Smooth Scan: Robust Query Execution with a Statistics-oblivious Access Operator

Technical Report

Ref: EPFL-REPORT-200188

Renata Borovica-Gajic

Advisor: Anastasia Ailamaki

Data-Intensive Applications and Systems laboratory (DIAS)
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne

renata.borovica@epfl.ch

ABSTRACT

Query optimizers depend heavily on statistics representing column distributions to create efficient query plans. In many cases, though, statistics are outdated or non-existent, and the process of refreshing statistics is very expensive, especially for ad-hoc workloads on ever bigger data. This results in suboptimal plans that severely hurt performance. The main problem is that any decision, once made by the optimizer, is *fixed* throughout the execution of a query. In particular, each logical operator translates into a fixed choice of a physical operator at run-time.

In this paper, we advocate for *continuous adaptation and morphing* of physical operators throughout their lifetime, by adjusting their behavior in accordance with the statistical properties of the data. We demonstrate the benefits of the new paradigm by designing and implementing an adaptive access path operator called *Smooth Scan*, which morphs continuously within the space of traditional index access and full table scan. Smooth Scan behaves similarly to an index scan for low selectivity; if selectivity increases, however, Smooth Scan progressively morphs its behavior toward a sequential scan. As a result, a system with Smooth Scan requires no access path decisions up front nor does it need accurate statistics to provide good performance. We implement Smooth Scan in PostgreSQL and, using both synthetic benchmarks as well as TPC-H, we show that it achieves robust performance while at the same time being statistics-oblivious.

1. INTRODUCTION

Perils of Query Optimization Complexity. Query execution performance of database systems depends heavily on query optimization decisions; deciding which (physical) operators to use and in which order to place them in a plan is of critical importance and can affect response times by several orders of magnitude [28]. To find the best possible plan, query optimizers typically employ a cost model to estimate performance of viable alternatives. In turn, cost models rely on statistics about the data. With the growth in complexity of decision support systems (e.g. templatized queries, UDFs) and the advent of dynamic web applications, however, the optimizer's grasp of reality becomes increasingly loose and it becomes more difficult to produce an optimal plan [19]. For instance, to defy complexity and make up for lack of statistics, commercial database management systems often assume uniform data distributions and attribute value independence, which is in reality hardly the case [10]. As a result, database systems are increasingly confronted with suboptimal plans and subpar performance [5, 13, 15, 32, 34, 37].

Motivating Example. To illustrate the severe impact of incomplete statistics and consequent suboptimal access path choices, we use a state-of-the-art commercial system, referred to as DBMS-X, and run the TPC-H benchmark [39] (the exact set-up is discussed in Section 6.2). When considering access paths, the optimizer needs accurate statistics to estimate the tipping point between a full scan and an index scan to make the proper choice. Figure 1 demonstrates the impact of suboptimal index choices after tuning DBMS-X for TPC-H; the graph shows normalized execution times over non-tuned performance. Despite using the official tuning tool of DBMS-X in the experiment, for several queries performance degrades significantly after tuning (e.g., up to a factor of 400 for Q12).[1] The only change compared to the original plan of Q12 is the type of access path operator. This decision however prolonged the execution time from a minute to 11 hours.

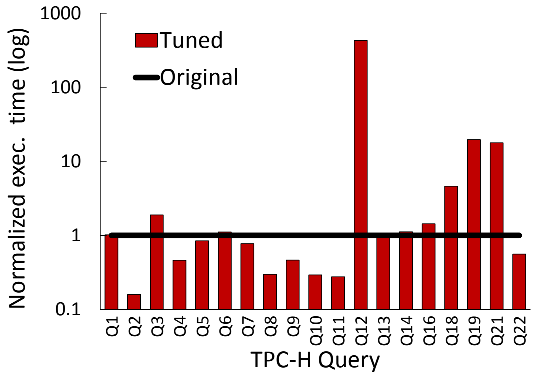


Figure 1: Non-robust Performance due to Optimization Errors in a State-of-the-art Commercial DBMS when running TPC-H.

Robust Execution. The core of the problem of suboptimal plans lies in the fact that even a small estimation error may lead to a drastically different result in terms of performance. For instance, one tuple difference in cardinality estimation can *swing the decision* between an index scan and a full scan, possibly causing a significant performance drop. Overall, this results in *unpredictable performance* thereby affecting the *robustness* of the system. In addition, the overall behavior is driven by the accuracy of statistics present in the current server, which aggravates the testing *repeatability* across different servers or even different invocations (since statistics might change in between). Stability and predictability, that imply that similar query inputs should have similar execution performance, are major goals for industrial vendors towards respecting service level agreements (SLA) [33]. This is exemplified, nowadays, in cloud environments, offering paid-as-a-service functionality governed by SLAs in environments which are much more ad-hoc than traditional closed systems. In these cases, a system's ability to efficiently operate in the face of unexpected and especially adverse run-time conditions (e.g., receiving more tuples from an operator than estimated) becomes more important than yielding great performance for one query input while suffering from severe degradation for another [21]. We define *robustness in the context of query processing as the ability of a system to efficiently cope with unexpected and adverse conditions, and deliver near-optimal performance for all query inputs.*

Past efforts on robustness focus primarily on dealing with the problem at the optimizer level [5, 11, 12]. Nonetheless, in dynamic environments with constantly changing workloads and data characteristics, judicious query optimization performed up front could bring only partial benefits as the environment keeps changing even after optimization. Orthogonal approaches on run-time adaptivity [3, 30, 32, 34], although promising, are lacking the flexibility at the level of access paths.[2] Furthermore, since the violation of the optimizer's estimates usually triggers reoptimization, these ap-

[1]Similar results have been presented in a related study [7].

[2]They are limited either in their scope (by ignoring intra-operator adaptivity, or by performing binary switching decisions that introduce risks and could lead to thrashing) or with respect to performance (by duplicating work and/or transforming operators into blocking ones).

proaches remain sensitive to the accuracy of statistics, which complicates testing across different environments.

Smooth Scan. We respond to the need for robust execution by introducing a novel class of access path operators designed with the goal of providing robust performance for every input regardless of the severity of cardinality estimation errors. Since the understanding of the data distributions is a continuous process that develops throughout the execution of a query plan and moreover since one execution strategy might not be optimal over the entire data set (i.e., we can have sparse and dense regions with respect to the tuple placement on disk), we need a new class of *morphable operators* that *continuously and seamlessly adjust* their execution strategy as the understanding of the data evolves. We introduce Smooth Scan, an operator that morphs between an index look-up and a full table scan, achieving near-optimal performance regardless of the operator's selectivity *and obliviously to the existing data statistics.* Our aim is to provide graceful degradation with respect to the selectivity increase and be as close as possible to the performance that could have been achieved if all necessary statistics were available. In addition, morphing relieves the optimizer from choosing an optimal access path a priori, since the execution engine has the ability to adjust its behavior at run-time as a response to the observed operator selectivity.

