cost of probing a dictionary is two cache misses and for simplicity we do not include it in the model, because in practice this is multiple orders of magnitude smaller than the remaining cost (for both scans and indexes). The variable width of the compressed values is captured by the ts parameter.

Result Writing. Another significant component of the total cost of a scan is the cost of result writing, TD_R. The cost to output the whole column would need N rows of rw bytes each:

$$TD_R = \frac{N \cdot rw}{BW_R} \quad (3)$$

In order to avoid branch mispredictions, which can be costly in tight loops, we always generate the result-set using predication [5, 11]. Since this write is either to an already cached line or a sequential piece of memory, it also executes at memory bandwidth speed.

Therefore, the cost of a scan of a single query i is:

$$SingleQueryCost = max(TD_S, PE) + s_i \cdot TD_R \quad (4)$$

Equation 4 is the combination of three logical parts. First, retrieving the data which is a function of the number of tuples, their size, and the memory bandwidth. Second, the CPU cost of evaluating the predicate over those, which depends on CPU characteristics. Third, we include the cost of writing back the results (IDs of the qualifying tuples) which happens at memory bandwidth. The model further differentiates between TD_S and TD_R; a scan may traverse a column or a group of columns depending on how the data are physically organized. The result of the scan can also be a single column or a group of columns.

Scan Sharing. We now extend the modeling to scan sharing where q queries share a single scan of the base data [63, 88]. Queries can share the cost of data movement by moving data up the memory hierarchy once, while analyzing multiple queries. However, the

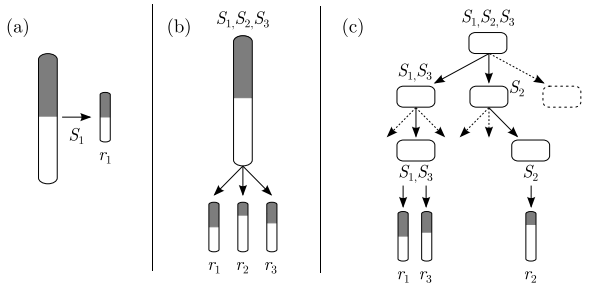

Figure 2: Sequential scan and secondary index scan access patterns of the select operator. Basic scans (a) iterate sequentially through the input and evaluate the predicate to find qualifying positions. Shared scans (b) iterate over the relation only once. Batched B$^+$-Tree traversal (c) issues prefetches at each level but only for the children that are needed.

predicate evaluation cost now increases with the number of queries. Given that data movement is one of the major bottlenecks, scan sharing is of critical importance. To perform scan sharing a system needs to first group queries based on the attributes they are predicated on. Then, a small part of the input data, one that easily fits into the hardware cache, is evaluated for each query in the batch iteratively (Figure 2(b)). This allows more than one query to use the target data before it is evicted from the processor cache, reducing memory bandwidth pressure. Each query uses a separate hardware thread to spread the work to every system core.

When multiple queries share a scan, the cost of data reads is the same as with a single query (assuming that all q queries start at the same time). However, we now have to consider the increased CPU cost to evaluate the q predicates. Regarding result writing, each query requires a separate result set. Hence, the cost to generate the result sets for the queries is $\sum_{i=1}^{q} s_i \cdot TD_R = S_{tot} \cdot TD_R$.

Putting everything together gives us the overall cost for an in-memory shared scan of q queries:

$$SharedScan = max(TD_S, q \cdot PE) + S_{tot} \cdot TD_R \quad (5)$$

Equation 5 is yet again the combination of three logical parts: data retrieval, predicate evaluation, and result writing, as shown in Figure 3(a). While the cost to retrieve the base data remains the same as in Equation 4, now we account for the q multiple predicate evaluations and corresponding result sets (by using total result size S_{tot} instead of individual query s_i).

2.3 Modeling In-Memory Secondary B$^+$-Trees

We now explore the possibility of using a secondary index as the access path. We use a B$^+$-Tree and we tune it for a given hardware and in-memory operation. In other words, we perform the same kind of state-of-the-art optimizations we have available for scans before comparing the two access methods.

We optimize the secondary index in two ways. First, the fanout of the tree is tuned to match the memory latency and bandwidth of the underlying hardware. Second, the intra-node search (locating the next child node to visit) is tuned to match the fanout.

Selects Using a Secondary Index. To integrate a B$^+$-Tree as a secondary index, the tree stores a copy of the indexed attribute in its leaves. The leaves also contain the respective rowIDs (i.e., positions of the indexed values in the base arrays). A select operator can then operate directly on top of the tree and output the result positions in the same way as a sequential scan does (though the positions are generally out of order which we discuss later).

Modeling Secondary Index Scan. An in-memory B$^+$-Tree probe has multiple steps. First, the tree is traversed to find the first leaf corresponding to the requested value range. Second, we traverse

the leaves reading the indexed data. Finally, we write the result set which is ordered by the indexed attribute (if we follow the natural order generated by the index). Alternatively, the data is sorted on the rowID in order to deliver an identical result as the sequential scan. Below, we analyze and model these steps in detail.

Tree Traversal. For a B$^+$-Tree with fanout b the tree traversal cost, in seconds, from the root to the first corresponding leaf, is a function of the number of tuples N and is given by:

$$TT = (1 + \lceil log_b(N) \rceil) \cdot \left(C_M + \frac{b \cdot C_A}{2} + \frac{b \cdot f_p \cdot p}{2} \right) \quad (6)$$

Equation 6 assumes that traversing from a node to its child results in a random memory access which will be incurred based on the height of the tree and that, on average, for each internal node, half of the keys will be read sequentially and the corresponding search predicate will be evaluated to find the target child node.

Leaves Traversal. Using the selectivity of a query i, s_i, we can calculate approximately the number of leaf nodes that the system must traverse to collect the qualifying rowIDs. Since the leaf nodes are in arbitrary memory locations, each time we need to visit a new leaf (when the result spans more than one leaf) a cache miss is incurred. Thus, given that N/b is the number of leaves in the tree, the leaf traversal cost for each query is:

$$s_i \cdot TL, \text{ where, } TL = \frac{N \cdot C_M}{b} \quad (7)$$

Data Traversal for Secondary Indexes. Reading the rowIDs within a leaf node, however, can be modeled as properly prefetched sequential accesses of values and rowIDs, using the bandwidth that leaf contents can be traversed, BW_I. Thus, the cost for each query depends on the attribute width (aw) and the offset width (ow):

$$s_i \cdot TD_I, \text{ where, } TD_I = \frac{N \cdot (aw + ow)}{BW_I} \quad (8)$$

When data is compressed with order preserving dictionary compression the index traversal benefits from accessing more keys in each node potentially resulting in a shorter tree (fewer cache misses). Similar to scan this is at a cost of two probes at the dictionary to get the codes for the low and high query bounds. Contrary to scans, for secondary indexes we assume a single dictionary per column.

Result Writing. Another significant component of the total cost of a secondary index probe is the cost of result writing. Similar to the sequential scan, result writing adds an overhead (in seconds) of: $RW = s_i \cdot N \cdot \frac{rw}{BW_R}$ (which is in fact equal to $s_i \cdot TD_R$).

Sorting the Result Set. To deliver the same result to the next operator as the scan operator, a secondary index traversal for a query i needs to sort the output based on the rowIDs. Accounting one cache access for each comparison the sorting cost is equal to:

$$SC_i = s_i \cdot N \cdot log_2(s_i \cdot N) \cdot C_A \quad (9)$$

The sorting cost can be further reduced if more efficient SIMD-aware sorting algorithms are employed (see Appendix D). While this sorting is not needed when performing a secondary index traversal independently, it is a necessary cost to offer an access method directly comparable with a scan. This also depends on where the secondary index probe is in the query plan. For example, in modern column-stores where tuple-reconstruction is a heavy cost component (i.e., fetching an attribute/column based on a selection on another attribute) having an unsorted result from a select operator will force a tuple reconstruction operator to operate with random access patterns [3, 38]. On the other hand, a full query optimizer can decide to use the output of a scan ordered on its values if this can benefit subsequent operators, like joins and aggregates.

Putting everything together, the cost (in seconds) of a single query using a secondary index traversal is:

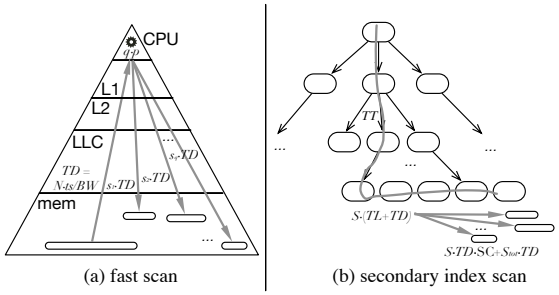

Figure 3: Illustration of access path operation. The left hand-side (a) of the figure shows the cost of a shared fast scans (Equation 5). The right hand-side (b) shows the cost of a memory optimized secondary index scan (Equation 13).

$$SingleIndexProbe = TT + s_i \cdot (TL + TD_I) + RW + SC_i \quad (10)$$

Concurrent Index Access. We now extend the analysis for the case where q queries want to select over the same column (or groups of columns) at the same time. Similar to the optimizations for the sequential scan, more than one query can share an index scan as well. For ease of exposition, this analysis assumes that all q queries arrive at the same time. The goal is to utilize the parallelism offered by modern multi-core processors and to minimize data movement by sharing data accesses. A shared index scan first divides the concurrent overlapping queries by the number of hardware threads available, t. Each thread is responsible for q/t queries. While this is not explicitly modeled, concurrent accesses often lead to natural sharing in the cache as different queries traverse overlapping parts of the tree. As a result, the model in this section presents a *worst case* analysis. Natural sharing is most common at the top levels of the tree, so the tree traversal cost is dominated by the other terms.

In a shared index scan of q queries, the tree is traversed in total q times. Figure 3(b) shows a shared index scan corresponding to multiple queries. Each query traverses the tree, the corresponding leaves, and the data independently. Some of these accesses may be cached due to a previous query but in the worst case they are always misses. As a result the overall cost of accessing leaves and data is given by the total selectivity S_{tot}. Similarly, the result writing cost is about the overall selected data, given by S_{tot}. Finally, we account for the cost of sorting the results on rowID for all q queries. Summing up the individual cost (shown in Equation 9), we get: $\sum_{i=1}^{q} SC_i = \sum_{i=1}^{q} s_i \cdot N \cdot log_2(s_i \cdot N) \cdot C_A$.