Contributions. Our contributions are as follows:

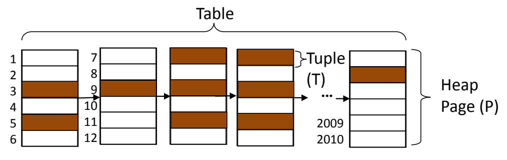
- We propose a new paradigm of smooth and morphable physical operators that adjust their behavior and transform from one operator implementation to another according to the statistical properties of the data observed at run-time.
- We design and implement a statistics-oblivious Smooth Scan operator that morphs between an index access and a full scan as selectivity knowledge evolves at run-time.
- Using both synthetic benchmarks and TPC-H, we show that Smooth Scan, implemented fully in PostgreSQL, is a viable option for achieving near-optimal performance throughout the entire selectivity interval, by being either competitive with or outperforming existing access path alternatives.

2. BACKGROUND

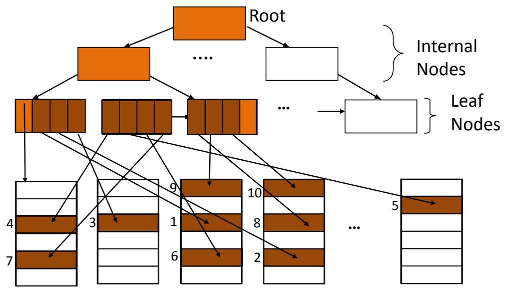
In order to fully understand the advantages and the mechanisms of the Smooth Scan operator, this section provides a brief background on traditional access path operators.

Full Table Scan is employed when there are no alternative access paths, or when the selectivity of the access operator is estimated to be high (above 1-10% depending on the system parameters). The execution engine starts by fetching the first tuple from the first page of a table stored in a heap, and continues accessing tuples sequentially inside the page. It then accesses the adjacent pages until it reaches the last page. Figure 2a depicts an example of a full scan over a set of pages in the heap; the number placed on the left-hand side of each tuple indicates the order in which it is accessed. Even if the number of qualifying tuples is small, a full table scan is bound to fetch and scan all pages of a table, since there is no information on where tuples of interest might be. On the positive side, the sequential access pattern employed by the full table scan is one to two orders of magnitude faster than the random access pattern of an index scan.

Index Scan. Secondary (non-clustered) indices are built on top of data pages. They are usually B$^+$-trees containing pointers to tuples stored in the heap. Figure 2b depicts a B$^+$-tree built on top of the same table we used in Figure 2a. The leaves of the tree point to the heap data pages. A query with a range predicate needs to traverse the tree once in order to find the pointer to the first tuple that qualifies, and then it continues following adjacent leaf pointers until it finds the first tuple that does not qualify. The upside of this approach, compared to the full scan, is that only tuples that are needed are actually accessed. The downside is the random access pattern when following pointers from the leaf page(s) to the heap (shown as lines with arrows). Since the random access pattern is much slower than the sequential one, performance degrades quickly if many tuples need to be selected. Moreover, the more tuples qualify, the higher the chance that the index scan needs to visit the same page more than once.



(a) Full Table Scan



(b) Index Scan

Figure 2: Access Paths in a DBMS.

Sort Scan (Bitmap Scan) represents a middle ground between the previous two approaches. Sort Scan still exploits the secondary index to obtain the identifiers (TIDs) of all tuples that qualify, but prior to accessing the heap, the qualifying tuple IDs are sorted in an increasing heap page order. In this way, the poor performance of the random access pattern gets improved by transforming the access into a (nearly) sequential pattern, easily detected by disk prefetchers. This strategy, however, has dramatic influence on the execution model. The index access that traditionally followed the pipeline execution model, now gets transformed into a blocking operator which can be harmful, especially when the index is used to provide an *interesting ordering* [35]. One advantage of B-tree indices comes from the fact that tuples are accessed in the sorted order of attributes on which the index is built. Sorting of tuple IDs based on their page placement breaks the natural index ordering that needs to be restored by introducing a sorting operator above the index access (or up in the tree). In addition, the blocking operator so early in the execution plan could stall the rest of the operators; if they require a sorted input, their execution can start only after the second sort finishes.

3. SMOOTH ACCESS PATHS

We now present the concept of smooth access path operators that adjust their execution strategy at run-time to fit the data distributions. We first discuss Switch Scan that switches access path strategy with a binary decision at run-time. We then introduce Smooth Scan that instead of making a single on/off decision, gradually and adaptively shifts its behavior between access path patterns, avoiding performance drops.

Switch Scan. The main cause for suboptimal access paths comes from a wrong selectivity estimation. One approach to resolve the problem is to monitor result cardinality during query execution

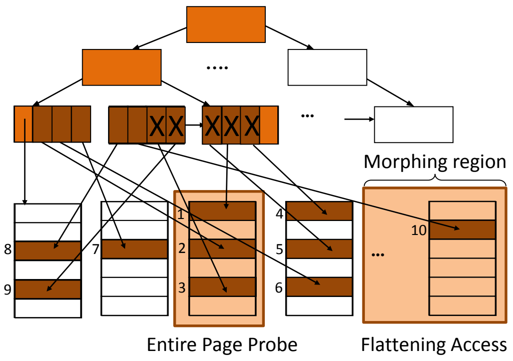


Figure 3: Smooth Scan Access Pattern.

and *switch* the access path strategy when we realize that the initial estimation was wrong. Once the actual cardinality exceeds the expected result cardinality, we can throw away all the work performed until that point and restart the execution with a different access path. A more advanced approach reuses the existing intermediate results, e.g., by remembering which tuples and pages the previous access path already visited.

Switching Perils. Switch Scan bounds the worst case execution time as it will never degrade as much as an index scan only approach. However, it is still not a robust approach. The main problem with Switch Scan is that it is based on a *binary decision* and switches completely when a certain cardinality threshold is violated. This means that even a single extra result tuple can bring a *drastically different performance* result if we switch access paths. We refer to the effect of a sudden increase in execution time as a *performance cliff*. The performance hit together with the uncertainty whether the overhead incurred at the time of a change will be amortized over the remaining query time renders this approach volatile and non-robust.

Smooth Scan. The core idea behind Smooth Scan is thus to *gradually* transform between two strategies, i.e., a non-clustered index look-up and a full table scan, maintaining the advantages of both worlds. Our main objective is to provide *smooth behavior*, i.e., at no point should an extra tuple in the result cause a performance cliff. Smooth Scan instead *morphs* its behavior incrementally, and continuously, causing only gradual changes as it goes through the data and its estimation about result cardinality evolves.

3.1 Morphing Mechanism

During a single scan Smooth Scan can be in three modes, while morphing between an index and full scan. In each mode the operator performs a gradually increasing amount of work as a result of the selectivity increase.

Mode 1: Entire Page Probe. To avoid repeated page accesses from which the index scan suffers, in this mode Smooth Scan analyzes *all* records from each heap page it loads to find qualifying tuples, trading CPU cost for I/O cost reduction. Since the cost of an I/O operation translates to an order of million CPU instructions [17], Smooth Scan invests CPU cycles for reading additional tuples from each page with minimal CPU overhead. Figure 3 depicts the access pattern of a Smooth Scan in this mode. As in Figure 2, the number at the left-hand-side of each tuple indicates the order in which the access path touches this tuple. Within each page, Smooth Scan accesses tuples sequentially.

Mode 2: Flattening Access. When the result cardinality grows, Smooth Scan amortizes the random I/O cost by flattening the random pattern and replacing it with a sequential one. Flattening happens by reading additional adjacent pages from the heap, i.e., for each page it has to read, Smooth Scan prefetches a few more adjacent pages (read sequentially). An example of a morphing region is depicted in Figure 3.

Mode 2+: Flattening Expansion. Flattening Access Mode is in fact an ever expanding mode. When it first enters Flattening Access Mode, Smooth Scan starts by fetching one extra page for each page it needs to access. However, when it notices result selectivity increase, Smooth Scan progressively increases the number of pages it prefetches by multiplying it with a factor of 2. The reason is that, as selectivity increases, the I/O increase of fetching more potentially unnecessary pages could be masked by the CPU processing cost of the tuples that qualify. In this way, as the result cardinality increases more, Smooth Scan keeps expanding, and conceptually it morphs more aggressively into a full table scan.

3.2 Morphing Policies

We now describe how Smooth Scan changes modes.

Greedy Policy. Assuming a worst case scenario, Smooth Scan can perform morphing expansion after each index probe. In this way, the morphing expansion greedily follows the selectivity increase. The upside of this approach is that, due to its fast convergence, its worst case performance resembles the performance of a full scan. The downside is that in the case of low selectivity it introduces an overhead of reading unnecessary pages that could not be masked by useful work.

Selectivity Increase Driven Policy. Blindly morphing between the modes may introduce too much overhead if the I/O cost cannot be overlapped with useful work. With this policy, Smooth Scan continuously monitors selectivity at run-time, and it expands the morphing region when it notices a selectivity increase. In particular, Smooth Scan computes the result selectivity over the last morphing region and it increases the morphing region size each time the local selectivity over the last morphing region (Eq. (1))[3] is greater than the global selectivity over the so far seen pages (Eq. (2)). If selectivity does not increase, Smooth Scan keeps the previous morphing region size. This way, morphing is performed at a pace which is purely driven by the data and the query at hand.

$$sel_{local} = \frac{\#P_{res\_region}}{\#P_{seen\_region}} \tag{1}$$

$$sel_{global} = \frac{\#P_{res}}{\#P_{seen}} \tag{2}$$

Elastic Policy. When considering large data sets, it is unlikely that a single execution strategy will be optimal during the whole scan of a big table in a given query; dense and sparse regions with respect to the tuple placement on disk frequently appear in such a context due to skewed data distributions. To benefit from the density discrepancy and use skew as an opportunity, Smooth Scan uses the Elastic Policy to morph two-ways; it increases the morphing size over a dense region, while it decreases the morphing region size when it passes through a sparse region. More precisely, if the local selectivity over the last morphing region is higher than the global selectivity over all tuples seen so far, then this implies we are in a denser region, hence we double the morphing size. In the opposite case, we decrease the morphing size for the next morphing region.

3.3 Morphing Triggering Point

We now present Smooth Scan morphing triggers.

Optimizer Driven. Smooth Scan can be introduced to the existing query stack as a reaction to unfavorable conditions, i.e., as a robustness patch. With this strategy, we initiate morphing once the

[3]The meaning of the parameters can be found in Table 1.

result cardinality exceeds the optimizer's estimate. A cardinality violation is an indication that the optimizer's estimate is inaccurate and that the chosen access path might be suboptimal. After triggering, Smooth Scan can morph with either of the policies described in Section 3.2.

SLA Driven. Another option is to take action only when in danger of violating a performance threshold, i.e., a service level agreement (SLA). For example, let us assume a given time T as an upper bound (SLA) for the operator execution. In this case, Smooth Scan continuously monitors execution and has a running estimate of the expected total cost (based on the cost model discussed in Section 5). The moment we realize that unless we switch to more conservative behavior we will not be able to guarantee the SLA target performance, we trigger morphing with Smooth Scan.

Eager Approach. An alternative approach, which we favor, is to completely replace access paths with Smooth Scan. With this strategy, we eagerly start with Smooth Scan immediately as of the first tuple. In this way, we guarantee that the total number of page accesses will be equal to the total number of heap pages in the worst case. Moreover, with this strategy there is no need to record tuples produced before morphing has started (to prevent result duplication), which provides additional benefit and decreases bookkeeping information.

In our experiments Eager is the default strategy. We study other strategies in detail in the experimental section.

4. INTRODUCING SMOOTH PATHS INTO POSTGRESQL

In this section, we discuss the design details of smooth access operators, and their interaction with the remaining query processing stack. We implement our operators, both the Switch Scan and Smooth Scan families in PostgreSQL 9.2.1 DBMS as classical physical operators existing side by side with the traditional access path operators. During query execution, the access path choice is replaced by the choice of Smooth Scan, while the upper layers of query plans generated by the optimizer remain intact. Unlike the dynamic reoptimization approaches proposed in [2,3], our proposal requires minimal changes to the existing database architecture.

Switch Scan could conceptually be considered as an instance of Smooth Scan with a threshold driven policy (usually the optimizer's result cardinality estimate) that abruptly switches to Full Scan after reaching the threshold. Therefore, we do not discuss it separately; except when performing the experimental evaluation in Section 6.6.

4.1 Design Details

To make the Smooth Scan operator work efficiently, several critical issues need to be addressed.

Page ID Cache. To avoid processing the same heap page twice (since multiple out-of-order leaf pointers of the index can point to the same page), Smooth Scan keeps track of the pages it has read and records them in a Page ID Cache. The Page ID Cache is a bitmap structure with one bit per page. Once a page is processed its bit is set to 1. When traversing the leaf pointers from the index, a bit check precedes a heap page access. Smooth Scan will access the heap page only if that page has not been accessed before. Otherwise, we skip the leaf pointer (X in Figure 3) and continue the leaf traversal.