Putting everything together, the overall cost of an in-memory shared secondary index scan is as follows:

$$\begin{aligned} ConcIndex =& q \cdot TT + S_{tot} \cdot (TL + TD_I) + S_{tot} \cdot TD_R \\ &+ \sum_{i=1}^{q} s_i \cdot N \cdot log_2(s_i \cdot N) \cdot C_A \end{aligned} \quad (11)$$

In Appendix A we present an analysis based on the definition of entropy to show that the sorting cost $SC = \sum_{i=1}^{q} s_i \cdot N \cdot log_2(s_i \cdot N)$ has the following maximum value:

$$MaxSC = S_{tot} \cdot N \cdot log_2(S_{tot} \cdot N) \quad (12)$$

Putting together Equation 11 and 12 (and allowing the sorting cost estimation to be corrected by a factor f_c), the worst case for the cost of a concurrent index access becomes:

$$\begin{aligned} ConcIndex =& q \cdot TT + S_{tot} \cdot (TL + TD_I) \\ &+ S_{tot} \cdot TD_R + SF \cdot CA, \ \ where \end{aligned} \quad (13)$$

$$SF = S_{tot} \cdot N \cdot log_2(S_{tot} \cdot N) \quad (14)$$

Equation 13 is the combination of five logical parts: tree traversal, leaves traversal, data retrieval, result writing, and result sorting. When compared with Equation 10, the concurrent index access may traverse the tree multiple times (one for each query). Finally, the amount of work to write and sort the result depends on the sum of the selectivity of each query.

2.4 Evaluating Access Path Selection

We now discuss *access path selection*. Using the model developed in the previous sections we construct a ratio of the two costs using Equations 5 and 13. We call this ratio *APS* (Access Path Selection) and it is defined as $APS = \frac{ConcIndex}{SharedScan}$. The optimizer in a modern analytical system can use this model to determine which access path to deploy (more in §3). When $APS \geq 1$ a scan should be used; when $APS < 1$ a secondary index access is beneficial.

In the rest of this section we present an analysis that provides intuition and observations about access path selection choices and cases, demonstrating evidence that it is indeed useful to support alternative access paths in modern analytical systems and how these choices are different compared to traditional systems and optimizers. The analysis in this section is based on modeling. In Section §4, we present a detailed experimental analysis that corroborates the results and provides further insights.

When Should We Switch Access Paths? To ease the presentation of the *APS* ratio, we use the four quantities defined in Equations 1, 3, 6, 7, 8, and 14, for both the *SharedScan* and the *SharedIndex* costs: the cost to *traverse the tree* (TT), the cost to *traverse the leaves* (TL), the cost to *traverse the data* (TD_S, TD_R, and TD_I), and the *sorting factor* (SF). The costs related to the tree, TT and TL, depend on the relation size, the index design and the hardware characteristics. The costs to traverse the data depend on the data size and the sustainable read bandwidth. Finally, the sorting factor SF is incurred by sorting the index output before passing it on to the next operator in order to build a direct competitor of the scan operator. Using Equations 5 and 13 the access path selection ratio is defined as follows:

$$APS = \frac{q \cdot TT + S_{tot}\,(TL + TD_I + TD_R) + SF \cdot C_A}{max\,(TD_S, q \cdot PE) + S_{tot} \cdot TD_R} \tag{15}$$

In fact, *APS* is a multivariate function, that depends on all parameters presented in Table 1. For each instance of a system, though, the hardware and the physical layout decisions are already made. In addition, at runtime the data set size is also known, so runtime access path selection is effectively a function of the level of query concurrency q and the total selectivity S_{tot}: $APS\,(q, S_{tot})$.

Following the derivation presented in Appendix B the APS ratio can be rewritten as follows:

$$\begin{aligned} APS\,(q, S_{tot}) = & \frac{q \cdot \frac{1+\lceil log_b(N) \rceil}{N} \cdot \left(BW_S \cdot C_M + \frac{b \cdot BW_S \cdot C_A}{2} + \frac{b \cdot BW_S \cdot f_p \cdot p}{2} \right)}{max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}} \\ & + \frac{S_{tot} \left(\frac{BW_S \cdot C_M}{b} + (aw + ow) \cdot \frac{BW_S}{BW_I} + rw \cdot \frac{BW_S}{BW_R} \right)}{max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}} \\ & + \frac{S_{tot} \cdot log_2\,(S_{tot} \cdot N) \cdot BW_S \cdot C_A}{max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}} \end{aligned} \tag{16}$$

Understanding the APS Fraction. Equation 16 is comprised from two different parts. The first one takes into account query concurrency q, the data set size N, the index branching factor b, and the hardware trade-offs: bandwidth vs. LLC miss, and bandwidth vs. L1 latency. The second part is mainly affected by the predicate evaluation cost, the total selectivity S_{tot}, and the data layout.

Let us now examine step by step the two parts of the numerator and the denominator. The first part of the numerator is affected by the level of concurrency, however, it has a rather small impact for low q because of the term $\frac{\lceil log_b(N) \rceil}{N}$. This means that for low concurrency, the cost of traversing the index is very small. The cost increases only slightly as concurrency increases. The products $BW_S \cdot C_M$ and $b \cdot BW_S \cdot C_A$ depend on hardware characteristics and can be explained as follows: The first is the number of bytes that can be read sequentially from the memory bus in the delay caused by an LLC miss and the second is the number of bytes that can be read sequentially from the memory bus in the same duration as b L1 accesses. Typically, the latter, is about one order of magnitude smaller than the first. Similarly, the term $b \cdot BW_S \cdot f_p \cdot p$ corresponds to the number of bytes that can be read sequentially during the time needed to perform the b comparisons when searching which pointer to follow. These three terms connect scan and index accesses from the hardware point of view (see more details in Appendix B.1).

The second part of the numerator in Equation 16 is heavily impacted by the total selectivity. The fraction $\frac{BW_S \cdot C_M}{b}$ gives how many index nodes can be read sequentially at the time of an LLC miss, while the remaining part depends largely on the layout of the index and the result set (and on the relative performance when scanning data, traversing the tree, and writing results). Lastly, the denominator is the only place that we see the tuple size ts. This, in turn, means that having a larger tuple size (that is, having a row store, or a column group storage) lowers the overall value of the ratio indicating a more useful index. In addition to that, the denominator includes the cost of predicate evaluation, which will make the scan more expensive when it is larger than the data movement, which is the case as concurrency q increases.

For small values of q the second part of the numerator dominates, and as a result the dynamic parameter for the decision between scan and index is the total selectivity S_{tot}. On the other hand, as the number of concurrent queries increases (which is an observed workload trend [69]), the first part of the numerator and the predicate evaluation part of the denominator in Equation 16 dominate, hence increasing the significance of query concurrency. Another way to view the access path selection model in Equation 16 is consider what happens as the data set size increases. For larger N the impact of q is smaller, however, the observed trends in workloads and data sets is that both q and N would increase hence keeping the concurrency as one of the decisive factors of the comparison between shared scans and concurrent index accesses.

Model Verification. We verify the accuracy of the model using experimental data from four machines. For each set of experiments, we used a multidimensional unconstrained nonlinear minimization technique (Nelder-Mead) to fit the model. More details about the verification process are available in Appendix C.

2.5 Analysis of the APS Model

We now proceed with a detailed analysis that sheds light on additional aspects of access path selection and we summarize the findings in a number of observations. We study several cases based on the model for varying concurrency, selectivity, data size, tuple size, and hardware characteristics.

The parameterization of the workload, dataset, hardware, and secondary index tuning parameters, is based on our experimental setup (see §4). We parameterize the model to match our primary experimental server using $C_M = 180$ns, $C_A = 2$ns, $BW_S = 40$GB/s, $BW_I = 20$GB/s, and $BW_R = 20$GB/s. We denote this hardware as configuration $HW1$. We further model an alternate hardware configuration, $HW2$, with $C_M = 100$ns, and bandwidth $BW_S = 160$GB/s, $BW_I = 80$GB/s, and $BW_R = 80$GB/s. We study the impact

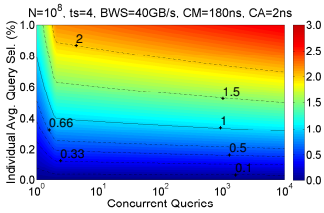

Figure 4: Access path selection is critical as either sequential scans or index scans can be the correct choice. The decision depends on query concurrency.

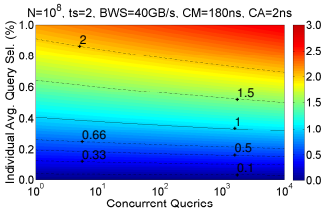

Figure 5: Access path selection is required when compression is employed ($ts = 2$). The points are different but there is still a choice to be made.

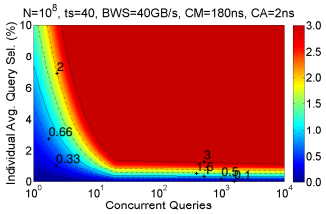

Figure 6: As we go from a single column (Figure 4) to column-groups it becomes even more important to support two access paths and selection.

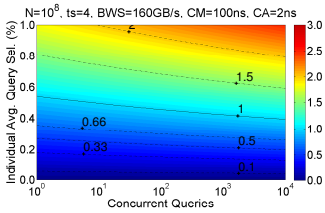

Figure 7: In the alternate setup $HW2$ (latency 100ns, bandwidth 160GB/s) access page selection is critical.

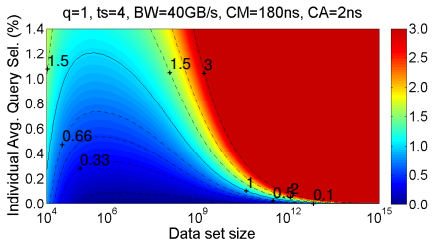

Figure 8: When selecting access path for a single query, the index is preferable when selectivity drops below $0.5-1\%$.

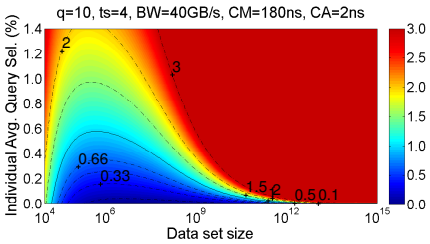

Figure 9: As we increase concurrency the cases that index is preferable correspond to smaller query selectivity.