Result Cache. If an index is chosen to support an interesting order (e.g., in a query with an ORDER BY clause), then the tuple order has to be respected. This means that a query plan with Smooth Scan cannot consume all tuples the moment it produces them. To address this, the additional qualifying tuples found (i.e., all but the one specifically pointed to by the given index look-up) are kept in the Result Cache. The Result Cache is a hash-based data structure that stores qualifying tuples. In this setting, an index probe is preceded by a hash probe of the Result Cache for each tuple identifier obtained from the leaf pages of the index. If the tuple is found in the Result Cache it is immediately returned (and could be deleted), otherwise Smooth Scan fetches it from the disk following the current execution mode. The cache deletion is done in a bulk fashion. We partition the Result cache into a number of smaller caches that can be deleted once all tuples from an instance are produced. By grouping the caches per key ranges, we can remove all items from one cache as soon as the key range of the cache is traversed. The key range intervals are decided by looking at the root page of the index, since the root page is a good indicator of the key value distributions. Even more precise information on key range distributions could be obtained by looking at the internal node pages if they are cached in memory (which is to be expected since these pages are usually 1‰ to 1% of data pages).

Tuple ID Cache. If we switch from the traditional index scan following the Optimizer or SLA Driven strategy, we have to ensure that the result tuples will not be duplicated. This could happen if a result tuple is produced by following the traditional index, and later on we fetch the same page with Smooth Scan. To address this issue, we keep a cache of tuple IDs produced with the traditional access in a bitmap-like structure. Later, while producing tuples with Smooth Scan we perform a bit check if the tuple has already been produced. The overhead of the Tuple ID Cache, while relatively low, could be avoided if a DBMS maintains a strict $(index_{key}, TID)$ ordering in the secondary index. Then it is sufficient to remember the last tuple we reached with the traditional index, and ignore tuples with $(index_{key}, TID)$ lower than that last tuple.

Discussion. Both the Page ID and Tuple ID Cache are bitmap structures, meaning that their size is significantly smaller than the data set size (they easily fit in memory). To illustrate, their size is usually a couple of MB for hundreds of GB of data. In the Tuple ID cache, we keep IDs of the tuples produced with the traditional index, which is in practice significantly lower than the overall number of tuples. The Result Cache is an auxiliary structure whose size depends on the access order of tuples, the number of attributes in the payload, and the overall operator selectivity. By grouping caches per key value, we are able to delete them as soon as they are not needed. If memory becomes scarce, cache spilling could be employed by using overflow files. Caches containing the ranges the furthest from the current key range are spilled into the overflow files that are read upon reaching the range keys belong to.

4.2 Interaction with Query Processing Stack

Smooth Scan is an access path targeted primarily at preventing severe degradation due to unexpected selectivity increase. Nonetheless, its impact goes much beyond.

Simplified Query Optimization. Smooth Scan simplifies the query optimization process. Effectively, when choosing the access path for a select operator the optimizer can always choose a Smooth Scan. The Smooth Scan will then make all decisions on-the-fly during query execution.

Interaction with Other Operators. The output of Smooth Scan is input for other operators in a query plan. Depending on the next operator a different variation of Smooth Scan may be used. For example, if a Merge Join follows Smooth Scan, then the variant of Smooth Scan with the result caching will be used. If instead Index Nested Loops Join (INLJ) is performed, Smooth Scan does not have to respect the order of tuples coming from the outer input, hence it can produce tuples the moment it finds them. If the

Table 1: Cost model parameters

Parameter	Description
T_S	Tuple size (bytes).
$\#T$	Number of tuples in the relation.
P_S	Page size (bytes).
$\#T_P$	Number of tuples per page.
$\#P$	Number of pages the relation occupies.
K_S	Size of the indexing key (bytes).
sel	Selectivity of the query predicate(s) (%).
$card$	Number of result tuples.
$card_{mX}$	Number of tuples obtained with Mode X.
$m0_{check}$	0 or 1. Was a traditional index employed first?
$rand_{cost}$	Cost of a random I/O access (per page).
seq_{cost}	Cost of a sequential I/O access (per page).
cpu_{cost}	Cost of a CPU operation (per tuple).
$\#P_{res}$	Number of pages containing result tuples.
$\#P_{res\_region}$	Number of pages with result in current region.
$\#P_{seen}$	Number of pages seen so far.
$\#P_{seen\_region}$	Number of pages in the current region.
$\#rand_{io}$	Number of random accesses.
$\#seq_{io}$	Number of sequential accesses.
Derived values	
$fanout$	B$^+$-tree fanout.
$\#leaves$	Number of leaf pages in B$^+$-tree.
$\#leaves_{res}$	Number of leaf pages with pointers to results.
$height$	Height of B$^+$-tree.
OP_{io_cost}	Cost of an operator in terms of I/O.
OP_{cpu_cost}	Cost of an operator in terms of CPU.
CR	Competitive ratio.

ordering requirement is placed by some of the operators up in the tree, we still employ the first option. If Smooth Scan serves as an inner input (i.e., a parameterized path) to an INLJ join, the results per join key could be produced in an arbitrary order. Smooth Scan thus performs morphing per key value which reduces the number of repeated and random access for that particular key. The latter helps in the case of multiple matches per key (e.g., in a PK-FK relationship).

Beyond Traditional Join Operators. We have seen how Smooth Scan enables graceful degradation of joins, by reducing random and repeated I/O accesses either at the table level (when served as an outer input) or per join key value (e.g. when served as an inner input to INLJ). By employing the same concept of smooth morphing and transformation, we could benefit even more at the level of join operators. For instance, by performing caching of additional (qualifying) tuples from the inner input found along the way (i.e., for each page we fetch, we put the remaining tuples in the cache), INLJ morphs into a variant of Hash Join (HJ) overtime, with the index used only when a tuple is not found in the cache. Similarly, MJ morphs into a symmetric Hash Join [41], frequently used in data streaming environments due to its pipelining nature and amenability for operator reordering at run-time.

Ultimately, a morphable join and a morphable access path operator can significantly reduce the complexity and fragility of existing query optimizers. We, however, leave a discussion on the join operators as an avenue of future work, and do not use the proposed join optimization here.

5. MODELING SMOOTH ACCESS PATHS

To better grasp the behavior of different access path alternatives, and to answer the critical questions of which policy and mode we should use and when, in this section we model the operators analytically. Since Smooth Scan trades CPU for I/O cost reduction, we model the cost of the operators both in terms of the number of disk I/O accesses, and their CPU cost. Since one I/O operation maps to an order of million CPU cycles [17], it is expected the overall cost to be dominated by the I/O component; nonetheless, we disclose CPU costs for completeness. Moreover, we make a distinction between the cost of a sequential and random access, since the nature of the accesses drives the overall query performance.

$$\#T_P = \lfloor \frac{P_S}{T_S} \rfloor \quad (3)$$

$$\#P = \lceil \frac{\#T}{\#T_P} \rceil \quad (4)$$

$$fanout = \lfloor \frac{P_S}{1.2 \times K_S} \rfloor \quad (5)$$

$$\#leaves = \lceil \frac{\#T}{fanout} \rceil \quad (6)$$

$$height = \lceil \log_{fanout}(\#leaves) \rceil + 1 \quad (7)$$

$$card = sel \times \#T \quad (8)$$

$$\#leaves_{res} = \lceil \frac{card}{fanout} \rceil \quad (9)$$

$$OP_{cost} = OP_{io\_cost} + OP_{cpu\_cost} \quad (10)$$

Table 1 contains the parameters of the cost model. Formulas calculating the cost of the non-clustered index scan and the full scan are presented for comparison purposes (similar cost model formulas are found in database text books). We assume indices are implemented as B$^+$-trees, with k as the tree fanout. Equations 1-7 are base formulas used for all access path operators. The final cost of every operator is a sum of its I/O and CPU costs. We simplify the calculations by assuming every page is filled completely(100%) and that heap pages and index pages are of the same size (P_S). Lastly, we assume that T_S already includes a tuple overhead (usually padding and a tuple header). In Eq. (5), we calculate the fanout of the B$^+$-tree by adding 20% of space per key for a pointer to a lower level. For Eq. (6) and (9), we assume that every tuple stored in a heap page has a pointer to it in a leaf page of the index.

Full Table Scan. The cost of full scan does not depend on the number of tuples that qualify for the given predicate(s). Thus, regardless of the selectivity of the operator its cost remains constant. As shown in Eq. (11), the I/O cost is equal to the cost to fetch all pages of the relation sequentially. Once we fetch a page, we perform a tuple comparison for all tuples from the page to find the ones that qualify. In Eq. (12) (and further on) we assume that each comparison invokes one CPU operation.

$$FS_{io\_cost} = \#P \times seq_{cost} \quad (11)$$

$$FS_{cpu\_cost} = \#T \times cpu_{cost} \quad (12)$$

Index Scan. To fetch the tuples with the (non-clustered) index scan, we traverse the tree once to find the first tuple that qualifies (*height* in Eq. (13)). For the remaining tuples, we continue traversing the leaf pages from the index ($\#leaves_{res} \times seq_{cost}$) and use tuple IDs we found to access the heap pages, potentially triggering a random I/O operation per look-up. While traversing the tree for every index page we perform a binary search in order to find a pointer of interest to the next level (in Eq. (14)). For each tuple obtained by following the pointers from the leaf we perform a tuple comparison to see whether it qualifies.

$$
\begin{aligned}
IS_{io\_cost} &= (height + card) \times rand_{cost} \\
&+ \#leaves_{res} \times seq_{cost} && (13) \\
IS_{cpu\_cost} &= (height \times log_2(fanout) + card) \times cpu_{cost} && (14)
\end{aligned}
$$

Smooth Scan. We calculate the cost of Smooth Scan for each mode separately. Overall result cardinality is split between the modes (Eq. (15)). Like the index scan, the cost of the smooth scan access is driven by selectivity. Assuming uniform distribution of the result tuples (worst case scenario), the number of pages containing the result is calculated in Eq. (16).

$$
\begin{aligned}
card &= card_{m0} + card_{m1} + card_{m2} && (15) \\
\#P_{res} &= min(card, \#P) && (16)
\end{aligned}
$$

Mode 0: Index Scan. If the traditional index is employed prior to morphing, the I/O cost to obtain first $card_{m0}$ tuples is identical to the cost of the index scan for the same number of tuples, hence we omit the formula. A slight difference is in calculating the CPU cost of the operator in Mode 0 (the multiplier 2 in Eq. (17)), to add tuple IDs to the Tuple ID cache.

$$
\begin{aligned}
SS_{cpu\_cost\_m0} &= (height \times log_2(fanout) \\
&+ card_{m0} \times 2) \times cpu_{cost} && (17)
\end{aligned}
$$

Mode 1: Entire Page Probe. We calculate the number of tuples for which Mode 1 is going to be employed in Eq. (18) (again worst case). Every page is assumed to be fetched with a random access (Eq. (19)). Once we obtain a page, we perform a tuple comparison for all tuples from the page (the first part of Eq. (20)). Before fetching the page we check whether the page is already processed, and upon its processing we add it to the Page Cache (the second part of Eq. (20)). Finally, if we started with the traditional index for each tuple we have to perform a check whether the tuple has already been produced in Mode 0 (the third part Eq. (20)). In the case we need to support an interesting order, the Result Cache will be used as a replacement for the Tuple ID cache functionality. In that case we only mark the tuple ID as a key in the cache, without copying the actual tuple as a hash value; the probe match without the actual result thus signifies that the tuple has already been produced. Thus, the CPU cost remains (roughly) the same in both cases.

$$
\begin{aligned}
\#P_{m1} &= \min(card_{m1}, \#P) && (18) \\
SS_{io\_cost\_m1} &= \#P_{m1} \times rand_{cost} && (19) \\
SS_{cpu\_cost\_m1} &= (\#P_{m1} \times \#T_P + \#P_{m1} \times 2 \\
&+ \#P_{m1} \times \#T_P \times m0_{check}) \times cpu_{cost} && (20)
\end{aligned}
$$

Mode 2: Flattening Access. We calculate the maximum number of pages to fetch with Mode 2 in Eq. (21). Notice that pages processed in Mode 1 are skipped in Mode 2. The nature of the morphing expansion in Mode 2 of Smooth Scan is described with Eq. (22). The solution of the recurrence equation is shown in Eq. (23). In our case, n is the number of times we expand the morphing size (i.e., the number of times we perform a random I/O access) and $f(n)$ translates to the number of pages to fetch with Mode 2 ($\#P_{m2}$). We obtain the minimum number of random accesses (jumps) to fetch all pages containing the results from Eq. (25). This number is the best case scenario, when the access pattern is such that all pages are fetched with the flattening pattern without repeated accesses. The worst case scenario number of random accesses is shown in Eq. (26). When selectivity is low, the number of random I/O accesses could at worst be equal to the number of pages that contain the results. Nonetheless, there is an upper bound to it, equal to the logarithm of the number of pages in total.