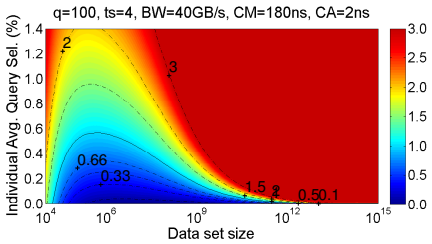

Figure 10: Even for larger levels of concurrency there is still a case for access path selection.

of relation size (varying N between 10^4 and 10^{15} tuples), varying ts between 4 bytes for 1 column and 40 for 10 columns. We further study compression, which effectively means to consider ts less than 4 bytes (we test with widths down to 2 bytes). The values for rw, aw, and ow are all equal to four bytes. We set the fanout of the tree to be $b = 21$ matching our experimentation to find a memory optimized B$^+$-Tree fanout on our machine.

Interpreting the Model Figures. The results are shown in Figures 4 through 10. Each figure shows a three dimensional plot of the APS ratio from 90 degrees, effectively showing a 2D projection of the plot with the color-map indicating the value at each point of the graph, and contour lines to better explain the graph. In Figures 4 through 7, the x-axis shows the query concurrency q, and the y-axis the average individual query selectivity (s_i). The x-axis of Figures 8 through 10 is the relation size N, and the y-axis remains the same. The APS ratio can take positive values between ≈ 0 and $\gg 1$, however, the interesting range of values is close to 1, where the decision between using a shared scan or a concurrent secondary index access is made. As a result, in each figure when $APS > 3$ we use the same color; the differentiation of the color is focused in the interesting range $[0,3]$. Dark blue areas have very low APS ratio and indicate cases that using a secondary index would be greatly beneficial, light blue areas correspond to a workload that a secondary index would provide a speedup of $2-3\times$, while turquoise areas correspond to the workload that index and scan are almost on par. Yellow areas the scan is $1.5-2\times$ faster, and red and especially dark red areas correspond to cases that using shared scanning is clearly preferred (more than $3\times$ difference).

We start our discussion with Figure 4. Here we compare scan and secondary index for a pure columnar layout using the values for our primary experimental server. It is interesting to see how fast the break-even point changes and the impact of making the wrong decision. For readability we plot a solid line where $APS = 1$ and dashed lines for $APS = \{0.1, 0.33, 0.5, 0.66, 1, 1.5, 2, 3\}$. When the ratio $APS = 0.66$, $\frac{ConcIndex}{SharedScan} = \frac{2}{3} \Rightarrow SharedScan = 1.5 \cdot ConcIndex$. The inverse is when $APS = 1.5$, $\frac{ConcIndex}{SharedScan} = 1.5 \Rightarrow ConcIndex = 1.5 \cdot SharedScan$. In these two cases, making the wrong decision would cost 50% performance loss. Similarly, making the wrong decision for the pair $\{0.5, 2\}$ corresponds to a $2\times$ slowdown (for example choosing index incorrectly would follow the 2 line in the dark red of any of the figures), and $\{0.33, 3\}$ a $3\times$ slowdown.

Observation 2.1. *In a modern main-memory optimized analytic data system there is a break-even point regarding the performance of a shared scan and a shared index scan; access path selection is needed to maximize performance in all scenarios.*

The shape of each line in Figure 4 depends on query concurrency. As we process more queries concurrently, the access path selection point changes. In other words, the access path selection depends – not only on selectivity, hardware, and data layout – but also on the concurrency of the workload. As concurrency increases, a secondary index is beneficial for lower individual query selectivity of the batched queries. On the other hand, when concurrency is very high, a shared scan is preferable. In fact there is a maximum point of concurrency beyond which a scan is always the right choice (we show in the experiments that this point is not achievable with current hardware but the potential remains). Compression (Figure 5) leads to similar behavior. The main difference is that scans are somewhat more beneficial even for concurrency as low as one query, however, access path selection is still required.

Observation 2.2. *Unlike traditional query optimization decisions, choosing between a sequential scan and an index scan in a modern system depends on concurrency in addition to selectivity.*

So far we have considered a storage model of one column at a time. However, many modern systems follow hybrid storage models to match the workload and access patterns exactly [4, 19, 25, 33]. We continue the analysis by varying tuple size which models (a) hybrid systems where data is stored as column-groups and (b) data compression effects, i.e., even if data is stored one column at a time compression levels may vary, affecting access path selection.

Figure 6 shows the impact of a different physical organization where query processing is performed over a group of ten columns, having $ts = 40$ bytes (instead of one column at a time in Figure 4). The observations remain similar to the previous analysis; the difference now is that the sweet spot has moved to both higher concurrency and lower selectivity making access path selection even

more crucial. For example, while $0.4 - 0.8\%$ selectivity is the maximum threshold for single column-at-time storage in Figure 4, now it climbs to 8% for a single query, but drops towards 0.4% as concurrency increases. Overall for larger tuple sizes due to hybrid storage (or due to limited opportunities for compression), indexing becomes more useful; however there is always a use case for both access paths.

Observation 2.3. *In hybrid systems supporting column-groups, secondary indexes are useful in more cases than plain column-stores because using a secondary index helps to avoid moving a larger amount of unnecessary data through the memory hierarchy.*

In Figure 7 we repeat the same analysis as in Figure 4 but this time using the alternate hardware configuration $HW2$. Changing the cache access cost and main memory access cost changes the points at which we change from sequential scan to secondary index scan, making the system slightly more suitable for secondary indexes. Even though the actual points are slightly different, overall the shape of the curves and the high-level observation remains very similar; all systems require access path selection and concurrency plays a major role.

Observation 2.4. *Although hardware characteristics change the point where a sequential scan is preferred over a secondary index scan, all systems require run time analysis to make the decision.*

Figure 8 shows how the break-even point changes as we vary the data size between 10^4 and 10^{15} tuples. A first observation is that the sorting overhead makes the secondary index less useful as scan outperforms secondary indexes for lower selectivity. In addition to that, we still observe the impact of query concurrency. Figures 9 and 10 show that with higher concurrency there is a small data size (less than 10^5 tuples) that scan will always be better, however, we cannot keep increasing the level of concurrency arbitrarily. Shared scans of tens or hundreds of queries are possible, but as we move past 256 or more concurrent queries, hardware resources like L1 cache and TLB get thrashed and performance drops. Overall, for data set sizes between 10^4 and 10^{15} tuples a decision between a shared sequential scan and a secondary index scan should be made, and the decision point depends on the query concurrency.

Observation 2.5. *Access path selection becomes more crucial for bigger data inputs as data movement becomes more expensive – and as a result every wrong decision has a larger cost.*

Putting Everything Together. Modern data systems can benefit from multiple access paths. The decision between a secondary index and a scan depends on a number of factors: query selectivity, hardware properties (cache and memory latency, memory bandwidth), as well as *query concurrency*.

3. INTEGRATING COST-BASED APS

Having demonstrated that more than a single access path may be useful in modern systems and having built a model to make the dynamic decision of access path selection, we now move forward to discuss how to integrate this technology in modern main-memory optimized analytical systems. We explain how integration should be done and its side-effects using our storage and execution engine, FastColumns. FastColumns includes all the data layout and and access methods described in the previous sections including multiple data layouts: single column, column-groups, and full tuples as well as secondary tree indexes.

New Components. The key component we introduce in order to integrate access path selection is the new cost-based optimizer which implements the model described in the previous section to perform run time access path selection. The scheduler is another key component; it continuously collects incoming queries and batches queries that refer to the same data to prepare them for analysis by the optimizer. Several other existing components must collect the right statistics. These statistics are likely already collected in most systems. Overall, the integration to a modern analytics system is a minor change that does not include drastic changes in the architecture. This is shown visually in Figure 11 where we highlight the new APS module that needs to be part of the optimizer; the rest of the changes are about ensuring the system has the right statistics.

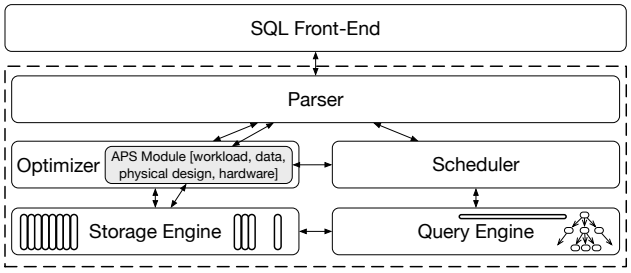

Figure 11: Integrating cost-based access path selection.

Continuous Data Collection. The system needs to monitor query concurrency, query selectivity, and dataset properties. It also must take into account hardware characteristics (which are tuned once per machine during initial setup). The optimizer uses this information to perform access path selection (for q concurrent queries).

To get the required information the optimizer communicates with both the storage engine, and the scheduler. In particular, the information with regards to the data properties (size and physical organization) is available from the storage engine, while the query concurrency and selectivity are properties known to the scheduler, which receives the workload from the front-end of the system. The remaining information regarding hardware specification is collected when the system is initialized on a new machine. This process can be automated. We use Intel's Memory Latency Checker tool [39] which allows us to document the memory latency and bandwidth across the NUMA nodes of the system.

Choosing Access Paths. In addition, the optimizer performs further analysis on the current batch of queries and the data they reference to determine the additional properties required for access path selection. These properties include the number of outstanding queries, the number of overlapping ranges, their total selectivity as well as their referenced data size and tuple size (column-group layout). The total query selectivity for each batch is estimated using the individual expected query selectivity.

Using the data described in the previous two paragraphs, the optimizer can decide during execution whether to use a secondary index or a scan for a batch of queries by running Equation 15 (APS) and checking whether the value is smaller than one (use the secondary index) or greater than one (use a sequential scan).

Fast Decisions. In a modern main-memory optimized analytical system it is critical to perform access path selection quickly. The reason is that response times in analytical systems are typically quite low, in many cases performance is sub-second (in part because of the advanced access paths, and in part because data is memory resident in many cases). As such a cost-based optimizer in a modern analytical system needs to take special care to ensure that optimization time does not become a critical component of the total cost. Our solution keeps the costs of data collection and runtime calculations low. Access path selection is in fact a small number of instructions, just enough to calculate the APS value. The number of outstanding queries, and their selectivity, are held and maintained in a similar way to other statistics, a simple count per attribute in

the case of outstanding queries, and using standard selectivity estimation techniques (like histograms). Even for queries that cost only a couple of seconds, APS can be evaluated very quickly (on our test machine the decision takes on the order of microseconds).

Error Propagation. The problem of choosing the wrong access path and error propagation has been studied extensively for traditional systems [23]. The new access path selection paradigm we propose in this paper takes a plethora of information into account, much more than traditional optimizers. However, it does not make the problem (or solutions) any more complex as all the additional information is information that the optimizer acquires either at run time in a precise way (the number of outstanding queries, and the total query selectivity which is based on the statistics that are already kept), or is information that it acquires at initialization time like hardware properties which is static and always accurate data.

4. EXPERIMENTAL ANALYSIS

In this section we experimentally demonstrate that when putting all modern access path optimizations together, there is still a need for access path selection in modern analytical systems. We show that unlike existing systems that make the decision based on a fixed selectivity threshold, the optimizer of a modern analytical system now needs to account for the number of concurrent queries with overlapping attributes. We also validate the access path selection model, demonstrating that it allows the optimizer to make fast real-time decisions in a variety of scenarios.

Hardware and Operating System Specifications. Our experimental infrastructure consists of a NUMA machine with 4 sockets, each equipped with an Intel Xeon E7-4820 v2 Ivy Bridge processor running at 2.0GHz with 16MB of L3 cache. Each processor has 8 cores and supports hyper-threading for a total of 64 hardware threads. The machine includes 1TB of DDR3 (@1066MHz) main memory, evenly distributed across the sockets, and four 300GB 15K RPM disks configured in a RAID-5 array. We run 64-bit Debian "Jessie" version 8.6 on Linux 3.16.7. We also conduct experimentation on alternate hardware, using Amazon EC2 dedicated instances (details are described in the relevant experiment).

Experimental Methodology. For each experiment described in this section we use a synthetic data set of uniformly distributed 32-bit integers. We vary the data size between 100 million and 500 million tuples. The workload is a sequence of select-based queries with variable selectivity (as noted in the individual experiments) for which we report the time measured to execute each select operator. We vary the query concurrency between a single query and 512 queries. The costs include result materialization and sorting in rowID order in the case of secondary index scan. The data points reported are each the arithmetic mean of ten trials, with standard deviation within 3%. Experiments are strictly main-memory resident; data is loaded if necessary prior to any measurements and there is no disk I/O involved. Faster data reading favors scans, which always read all the data. It also means that that access path selection must be done quickly.

Implementation. All experiments are performed in the FastColumns storage and execution engine described in Section 3. There is no SQL front-end for FastColumns yet so the SQL parsing cost is excluded (for all experiments); all queries are described in a domain specific language which maps to the logical plan of the query.

No Single Access Path Always Performs Best. We begin our analysis by comparing the latency of a single sequential scan that uses SIMD and multiple hardware threads to a multithreaded index scan. We introduce concurrency in the next experiment. The goal here is to show that access path selection is needed even on a fully tuned system while executing a single query. This is the case in modern analytical systems that do not yet have support for shared scans.

Figure 12 shows the latency of each access method on a 100 million tuple input as we vary query selectivity between 0% and 100% along the x-axis. We generate two fit lines based on the data in the low selectivity points to find the point where the performance preference shifts from one access method to the other. Although the crossover is at a low percent, the performance difference is significant. The query using the secondary index can run many times faster than the scan which makes for a substantial difference if we consider total workload performance, i.e., for thousands (or more) queries. The result size in Figure 12 is in the order of several million keys. Hence, the secondary index is useful when the result cardinality is anywhere between 0 and 1.5 million keys (1.5% of 100 million values of the input data).

The Influence of Concurrency. We now introduce concurrency into our analysis. This means that both sequential scan and secondary index scan can use shared execution to amortize the data movement cost in addition to the other optimizations. In this experiment, we repeat the same experiment as in Figure 12, with the difference being that we vary the number of concurrent queries selecting over the same data. We repeat the experiment varying concurrency between 1 and 512 queries.

Figure 13 shows the results. Each point in Figure 13 represents the crossover point selectivity observed for this level of concurrency (x-axis). That is, each plot point is the result of one trial of Figure 12 with the concurrency varied. In this way, this graph is parsed in the same way as Figure 1 in the Introduction: any query batch of q queries with aggregate selectivity above the one shown in Figure 13 for q should use a scan, otherwise the optimizer should use an index scan. As concurrency varies in Figure 13 the crossover point also moves. With more concurrent queries, scan becomes more useful because queries can amortize the cost of reading the entire column. The index scan also benefits from sharing, however, not as dramatically as sequential scan. Traversing the leaves of a tree is more expensive when compared to a contiguous array. Nevertheless, the crossover point plateaus eventually, that is, there is always a case for access path selection. In Section 2.4 we explained that sharing only works up until the point where write costs overwhelm the benefits of shared execution. We can see this at the far right of Figure 13. The performance of shared scan deteriorates when attempting to share 512 selects simultaneously. However, the performance can be retained by batching the 512 queries into two runs of 256 (the point labeled "512-batch").

Observation 4.1. *Concurrency, the number of active queries that need to select over the same data, is a critical factor that affects access path selection.*

The Influence of Data Set Size. The next parameter we study is data size. We repeat the same experiment as in Figure 12 multiple times, varying the data size and as in Figure 13, we plot the crossover point for each run (data size).

The results are shown in Figure 14. Effectively, as the data size grows the crossover point rises to a higher selectivity reaching a maximum point before starting to drop gradually.

When scanning the input column, as we increase its size, the cost increases linearly ($O(N)$); the cost is proportional to the extra tuples the scan has to check and the corresponding increase in result set size. For a secondary index scan, there are three steps. First, as we increase the size of the input column the tree becomes higher logarithmically ($O(log(N))$). Second, the number of qualifying tuples in a given range, which affects both the number of leaves traversed and the result set size, increases linearly ($O(N)$). Third, the cost

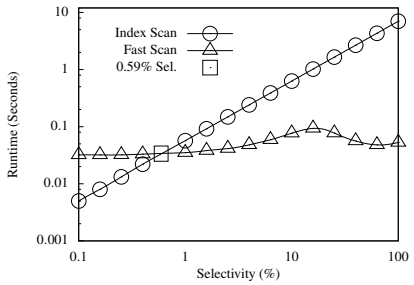

Figure 12: There exists a crossover point for access path selection in analytical systems even when $q = 1$.

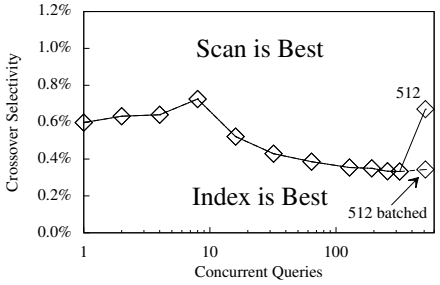

Figure 13: The number of concurrent queries is a critical component of access path selection in analytical data systems.

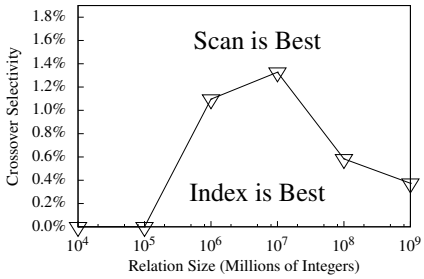

Figure 14: The crossover point is also affected by the data set size ($q = 8$).

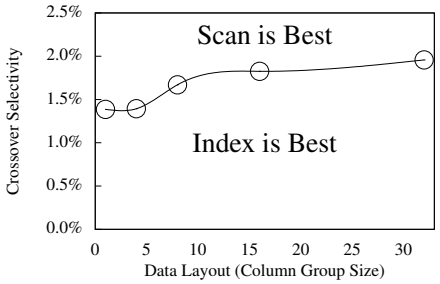

Figure 15: Scans with strided accesses are less efficient increasing the opportunities where an index scan is beneficial.

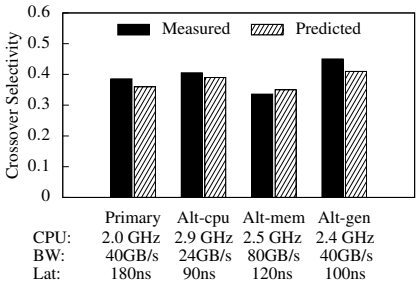

Figure 16: FastColumns is able to accurately predict the crossover point for different hardware configurations.

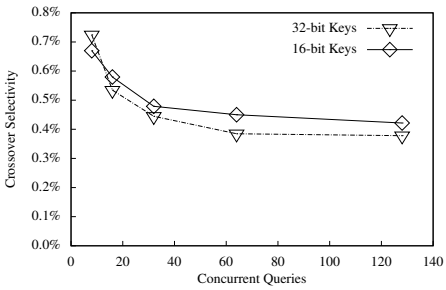

Figure 17: Working directly over compressed data gives a slight advantage for scans.

of sorting the result of the index scan (so it is in row-id order) increases following $O(N \cdot log(N))$ complexity.

Observation 4.2. *As the data set size increases, the selectivity crossover point reaches a maximum and then drops gradually.*

The Influence of Data Layout. In our next experiment we demonstrate the effect of data layout and that access path selection is also needed in analytical systems that support hybrid layouts. We repeat the experiment of Figure 12 but vary the number of columns forming a column-group in the data layout. Effectively we test for pure column-store layout and increasingly bigger column-group hybrid layouts (up to 32 columns).

Figure 15 shows the results. It is the same plot style as Figures 13 and 14; each point represents the crossover selectivity point for the respective data layout on the *x*-axis. As the column-group size increases towards hybrid layouts, secondary index scan becomes increasingly beneficial in a wider range of selectivities. This is because similarly to past row-store systems, the secondary index allows for increasing data skipping capabilities over wider column-groups that have to touch multiple columns at the same time, even when scanning a single column.

Observation 4.3. *Given the trend towards hybrid layouts, we expect access path selection to become increasingly important.*

The Influence of Hardware Properties. In our next experiment we show that our access path selection model works with significantly different hardware as well with different memory bandwidth and cache properties. We repeat the same experiment presented in Figure 12, but this time we use three Amazon EC2 instances: 1) a general purpose instance (m4.4xlarge) with 16 virtual cores (Intel Xeon E5-2676 v3) and 64GB of RAM; 2) a compute instance (c4.8xlarge) with 36 virtual cores (Intel Xeon E5-2666 v3) and 60GB of RAM; and 3) a memory instance (r3.8xlarge) with 32 virtual cores (Intel Xeon E5-2670 v2) and 244GB of RAM.

Figure 16 shows the results. The graph plots the crossover points for access path selection. The main observation is that in all four different hardware configurations there is always a crossover point between the access methods; we always need access path selection for optimal performance regardless of the hardware configuration. In addition, the crossover point is different across the various hardware instances, meaning that in order to perform effective access path selection the optimizer of a system should be able to capture the properties of its run time environment. Figure 16 shows that our access path selection model can indeed capture these environment properties. The APS model bars closely follow the experimentally measured crossover point which means that our model can accurately predict the point where we should switch access path on all hardware configurations.