Since both formulas (25) and (26) converge to the same value equal to $log_2(\#P + 1)$, we use this value in the remainder of the section. The I/O cost of Mode 2 of Smooth Scan is shown in Eq. (27), and is equal to the cost of the number of jumps with a random access pattern, plus the cost to fetch the remaining number of pages with a sequential pattern. The CPU cost per page in Mode 2 is identical to the cost per page in Mode 1 (Eq. (28)).

$$
\begin{aligned}
\#P_{m2} &= \min(card_{m2}, \#P - \#P_{m1}) && (21) \\
f(i+1) &= 2 \times f(i), i = 0..n && (22) \\
f(0) &= 0, f(n) = 2^n, n >= 0 && (23) \\
\#P_{m2} &= \sum_{i=0}^{\#rand_{io}(m2\_min)} 2^i && (24) \\
\#rand_{io}(m2\_min) &= \log_2(\#P_{m2} + 1) && (25) \\
\#rand_{io}(m2\_max) &= \min(\#P_{m2}, \log_2(\#P + 1)) && (26) \\
SS_{io\_cost\_m2} &= \#rand_{io}(m2) \times rand_{cost} \\
&+ (\#P_{m2} - \#rand_{io}(m2)) \times seq_{cost} && (27) \\
SS_{cpu\_cost\_m2} &= (\#P_{m2} \times \#T_P + \#P_{m2} \times 2 \\
&+ \#P_{m2} \times \#T_P * m0_{check}) \times cpu_{cost} && (28)
\end{aligned}
$$

Finally, the overall cost is the sum of the operator CPU and I/O costs for all employed modes(Eg. (29)).

$$
\begin{aligned}
SS_{io\_cost} &= SS_{io\_cost\_m0} + SS_{io\_cost\_m1} + SS_{io\_cost\_m2} \\
SS_{cpu\_cost} &= SS_{cpu\_cost\_m0} + SS_{cpu\_cost\_m1} + SS_{cpu\_cost\_m2} \\
SS_{cost} &= SS_{io\_cost} + SS_{cpu\_cost} && (29)
\end{aligned}
$$

The cost model formulas allow us to predict the cost of Smooth Scan policies, or to decide when is time to trigger a mode change. For instance, for the SLA driven strategy we know the overall operator cost defined by an SLA. Based on that cost and Eq. (29), we could calculate the cardinality, i.e., the triggering point for Smooth Scan calculated for the worst case scenario (selectivity 100%).

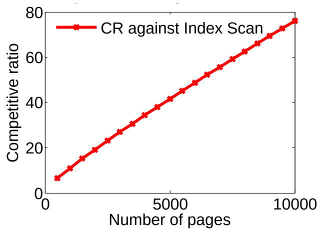
5.1 Competitive Analysis

In this section we perform a competitive analysis comparing the Smooth Scan operator against optimal decisions throughout the entire selectivity interval. We calculate a *competitive ratio* (CR) as the maximum ratio between the cost of Smooth Scan and the optimal solution throughout the entire selectivity interval (Eq. (30)). This number shows the maximum discrepancy from the optimal solution. The competitive ratio is a viable metric when considering robustness, since it provides bounds on the worst case suboptimality. We first consider Smooth Scan with the Greedy Policy according to which we increase the morphing size after each index access. Then, we consider the Selectivity Increase (SI) Driven policy that increases the morphing region size as a response to the observed local selectivity increase. Lastly, we consider the refinement introduced with the Elastic Policy according to which the morphing expansion is performed only in the dense regions of the data set, while we decrease the morphing size in sparse regions.

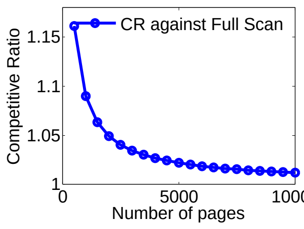
$$
\begin{aligned}
CR &= \max\left(\frac{SS_{cost}}{OP_{optimal}}\right), sel \in [0, 100\%] \\
&= \max\left(\frac{SS_{cost}}{\min(IS_{cost}, FS_{cost}, Oracle)}\right) && (30)
\end{aligned}
$$

5.1.1 *Greedy Policy*

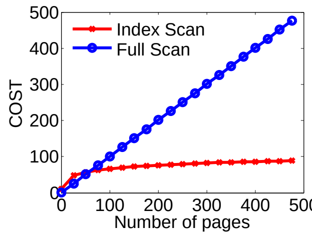
The worst case scenario for the Greedy policy is when everything we fetch with the flattening access pattern is useless, i.e., it does not



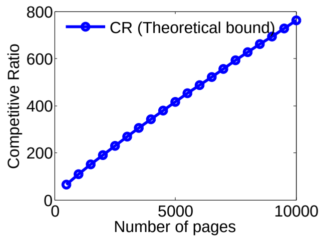
(a) CR against Index Scan



(b) CR against Full Scan



(c) Full vs. Index Scan



(d) CR Theoretical Bound

Figure 4: Competitive Analysis of Greedy Policy for Highest Page Miss (Worst Case Scenario)

contain any tuple of the result set. In this case, the number of fault pages (pages which do not contain the result tuples) is maximized. This can happen when the next tuple is always one page ahead of the current morphing region. Of course, the order does not have to be such that the page is strictly ahead, but without the loss of generality we assume this use case scenario, while in order to cover the most adversarial behavior we consider index accesses between morphing regions to be entirely random.

Figure 5a depicts this use case scenario. With striped lines we denote pages containing results, while empty squares denote fault pages (i.e., page misses). Below each figure describing the result distribution pattern, we show the number of page hits (dividend) per the morphing region size (divisor). The case when Greedy Smooth Scan is least effective is when the number of page hits is equal to the maximum number of (random) jumps distributed over the entire table (depicted in Figure 5a). With the selectivity increase above this number, Smooth Scan's number of I/O accesses remains constant since all pages of the table have been accessed, and thus Smooth Scan only benefits from further selectivity increase. Therefore, the worst case performance of Smooth Scan is when the cardinality is equal to the number of random jumps (Eq. (31)). Eq. (32) shows the cost of Smooth Scan for this use case scenario.

$$card = \#rand_{io} = \log_2(\#P+1) \tag{31}$$

$$\begin{aligned} SS_{cost} &= \#rand_{io} \times rand_{cost} \\ &+ (\#P - \#rand_{io}) \times seq_{cost} \end{aligned} \tag{32}$$

$$CR = \frac{SS_{cost}}{\min(\#rand_{io} \times rand_{cost}, \#P \times seq_{cost})} \tag{33}$$

To calculate the competitive ratio we first consider 2 alternatives: A) when Index Scan is the optimal solution, and B) when Full Scan is the optimal solution, both as a function of the table size (the number of pages in the table). Then, we compare Smooth Scan against a theoretical bound - an Oracle that fetches only pages it needs with a sequential pattern. This is a pure theoretical bound that gives the best possible theoretical performance. [4]

A) Index Optimal Solution. By assuming Index Scan is the optimal solution, Eq. (33) becomes:

$$CR = 1 + \frac{(\#P - \#rand_{io}) \times seq_{cost}}{\#rand_{io} \times rand_{cost}} \tag{34}$$

In this case, a CR is a monotonically increasing sublinear function that for $rand_{cost} = 10$ and $seq_{cost} = 1$ (which corresponds to characteristics of HDDs) and $\#P >= 500$ starts from a degradation of a factor of 5 and increases to a factor of 72 for $\#P = 10^4$ (depicted in Figure 4a).