Observation 4.4. *Our optimizer is not specially tuned for a single hardware configuration. Rather it works by isolating the parts of the system that are important to predicting performance and calculating the preferred access path.*

The Influence of Compression. Figure 17 shows how the crossover point is affected when we employ compression. The setting of this experiment is the same as before with the only difference that the 16-bit line corresponds to a scan over a compressed column (to 2 bytes). This allows the scan to consume more tuples with the same SIMD commands, but slightly increases the overhead of writing the result. Overall, compression does not change the main message; rather, it moves the crossover slightly in favor of scans.

Observation 4.5. *Employing dictionary compression gives a small advantage for scans, however, both access paths remain useful.*

Empirically Validating Access Path Selection. So far we have studied the parameters that affect access path selection in isolation and we have shown that they should be taken into account. In our next experiment, we test the cost based optimizer across various workloads to demonstrate that it can make dynamic and effective access path selection decisions.

We set-up this experiment as follows. We create nine distinct workloads with variable properties. In particular, the nine workloads include variable selectivity and different levels of concurrency: low, medium, and high. Low selectivity is a point get,

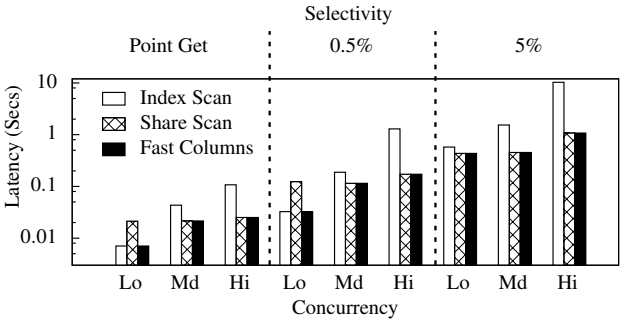

Figure 18: The workload characteristics change the optimal data access path. APS is needed to choose the fastest.

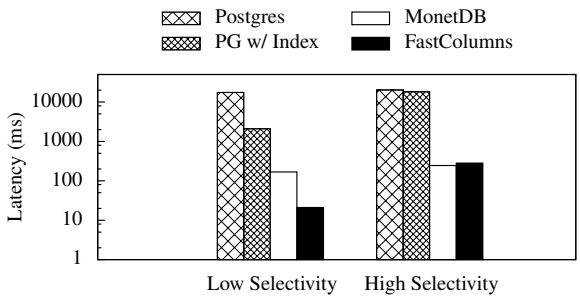

Figure 19: TPC-H Scale Factor 10: Access path selection brings optimal performance in diverse workloads.

medium is 0.5% of the relation, and high is 5%. For concurrency, a single query represents low, 64 queries is chosen for medium, and 640 queries is high concurrency. We create the nine workloads by making all possible combinations of selectivity and query concurrency. We test both secondary index scans and sequential scans. We compare those against the predictions of the optimizer which evaluated each access method and dynamically decides which one to use instead of using a fixed access method or existing strategies that do not consider concurrency.

Figure 18 shows the results. None of the individual access methods is best across all workloads. Instead, each access method is best only for a subset of the cases. On the other hand, our APS model can always match the best behavior by picking the correct access method dynamically at run time. For example, for the case of a single query interested in less than 0.5% of the relation, the model correctly chooses the index. As concurrency increases, the shared scan approach becomes the better choice for point gets and our solution makes the right choice. FastColumns also correctly chooses a shared scan for medium selectivity with medium or high concurrency, and for all high selectivity workloads. In order to make the correct access path decisions, the optimizer considers static influences (like LLC miss cost, memory bandwidth, etc.), as well as dynamic ones (number of in-flight queries, query selectivity).

Observation 4.6. *Accurate and dynamic access path selection can be performed by taking into account estimated selectivity, concurrency, data size, and hardware properties.*

TPC-H and Full Query Sanity Check. We compare FastColumns with two known and mature open-source implementations: MonetDB and PostrgreSQL. MonetDB is highly tuned for main-memory performance [37] and includes scans with tight `for` loops, columnar storage, multi-core optimizations and has practically zero functional overhead. Since MonetDB does not use shared scans, we test with a single query (but use multiple hardware threads) with the data resident in main-memory. We also include Postgres which has a mature B$^+$-Tree design.

The experiment runs on data generated for the TPC-H benchmark. We generated the `lineitem` table with a scale factor 10 which results in just under 60 million rows. We then modified TPC-H query 6 to form two types of workload, high and low selectivity, by choosing a larger or smaller range for the `l_shipdate` predicate. In the "low selectivity" run approximately 174 thousand rows qualify after applying the `l_shipdate` predicate, or 0.24% of the relation. In the "high selectivity" run approximately 15% of the relation qualifies based on `l_shipdate`.

Figure 19 shows the results. FastColumns chooses to process the query with a scan when greater than 1.6% of the data qualifies while it uses a secondary index scan when less of the data qualifies. The first observation from Figure 19 is that FastColumns provides optimized main-memory access paths since it is comparable to MonetDB in sequential scan performance. This works as a good sanity check for our analysis. More crucially, the next observation is that FastColumns can outperform MonetDB when only a small number of tuples qualify by using access path selection and a secondary index instead of relying on a scan. This reinforces the statement that modern analytical systems like MonetDB and other highly tuned systems should consider secondary indexes and access path selection for such workloads.

The results for Postgres are mainly used as a validation that fast scans have significantly changed the picture. Even without secondary indexing MonetDB outperforms Postgres for the queries where indexing is expected to help.

5. LESSONS LEARNED

The modeling and the performance analysis presented in this paper offers several insights as to how to design secondary indexing in today's analytical systems. Using our access path selection model we compute the crossover point over several years of hardware evolution. Table 2 shows how the crossover point in access path selection evolved from the 1980's to 2010's for HDD-based systems. We model the evolution of hard drives by using decreasing latency and increasing disk bandwidth based on past devices reported performance [8]. We further assume that the data size (in number of tuples) has been increasing roughly exponentially with time. Finally, as a representative tuple size we use 200 bytes (representative numbers from TPC-H) and as branching factor for the modeled index we use 250 [30].

The main observation from Table 2 is that as bandwidth improves, scans become useful in more cases and the crossover point decreases. The crossover point in Table 2 for 2010, corresponds to a disk-based column-store where using a disk-based secondary index has naturally less impact than in row-stores as a scan can read only the relevant column anyway. Recall the term $BW \cdot C_M$ from Section 2.4 which corresponds to the number of bytes read during a random access (traversing the tree). In a main memory system, this product is on the order of 7200 (using the 2016 column from Table 2), however, when we use disk bandwidth and latency the corresponding product is on the order of 10^6 (using the numbers from the 2010 column). The model predicts a crossover point in both cases, and in fact, main memory systems shift the balance back towards the index due to the more efficient random access in memory (with respect to available bandwidth), when comparing with disk resident indexes and columns. We also show two potential configurations, F1 and F2, for which the hardware has changed to either support much higher memory bandwidth, with modest decrease in latency (for F1), or much lower memory latency with modest increase in bandwidth (for F2). While new hardware puts pressure on indexes, we still need access path selection.

In a traditional row-store a fixed selectivity is the single run-time parameter used to choose an access path, after the hardware config-

Year	1980 (disk)	1990 (disk)	2000 (disk)	2010 (disk)	**2016 (mem)**	**F1 (mem)**	F2 **(mem)**
CPU (GHz)	–	–	–	–	2	4	4
HDD (ms)	10	8	2	2	–	–	–
HDD (MB/s)	40	100	500	500	–	–	–
Mem (ns)	–	–	–	–	180	100	20
Mem (GB/s)	–	–	–	–	40	160	80
# tuples	10^6	10^7	10^8	10^9	10^9	10^9	10^9
Tuple Size	200	200	200	4	4	4	4
Branching	250	250	250	250	21	21	21
Crossover	12.4%	6.2%	5.0%	0.1%	**0.6%**	**0.3%**	**0.5%**

Table 2: *Access path selection crossover point evolution.*

uration has been statically considered. Every query that is expected to return less than the threshold uses a secondary index if available. As the selectivity window that favors the index gets smaller, it becomes increasingly important to accurately account for all parameters during query execution: hardware, selectivity, system design, and now *query concurrency*.

We now summarize the lessons learned from this study:

Lesson 1. *There are cases for analytical queries when a secondary index scan is preferable to a sequential scan, even when data is main-memory resident which removes the disk I/O bottleneck.*

Lesson 2. *A fine-grained access path decision needs to take into account as a run-time parameter query concurrency, in addition to selectivity, hardware specs, system design, and data layout.*

Lesson 3. *Data set size can be pivotal. For small data sets scanning the data outperforms secondary indexes in all cases. Index remains useful for larger data sets as full attribute scans become costly.*

Lesson 4. *While the crossover between access methods is lower than in the past, it still corresponds to a growing result cardinality. For example, a query that selects* 0.6% *of 500 million integers, has 3 million qualifying tuples.*

Lesson 5. *While sharing data access minimizes repetitive reads, it includes the overhead to distribute the results to their consumers. Result sharing is efficient up to a batch size, because of the bookkeeping and the result distributing overhead.*

Lesson 6. *Access path selection is important for systems with columnar-storage, however, it becomes even more important for wider tuples in hybrid stores.*

Lesson 7. *As the cache and memory latency decreases, or the memory bandwidth decreases, secondary indexes become more beneficial. On the contrary, slower caches or memory, and faster memory buses benefit scan. In this way, future hardware generations that affect the balance between these hardware properties will also affect access path selection accordingly. These properties are captured by the APS model.*

6. RELATED WORK

Access Path Selection. Access path selection has been a primary design point since the very beginning of database systems [20, 70]. The primary decision is whether to use a sequential scan or a secondary index scan, and is based on statistics and a given threshold of selectivity which is a hard threshold either hard-coded in the system or decided during the tuning phase [29, 32]. More recently cost-model analysis has included hardware characteristics [16, 56].

In this paper, we follow the basic principles of access path selection for sequential scans and secondary index scans but for modern main-memory optimized analytical systems that employ access methods fully optimized for modern hardware and sharing. We build on past modeling approaches and we augment them by taking into account query concurrency. We show that, in addition to considering selectivity, hardware characteristics, and data storage (column-store, hybrid storage, or full tuples) like traditional optimizers do, we need to also take into account *query concurrency*.

Modern Optimizers and Designers. Modern systems use cost-based optimization for various decisions. For example, cost-based optimization based on statistics is used for select and join ordering [79]. Furthermore, offline physical design in modern systems uses cost-based analysis to perform decisions, e.g., about storage layouts [67] and column-store projections [66].

Our work in this paper is complementary to other cost-based decisions performed by modern optimizers. For example, join ordering and select ordering in terms of expected selectivity can work independent of the access path selection used. In addition, our work is also complementary to works on physical design as we have developed a methodology for runtime access path selection regardless of the underlying column (group) layout and global physical design. However, similar to how traditional physical design tools use optimizers (i.e., their cost model) during offline analysis [22], the APS model we present can be used by physical design tools to decide whether to create secondary indexes or not.

Sharing and Multi-Query Optimization. Multi-query optimization approaches have considered opportunities for work sharing across multiple queries [36, 68, 71, 72]. This mainly targets work that shares execution. More recently, sharing data movement has been considered [27, 28, 34, 41, 62, 63, 88], even in memory [63]. Our work is relevant to work on sharing execution and data movement in that we consider access paths that rely on sharing. Past work has mainly focused on developing new techniques for how sharing should be done. Here we present how to make an optimization decision between different access paths that all use sharing.

Delaying Optimization Decisions. While in past systems data flows primarily one tuple-at-a-time, modern analytical systems have experimented both with vector-at-a-time processing and column-at-a-time processing. A side-effect of column-at-a-time processing is that physical operator choice can be done at the very last minute once we have have full information about data input cardinality and other properties. For example, MonetDB uses column-at-a-time processing and decides the join algorithm and the inner/outer sides with full information on the input sizes [37]. Such approaches require fewer decision points during optimization. Similarly ideas like Smooth Scan [17] (that have only been studied in row-stores) delay access path decisions or provide hybrid access paths.

However, we still need access path selection the way it is described here. For example, when it comes to select operators (e.g., sequential scan versus secondary index scan) which touch base data, we have to make a choice before we start regardless of the data flow pattern. Whenever there are multiple access paths, the choice must be made before the data can be read.

7. CONCLUSIONS

In this paper, we demonstrate that access path selection is still required, since both secondary index and sequential scan accesses can prove beneficial for analytics. As more in-memory access paths are developed, access path modeling and selection remains a key challenge in modern analytical data systems.

We show that contrary to traditional approaches, access path selection should not use a fixed pivot point based only on machine hardware and selectivity. Instead, in order to decide the access path for a batch of queries, a system should take into account the following two dynamic parameters: (1) the selectivity of each query from the batch and (2) the number of concurrent queries. Finally, we demonstrate that the APS is a light-weight process that does not negatively affect the execution time even of low-latency queries.

Acknowledgments. We thank the reviewers for their valuable feedback. This work is supported by the National Science Foundation under grant IIS-1452595.

8. REFERENCES

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

[2] D. J. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *SIGMOD*, 2006.

[3] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization Strategies in a Column-Oriented DBMS. In *ICDE*, 2007.

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

[5] J. R. Allen, K. Kennedy, C. Porterfield, and J. D. Warren. Conversion of Control Dependence to Data Dependence. In *POPL*, 1983.

[6] G. Antoshenkov. Dynamic Query Optimization in Rdb/VMS. In *ICDE*, 1993.

[7] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez. The DataPath System: A Data-centric Analytic Processing Engine for Large Data Warehouses. In *SIGMOD*, 2010.

[8] M. Athanassoulis. *Solid-State Storage and Work Sharing for Efficient Scaleup Data Analytics*. PhD thesis, Ecole Polytechnique Federale de Lausanne, 2014.

[9] M. Athanassoulis, M. S. Kester, L. M. Maas, R. Stoica, S. Idreos, A. Ailamaki, and M. Callaghan. Designing Access Methods: The RUM Conjecture. In *EDBT*, 2016.

[10] M. Athanassoulis, Z. Yan, and S. Idreos. UpBit: Scalable In-Memory Updatable Bitmap Indexing. In *SIGMOD*, 2016.

[11] D. I. August, W.-m. W. Hwu, and S. A. Mahlke. A Framework for Balancing Control Flow and Predication. In *MICRO*, 1997.

[12] R. Barber, P. Bendel, M. Czech, O. Draese, F. Ho, N. Hrle, S. Idreos, M.-S. Kim, O. Koeth, J.-G. Lee, T. T. Li, G. M. Lohman, K. Morfonios, R. Müller, K. Murthy, I. Pandis, L. Qiao, V. Raman, R. Sidle, K. Stolze, and S. Szabo. Business Analytics in (a) Blink. *IEEE DEBULL*, 35(1):9–14, 2012.

[13] R. Barber, G. Lohman, V. Raman, R. Sidle, S. Lightstone, and B. Schiefer. In-Memory BLU Acceleration in IBM's DB2 and dashDB: Optimized for Modern Workloads and Hardware Architectures. In *ICDE*, 2015.

[14] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based Order-preserving String Compression for Main Memory Column Stores. In *SIGMOD*, 2009.

[15] P. Boncz, M. Zukowski, and N. J. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*, 2005.

[16] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the Memory Wall in MonetDB. *CACM*, 51(12):77–85, 2008.

[17] R. Borovica-Gajic, S. Idreos, A. Ailamaki, M. Zukowski, and C. Fraser. Smooth Scan: Statistics-Oblivious Access Paths. In *ICDE*, 2015.

[18] G. Candea, N. Polyzotis, and R. Vingralek. Predictable Performance and High Query Concurrency for Data Analytics. *VLDBJ*, 20(2):227–248, 2011.

[19] C. Chasseur and J. M. Patel. Design and Evaluation of Storage Organizations for Read-Optimized Main Memory Databases. *PVLDB*, 6(13):1474–1485, 2013.

[20] S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *PODS*, 1998.

[21] S. Chaudhuri. Query Optimizers: Time to Rethink the Contract? In *SIGMOD*, 2009.

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

[23] S. Christodoulakis. Implications of Certain Assumptions in Database Performance Evaluation. *TODS*, 9(2):163–186, 1984.

[24] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood. Implementation Techniques for Main Memory Database Systems. In *SIGMOD*, 1984.

[25] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA Database – An Architecture Overview. *IEEE DEBULL*, 35(1):28–33, 2012.

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

[27] G. Giannikis, G. Alonso, and D. Kossmann. SharedDB: Killing One Thousand Queries with One Stone. *PVLDB*, 5(6):526–537, 2012.

[28] G. Giannikis, D. Makreshanski, G. Alonso, and D. Kossmann. Shared Workload Optimization. *PVLDB*, 7(6):429–440, 2014.

[29] G. Graefe. The Five-Minute Rule Twenty Years Later. In *DAMON*, 2007.

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

[31] G. Graefe and L. D. Shapiro. Data Compression and Database Performance. In *SAC*, 1991.

[32] J. Gray and F. Putzolu. The 5 Minute Rule for Trading Memory for Disc Accesses and the 5 Byte Rule for Trading Memory for CPU Time. *Tandem Computers - Technical Report*, 1986.

[33] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE: A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2):105–116, 2010.

[34] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: A Simultaneously Pipelined Relational Query Engine. In *SIGMOD*, 2005.

[35] S. Héman, M. Zukowski, and N. J. Nes. Positional Update Handling in Column Stores. In *SIGMOD*, 2010.

[36] M. Hong, M. Riedewald, C. Koch, J. Gehrke, and A. Demers. Rule-based Multi-Query Optimization. In *EDBT*, 2009.

[37] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE DEBULL*, 35(1):40–45, 2012.

[38] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing Tuple Reconstruction in Column-Stores. In *SIGMOD*, 2009.

[39] Intel. Online reference. *https://software.intel.com/en-us/articles/intelr-memory-latency-checker*.

[40] T. Jäkel, H. Voigt, T. Kissinger, and W. Lehner. Pack Indexing for Time-Constrained In-Memory Query Processing. In *BTW*, 2013.

[41] R. Johnson, S. Harizopoulos, N. Hardavellas, K. Sabirli, I. Pandis, A. Ailamaki, N. G. Mancheril, and B. Falsafi. To Share or Not to Share? In *VLDB*, 2007.

[42] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise Parallel Predicate Evaluation. *PVLDB*, 1(1):622–634, 2008.

[43] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*, 2011.

[44] S. Khoshafian, G. P. Copeland, T. Jagodis, H. Boral, and P. Valduriez. A Query Processing Strategy for the Decomposed Storage Model. In *ICDE*, 1987.

[45] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *SIGMOD*, 2010.

[46] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. *KISS-Tree*: Smart Latch-Free In-Memory Indexing on Modern Architectures. In *DAMON*, 2012.

[47] J. Krüger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. *PVLDB*, 5(1):61–72, 2011.

[48] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T.-H. Lee, J. Loaiza, N. Macnaughton, V. Marwah, N. Mukherjee, A. Mullick, S. Muthulingam, V. Raja, M. Roth, E. Soylemez, and M. Zait. Oracle Database In-Memory: A Dual Format In-Memory Database. In *ICDE*, 2015.

[49] A. Lamb, M. Fuller, and R. Varadarajan. The Vertica Analytic Database: C-Store 7 Years Later. *PVLDB*, 5(12):1790–1801, 2012.

[50] P.-A. Larson, E. N. Hanson, and M. Zwilling. Evolving the Architecture of SQL Server for Modern Hardware Trends. In *ICDE*, 2015.

[51] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *SIGMOD*, 2014.

[52] V. Leis, A. Kemper, and T. Neumann. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *ICDE*, 2013.

[53] J. J. Levandoski, D. B. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE*, 2013.

[54] Y. Li, C. Chasseur, and J. M. Patel. A Padded Encoding Scheme to Accelerate Scans by Leveraging Skew. In *SIGMOD*, 2015.

[55] Y. Li and J. M. Patel. BitWeaving: Fast Scans for Main Memory Data Processing. In *SIGMOD*, 2013.

[56] S. Manegold, P. A. Boncz, and M. L. Kersten. Generic Database Cost Models for Hierarchical Memory Systems. In *VLDB*, 2002.

[57] Y. Mao, E. Kohler, and R. T. Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *EuroSys*, 2012.

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

[59] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[60] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD Vectorization for In-Memory Databases. In *SIGMOD*, 2015.

[61] O. Polychroniou and K. A. Ross. A Comprehensive Study of Main-memory Partitioning and its Application to Large-scale Comparison- and Radix-sort. In *SIGMOD*, 2014.

[62] I. Psaroudakis, M. Athanassoulis, and A. Ailamaki. Sharing Data and Work Across Concurrent Analytical Queries. *PVLDB*, 6(9):637–648, 2013.

[63] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory Scan Sharing for Multi-core CPUs. *PVLDB*, 1(1):610–621, 2008.