[4]The Oracle mimics the behavior of Sort Scan, while ignoring the sorting overhead.

B) Full Scan Optimal Solution. Assuming Full Scan is the optimal solution, Eq. (33) becomes:

$$CR = 1 + \frac{\#rand_{io} \times (rand_{cost} - seq_{cost})}{\#P \times seq_{cost}} \tag{35}$$

This function is a monotonically decreasing function, that for $rand_{cost}$ = 10 and seq_{cost} = 1 and $\#P >= 500$ starts from a CR of 1.16 (i.e., 16% of overhead when compared to the optimal solution) and reaches 4% for $\#P = 10^4$ (shown in Figure 4b). This is corroborated in our experiments, showing that Smooth Scan adds an overhead of max 20% when compared to Full Scan. For SSDs ($rand_{cost}$ = 2 and seq_{cost} = 1), this value decreases even more (due to lower discrepancy between random and sequential IO), starting with an overhead of only 7%.

When is A $<$ B. In Figure 4c we show when each of the alternatives is cheaper, for the case when the number of qualifying tuples is equal the number of random jumps (the worst case scenario depicted above).

$$\#rand_{io} \times rand_{cost} < \#P \times seq_{cost} \tag{36}$$

$$\log_2(\#P+1) \times rand_{cost} < \#P \times seq_{cost} \tag{37}$$

For $p >= 60$, $rand_{cost} = 10$ and $seq_{cost} = 1$ the inequality above holds, which means that for the number of pages larger than 60, Index Scan is the optimal solution for this use case scenario, which unfortunately puts a high soft bound on the worst case performance.

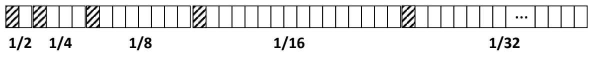
A similar sublinear function (with a higher degradation) is seen when comparing Smooth Scan against the optimal Oracle solution that fetches only needed pages with a sequential pattern (Figure 4d). The CR of Smooth Scan, when compared to Oracle, starts with a factor of 64 for 500 pages and reaches the value 760 for $\#P = 10^4$.

Discussion. From the competitive analysis we have seen that Greedy Smooth Scan is not a viable option for low selectivity since it can introduce too much overhead due to the high number of fault pages it fetches unnecessarily.

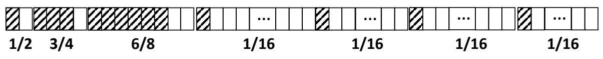
5.1.2 *Selectivity Increase Driven Policy*

Selectivity Increase Driven Policy uses selectivity to drive morphing, i.e., every time a local selectivity increase is noticed, the size of the morphing region gets increased. Figure 5b depicts the worst case distribution for this policy. With this policy, an initial high selectivity can mislead Smooth Scan to keep a high region size (in Figure 5b a morhing size of 16 is kept for the rest of the interval).

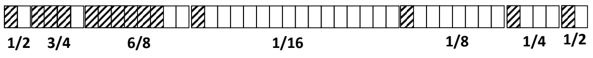
In order to increase the morphing size, SI Smooth Scan has to notice the selectivity increase over the last morphing region bigger than the selectivity seen so far (calculated in Eq. (1) and Eq. (2)). A minimal selectivity sequence that will trigger the morphing size increase has to be a sequence 1/2, 3/4, 6/8, 12/16, ..., $3 * 2^{i-2}$ $/2^i$, where the divisor denotes the size of the current morphing region and the dividend denotes the number of pages containing results in this region. Eq. (38) calculates the number of pages containing



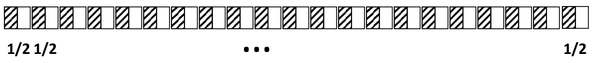
(a) Worst Case Distribution for Greedy Smooth Scan



(b) Worst Case Distribution for SI Smooth Scan

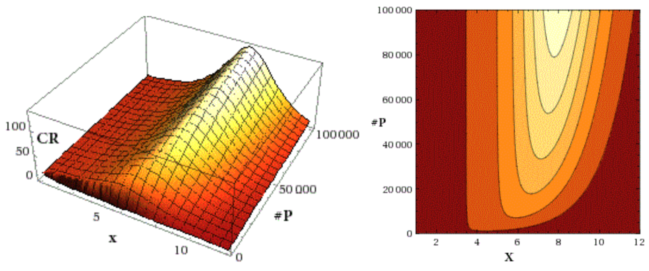


(c) Highest Page Miss Rate for Elastic Smooth Scan



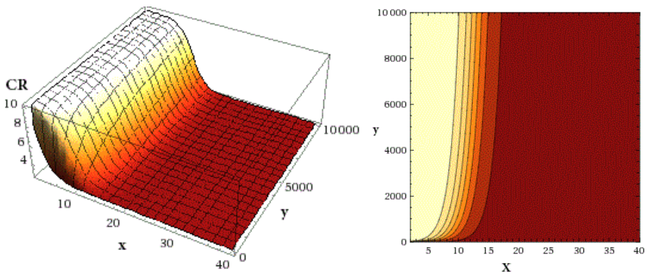
(d) Worst Case Distribution for Elastic Smooth Scan

Figure 5: Worst Case Result Distributions for Smooth Scan.



(a) CR against Oracle (b) Equidistant Contours

Figure 6: Competitive Analysis of SI Smooth Scan.



(a) CR against Oracle (b) Equidistant Contours

Figure 7: Competitive Analysis of Elastic Smooth Scan for Initial Selectivity Increase.

results needed to trigger such a behavior. After performing the morphing expansion x times, the remaining y morphing regions have a single match. The total number of accesses is shown in Eq. (39). In the following equations we replace y with Eq. (40) (derived from Eq. (39)). Since the total cost of Smooth Scan depends on both x and y, and since we can show y as $f(\#P,x)$, in Figure 6 we plot the Competitive Ratio against Oracle as a function of x and $\#P$; we plot the CR for characteristics of HDDs ($rand_{cost}$ = 10 and seq_{cost} = 1).