[64] V. Raman, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, L. Zhang, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, and S. Liu. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *PVLDB*, 6(11):1080–1091, 2013.

[65] J. Rao and K. A. Ross. Making B+-trees Cache Conscious in Main Memory. In *SIGMOD*, 2000.

[66] A. Rasin and S. Zdonik. An Automatic Physical Design Tool for Clustered Column-stores. In *EDBT*, 2013.

[67] P. Rösch, L. Dannecker, F. Färber, and G. Hackenbroich. A Storage Advisor for Hybrid-Store Databases. *PVLDB*, 5(12):1748–1758, 2012.

[68] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and Extensible

Algorithms for Multi Query Optimization. *SIGMOD Rec.*, 29(2):249–260, 2000.

[69] P. Russom. High-Performance Data Warehousing. *TDWI Best Practices Report*, 2012.

[70] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *SIGMOD*, 1979.

[71] T. K. Sellis. Multiple-Query Optimization. *TODS*, 13(1):23–52, 1988.

[72] T. K. Sellis and S. Ghosh. On the Multiple-query Optimization Problem. *TKDE*, 2(2):262–266, 1990.

[73] R. Sen, J. Chen, and N. Jimsheleishvilli. Query Optimization Time: The New Bottleneck in Real-time Analytics. In *IMDM*, 2015.

[74] A. Shahvarani and H.-A. Jacobsen. A Hybrid B+-tree as Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms. In *SIGMOD*, 2016.

[75] N. Shamgunov. The MemSQL In-Memory Database System. In *IMDM*, 2014.

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

[77] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. R. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: A Column-oriented DBMS. In *VLDB*, 2005.

[78] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained Partitioning for Aggressive Data Skipping. In *SIGMOD*, 2014.

[79] N. Tran, A. Lamb, L. Shrinivas, S. Bodagala, and J. Dave. The Vertica Query Optimizer: The Case for Specialized Query Optimizers. In *ICDE*, 2014.

[80] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable Performance for Unpredictable Workloads. *PVLDB*, 2(1):706–717, 2009.

[81] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *PVLDB*, 2(1):385–394, 2009.

[82] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and Rich Analytics at Scale. In *SIGMOD*, 2013.

[83] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *SIGMOD*, 2016.

[84] J. Zhou and K. A. Ross. Implementing Database Operations Using SIMD Instructions. In *SIGMOD*, 2002.

[85] M. Zukowski and P. Boncz. Vectorwise: Beyond Column Stores. *IEEE DEBULL*, 35(1):21–27, 2012.

[86] M. Zukowski, P. A. Boncz, and S. Héman. MonetDB/X100 - A DBMS In The CPU Cache. *IEEE DEBULL*, 28(2):17–22, 2005.

[87] M. Zukowski, S. Héman, and P. A. Boncz. Architecture-conscious Hashing. In *DAMON*, 2006.

[88] M. Zukowski, S. Héman, N. J. Nes, and P. Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *VLDB*, 2007.

[89] M. Zukowski, N. Nes, and P. A. Boncz. DSM vs. NSM: CPU Performance Tradeoffs in Block-oriented Query Processing. In *DAMON*, 2008.

APPENDIX

A. BOUNDS FOR SORTING COST

The last term of Equation 11 makes the equation unsolvable for total selectivity S_{tot}. Here, we first bound the maximum and the minimum values of the term $\sum_{i=1}^{q} N \cdot s_i \cdot log_2(s_i \cdot N)$ and then we use the worst case for our analysis. We use that $S_{tot} = \sum_{i=1}^{q} s_i$. Let us first expand the term:

$$\sum_{i=1}^{q} s_i \cdot N \cdot log_2(s_i \cdot N) =$$

$$= S_{tot} \cdot N \cdot \sum_{i=1}^{q} \frac{s_i \cdot N}{S_{tot} \cdot N} \cdot log_2\left(\frac{s_i \cdot N}{S_{tot} \cdot N} \cdot S_{tot} \cdot N\right)$$

$$= S_{tot} \cdot N \cdot \sum_{i=1}^{q} \frac{s_i \cdot N}{S_{tot} \cdot N} \cdot log_2\left(\frac{s_i \cdot N}{S_{tot} \cdot N}\right) +$$

$$S_{tot} \cdot N \cdot \sum_{i=1}^{q} \frac{s_i \cdot N}{S_{tot} \cdot N} \cdot log_2(S_{tot} \cdot N)$$

$$= S_{tot} \cdot N \cdot \sum_{i=1}^{q} \frac{s_i}{S_{tot}} \cdot log_2\left(\frac{s_i}{S_{tot}}\right) + \sum_{i=1}^{q} s_i \cdot N \cdot log_2(S_{tot} \cdot N)$$

$$= S_{tot} \cdot N \cdot \sum_{i=1}^{q} \frac{s_i}{S_{tot}} \cdot log_2\left(\frac{s_i}{S_{tot}}\right) + S_{tot} \cdot N \cdot log_2(S_{tot} \cdot N)$$

$$= S_{tot} \cdot N \cdot \left(\sum_{i=1}^{q} \frac{s_i}{S_{tot}} \cdot log_2\left(\frac{s_i}{S_{tot}}\right) + log_2(S_{tot} \cdot N)\right) \quad (17)$$

For given S_{tot}, N, and q the quantity of Equation 17 may have different values for different distribution of the s_i values. We identify the term $\sum_{i=1}^{q} \frac{s_i}{S_{tot}} \cdot log_2\left(\frac{s_i}{S_{tot}}\right)$ as an entropy term, $E(s_1, s_2, ..., s_q)$, because it has the properties of an entropy function: (i) $\sum_{i=1}^{q} \frac{s_i}{S_{tot}} = 1$ and (ii) $\frac{s_i}{S_{tot}} \leq 1$, as a result its maximum value is 0 and its minimum value is when all $\frac{s_i}{S_{tot}}$ are equal. This happens when $s_i = \frac{S_{tot}}{q}, \forall i$, which in turn means that:

$$\sum_{i=1}^{q} \frac{S_{tot}/q}{S_{tot}} \cdot log_2\left(\frac{S_{tot}/q}{S_{tot}}\right) = \sum_{i=1}^{q} \frac{1}{q} \cdot log_2\left(\frac{1}{q}\right) = log_2\left(\frac{1}{q}\right) \quad (18)$$

Putting together Equation 17 and 18 we can accurately calculate the maximum and the minimum sorting cost. The entropy term has maximum value 0 when all but one s_i are equal to 0 and only one of them is equal to S_{tot}, and it has minimum value $log_2\left(\frac{1}{q}\right)$ when all s_i are equal.

Finally, we have:

$$MinSC = S_{tot} \cdot N \cdot \left(log_2\left(\frac{1}{q}\right) + log_2(S_{tot} \cdot N)\right) \quad (19)$$

$$MaxSC = S_{tot} \cdot N \cdot log_2(S_{tot} \cdot N) \quad (20)$$

Note, that the calculation of the minimum and maximum value of this quantity can be equivalently calculated using the log sum inequality and the monotonicity of logarithm. The above, however, also explains under which conditions the bounds are sharp.

B. REWRITING THE APS RATIO

Equation 15 can be rewritten based on the initial parameters presented in Table 1.

$$APS = \frac{q \cdot TT + S_{tot}(TL + TD_I + TD_R)}{TD_S + S_{tot} \cdot TD_R} +$$
$$\frac{S_{tot} \cdot log_2(S_{tot} \cdot N) \cdot TD_R}{TD_S + S_{tot} \cdot TD_R} \Rightarrow$$

$$APS = \frac{q \cdot \frac{1+\lceil log_b(N) \rceil}{N} \cdot \left(BW_S \cdot C_M + \frac{b \cdot BW_S \cdot C_A}{2} + \frac{b \cdot BW_S \cdot f_p \cdot p}{2}\right)}{max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}} +$$
$$\frac{S_{tot}\left(\frac{BW_S \cdot C_M}{b} + (aw + ow) \cdot \frac{BW_S}{BW_I} + rw \cdot \frac{BW_S}{BW_R}\right)}{max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}} +$$
$$\frac{S_{tot} \cdot log_2(S_{tot} \cdot N) \cdot BW_S \cdot CA}{max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right) + S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}} \quad (21)$$

Equation 21 is used in the implementation of the model for producing the model-based analysis.

B.1 Hardware & Dataset Factors

Factor $BW_S \cdot C_M$. The product between scanning bandwidth BW_S and cache miss latency C_M gives the *number of bytes* scanned at memory bandwidth speed in the time of one *cache miss*. This factor is crucial when comparing the index vs. the scan performance.

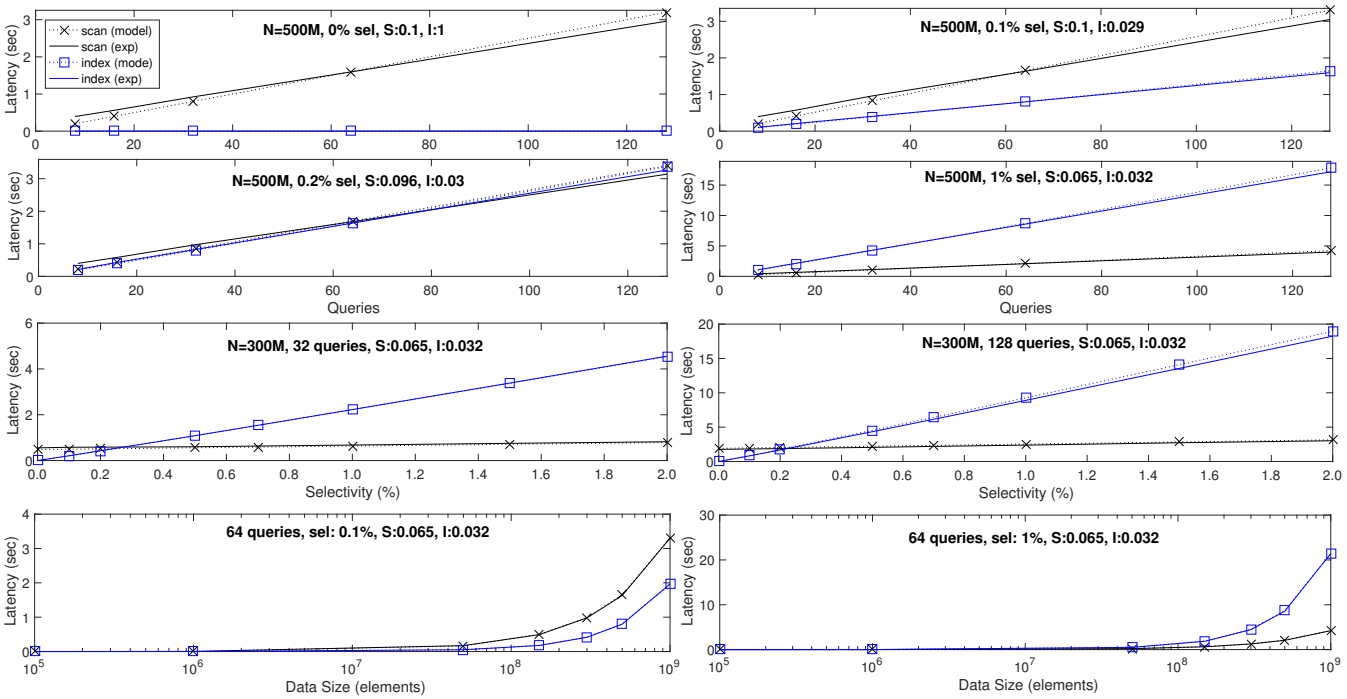

Figure 20: Through fitting, the model can closely match the performance of both shared sequential scans and index scans.

As the value of this product increases the scan becomes more beneficial because for each cache miss (i.e., random access of the tree nodes) corresponds to scanning more data. For lower values of this product, the index becomes more beneficial.

Factor $BW_S \cdot C_A$. The product between scanning bandwidth BW_S and cache miss latency C_M gives the *number of bytes* scanned at memory bandwidth speed in the time of one *cache access*. This factor has smaller impact than $BW_S \cdot C_M$ because it is typically one or two orders of magnitude smaller. The impact, however, albeit smaller follows the same trends.

Factor $max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right)$. The comparison between the tuple size ts and the product $2 \cdot f_p \cdot p \cdot BW_S$ informs us as to whether a scan is memory-bound or CPU-bound. The product $2 \cdot f_p \cdot p \cdot q \cdot BW_S$ takes into account the effective processor speed (i.e., processor speed and the effect of pipelining) and the memory bandwidth, and gives the *number of bytes* that can be sequentially scanned at the time of *calculating the two instructions* needed to evaluate the predicate of all q queries (i.e., the two comparisons needed with the beginning and the end of the range of each query). If this value is less than the tuple size ts, then the scan is memory-bound, otherwise it is CPU-bound.

C. MODEL VERIFICATION

We verified the accuracy of the model using experimental data from four machines. For each set of experiments, we used a multi-dimensional unconstrained nonlinear minimization technique (Nelder-Mead) to fit the model. We used the advertised hardware characteristics augmented by constant factors which we found to be the same across different experiments thereby verifying the model.

We augmented Equations 5 and 13. When modeling the scans, the fitting revealed that a constant factor is needed to adjust the cost of result writing. The equation is augmented with the new parameter α (Equation 22).

$$SharedScan = max(TD_S, q \cdot PE) + \alpha \cdot S_{tot} \cdot TD_R \quad (22)$$

The fitting process indicated that $\alpha = 8$, which explains the overlap when writing the result, due to using SIMD registers. Next, we augmented the equation for the index cost with a constant factor to account for our pessimistic expectation of the worst-case sorting cost. The new equation augmented with the new parameter f_s is shown in Equation 23.

$$\begin{aligned} ConcIndex =& q \cdot TT + S_{tot} \cdot (TL + TD_I) \\ &+ S_{tot} \cdot TD_R + f_c \cdot SF \cdot C_A \end{aligned} \quad (23)$$

The fitting process revealed that f_c is not a constant factor; rather, it is a function of N, fitted by the following expression (which is sublinear but more expensive than logarithmic with respect to N):

$$f_c = f_s \cdot \frac{N^{\beta} - 1}{\beta} \quad (24)$$

The values that describe accurately the index behavior are $\beta = 0.38$ and $f_c = 6 \cdot 10^{-6}$, we find them to be stable throughout different experiments with the same system.

C.1 Model Fitting

Here we describe the verification process for our primary experimental server, the full description of which can be found in Section 4. Figure 20 shows the fit of the model with respect to our primary experimental server when we vary a number of parameters including relation size (N), average query selectivity (s_i), and number of concurrent queries (q). The first four graphs correspond to a relation with size $N = 500$M tuples. The y-axis of each figure shows workload latency, and the x- axis shows the number of concurrent queries q. Each of the four figures has four lines. Two lines for the scan latency (model prediction, and experimental measurement), and two lines for index scan latency. Each figure corresponds to a different selectivity for each of the q queries in the batch; the first figure shows performance for queries with 0% selectivity, and the remaining figures 0.1%, 0.2%, and 1%. Next to the selectivity on top of every figure, we show the sum of the normalized least-square error for each access path.

The two graphs on the third row show the model accuracy as we vary the selectivity of each of the q queries (x-axis). The y-axis

still shows the workload latency. The leftmost graph corresponds to $q = 32$ and the rightmost to $q = 128$. Finally, the two graphs on the fourth row of Figure 20 show the accuracy of the model as we vary the data size in number of elements (x-axis). Similarly to the first four graphs, on the top of every figure we indicate the exact setup as well as the sum of the normalized least-square error for each access path. Overall, we observe that the fitted model provides a close match to the actual query latency.

We focus on the above numbers of selectivity, concurrency, and data size because these are the "interesting" ranges in which the APS switches from smaller than one to larger than one. Further, we experiment with concurrency of more than 8 queries to match our experimental setup: we always fully utilize an 8-thread socket of our experimental machine. Hence, for workloads of less than 8 queries the performance depends on the efficiency of the intra-operator parallelism, which is an open research problem. Finally, we fitted the model with different experimental setups leading to slightly different values for the fitting parameters (α, β, and f_c) which indicates that a training process of the model is generally required for every new setup. The training process of each system, though, requires only a small number of experiments before the model can accurately capture machine performance.

C.2 APS Equation With Fitting Parameters

The *APS* Equation 21 is now changed as follows:

$$APS = \frac{q \cdot \frac{1+\lceil log_b(N) \rceil}{N} \cdot \left(BW_S \cdot C_M + \frac{b \cdot BW_S \cdot C_A}{2} + \frac{b \cdot BW_S \cdot f_p \cdot p}{2} \right)}{max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right) + \alpha \cdot S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}} + \frac{S_{tot} \left(\frac{BW_S \cdot C_M}{b} + (aw + ow) \cdot \frac{BW_S}{BW_I} + rw \cdot \frac{BW_S}{BW_R} \right)}{max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right) + \alpha \cdot S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}} + \frac{f_s \cdot \frac{N^{\beta}-1}{\beta} \cdot S_{tot} \cdot log_2\left(S_{tot} \cdot N\right) \cdot BW_S \cdot CA}{max\left(ts, 2 \cdot f_p \cdot p \cdot q \cdot BW_S\right) + \alpha \cdot S_{tot} \cdot rw \cdot \frac{BW_S}{BW_R}} \quad (25)$$

For every new system we now need a fitting process to make sure the model accurately captures the execution environment. Once this process is completed, Equation 25 can be used to perform access path selection.

D. REDUCING SORTING COST

Careful use of SIMD instructions during the sorting step [61] can be used to further reduce the sorting cost, and hence the overall cost, of a secondary index scan.

The model can be adjusted to capture such optimizations by using the new cost of the new sorting algorithm to update Equation 14 that considers the sorting cost to be:

$$SF = \frac{S_{tot} \cdot N}{W} \cdot log\left(\frac{S_{tot} \cdot N}{W}\right) + S_{tot} \cdot N \cdot log\left(W\right), \quad (26)$$

where W is the SIMD-registers width

This change helps the case of the secondary index scan, effectively moving the crossover point to higher selectivity. Figure 21 is the equivalent of Figure 4 for $W = 4$. The observations from the analysis remain the same, and in fact the need for access path selection is further strengthened, because this optimization increases index performance, hence making the need for access path selection more pronounced.

E. ALTERNATIVE ACCESS PATHS

Lightweight Data Skipping. Data skipping is a form of scan enhancement. Zonemaps [26, 64, 78, 82] and Column Imprints [76] are prime examples. By keeping a small amount of metadata, like min/max information, on each of a number of zones in a column, a scan can quickly decide whether it needs to examine each zone based only on the metadata. Such approaches work quite well when data is clustered or fully sorted.

Our work is for full secondary indexing and can be extended to model data skipping as an enhancement of scans (e.g., by decreasing data input size in the model if statistics imply that several zones can be skipped). However, data skipping has its greatest impact when the column has a natural order (such as order dates for example). This means that in a table like TPC-H lineitem, only three or four of the sixteen attributes could stand to benefit from data skipping, selecting on any of the others would require either a sequential scan or secondary index scan.

Alternative Index Structures. In this paper, we chose to use a main-memory optimized B$^+$-Tree to model the access path of a secondary index. This decision is based on the fact that B$^+$-Trees have been used for indexing for several decades in database systems and there exist numerous optimizations [30]. In this way, we can contrast our findings in access path selection with those of traditional systems in as similar a comparison as possible.

In addition to main-memory optimized B$^+$-Trees, several index structures are specialized for modern hardware and can be examined for access path selection [10, 40, 45, 46, 52, 53, 57, 74, 83]. Such an analysis is orthogonal to the main point in this paper: access path selection is necessary and there is a new way to dynamically define the break-even point between a secondary index scan and a sequential scan. Some of these approaches [40, 46, 83] also consider the dimension of memory footprint in the trade-off space between read performance, update performance, and memory utilization [9]. The effect of alternative index techniques would be a shift in the balance, favoring indexing in more cases but there will still be a need for access path selection. This is a question that should be revisited every few years as systems change to include fundamentally new designs.

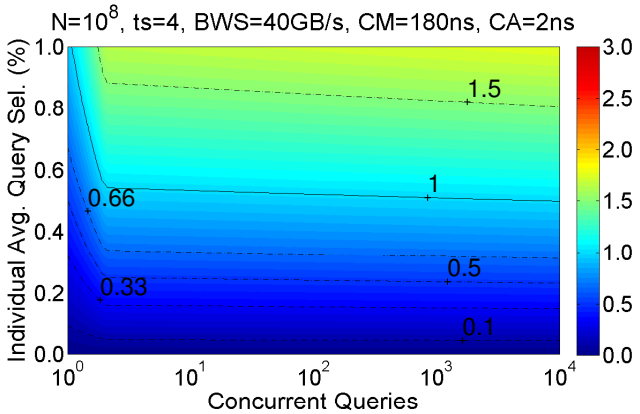

Figure 21: SIMD-aware sorting favors indexing.