For this use case scenario a CR is a monotonically increasing sublinear function that reaches a value of 100 for 100K pages for the x peak value of 8, i.e., for 8 morphing increase steps. We have experimented with higher page numbers for which we noticed a higher absolute value of CR with the x peak translated on the right.[5] Nonetheless, the overall trend is similar. The CR is a monotonically increasing sublinear function, which puts a soft-bound on the worst case performance of SI Smooth Scan. The same trend is noticed in the case of SSDs; the only difference is that the equidistant contours are a bit thinner.

$$\begin{aligned}
\#P_{res} &= 1+\sum_{i=0}^{x-2} 3\times 2^i+\sum_{i=1}^{y} 1 \\
&= 1+3*(2^{x-1}-1)+y && (38)\\
\#P &= \sum_{i=1}^{x} 2^i + 2^x * y \\
&= 2*(2^x-1)+2^x*y = 2^x*(2+y)-2 && (39)\\
y &= \frac{\#P+2}{2^x}-2 && (40)\\
\#rand_{io} &= x+y \\
SS_{cost} &= \#rand_{io}\times rand_{cost} \\
&+ (\#P-\#rand_{io})\times seq_{cost} && (41)\\
CR &= \frac{SS_{cost}}{\#P_{res}\times seq_{cost}} && (42)
\end{aligned}$$

5.1.3 *Elastic Policy*

Elastic Policy follows the selectivity pattern of the access, i.e., it increases the morphing size in the dense regions, and decreases it back in the sparse regions.

[5]This is expected since with more pages we can increase the morphing region size to a higher value, for which we need more steps.

High Page Miss Rate of Elastic Smooth Scan. In order to increase the morphing size, Smooth Scan has to notice the same selectivity increase pattern as the one described in Eq. (38). The behavior of Smooth Scan however differs in this case, since after noticing the selectivity drop, Elastic Smooth Scan progresively decreases the morphing size back to the value of 1 page. Therefore, Elastic Smooth Scan performs x times the region morphing increase and x times the region morphing decrease, after which it continues with the morphing size of 1 for the $(y-x)$ remaining tuples (assuming no local selectivity increase is noticed again). Eq. (43) calculates the total number of pages accessed.

$$\begin{aligned}
\#P &= \sum_{i=1}^{x} 2^i+\sum_{i=0}^{x} 2^i+(y-x) \\
&= 2*(2^x-1)+2^{x+1}-1+y-x \\
&= 2^{x+2}-3+y-x && (43)\\
\#rand_{io} &= x+y \\
SS_{cost} &= \#rand_{io}\times rand_{cost} \\
&+ (\#P-\#rand_{io})\times seq_{cost} && (44)\\
CR &= \frac{SS_{cost}}{\#P_{res}\times seq_{cost}} && (45)
\end{aligned}$$

Figure 7 shows a CR against Oracle for this use case and the characteristics of HDD (as a function of x and y; $\#P$ could be derived from Eq. (43)). The CR is a monotonically decreasing function that from an initial value of 10 for 1 random access, converges to a factor of 2 for $x > 10$ (which is expected to happen in reality). From this experiment we have seen that Elastic Smooth Scan has an expected CR of 2 for the use case for which SI Smooth Scan has a soft bound, hence it is the better alternative.

The highest number of page misses happens when the distribution is such that the number of pages in each morphing region for one half of the table is just enough to perform the expansion; after visiting this half the selectivity drops sharply with having only one resulting page per the remaining (shrinking) regions. Figure 5c depicts such a distribution. We calculate the CR for this scenario. Our analysis shows the theoretical bound of 2.45 for 100 pages that decreases to the value of 2.0001 for 3M pages, which corroborates our previous analysis.

Worst case CR for Elastic Smooth Scan. The previous experiment showed the worst scenario with respect to the number of fault page reads. Nonetheless, this is not the scenario with the worst case CR. The worst case for Elastic Smooth Scan appears when the number of random I/O accesses is maximized. This happens when the access is such that every second page has a result match (illustrated in Figure 5d). In this case, Elastic Smooth Scan keeps the morphing size of 2, since it never detects the local selectivity increase when compared to the one over so far seen pages. Therefore, Smooth Scan will perform $\#P/2$ random accesses, and the same amount of sequential accesses (to fetch adjacent pages).

$$\#rand_{io} = \frac{\#P}{2} \tag{46}$$

$$\begin{aligned} SS_{cost} &= \#rand_{io} \times rand_{cost} \\ &+ (\#P - \#rand_{io}) \times seq_{cost} \end{aligned} \tag{47}$$

$$\begin{aligned} CR &= \frac{SS_{cost}}{\min\left(\frac{\#P}{2} \times rand_{cost}, \#P \times seq_{cost}\right)} \\ &= \frac{\frac{\#P}{2} \times (rand_{cost} + seq_{cost})}{\min\left(\frac{\#P}{2} \times rand_{cost}, \#P \times seq_{cost}\right)} \\ &= \frac{(rand_{cost} + seq_{cost})}{\min(rand_{cost}, 2 \times seq_{cost})} \\ &= \frac{11}{2} = 5.5 \end{aligned} \tag{48}$$

The CR is calculated in Eq. (48). For characteristics of HDDs, with $rand_{cost} = 10$ and $seq_{cost} = 1$, the competitive ratio reaches the value of 5.5 when compared to Full Scan. The same ratio decreases in the case of SSDs ($rand_{cost} = 2$ and $seq_{cost} = 1$), reaching a factor of 3. The theoretical bound in this case is 11 for HDDs and 6 for SSDs, and is purely driven by the ratio between the random and sequential access, i.e., it is constant regardless of the table size.

Morphing increase size. A higher morphing increase factor than 2, leads to a higher Competitive Ratio. For instance, for this use case scenario, the morphing increase factor of 10 on HDD gives a competitive ratio of 19. Therefore, we have decided to use a factor of 2 as the morphing increase factor in the rest of the paper.

Discussion. Overall, Elastic Smooth Scan proves to be the most robust solution. This policy provides a strong firm bound on the suboptimality with the maximum theoretical CR of a factor of 11 in the case of HDDs and a factor of 6 in the case of SSDs regardless of the table size. Empirically, we have observed a CR of 2 in our experiments, which makes the operator even more appealing.

6. EXPERIMENTAL EVALUATION

We now present a detailed experimental analysis of Smooth Scan. We demonstrate that Smooth Scan achieves robust performance in a range of synthetic and real workloads without having accurate statistics, while existing approaches fail to do so. Furthermore, Smooth Scan proves to be competitive with existing access paths throughout the entire selectivity range, making it a viable replacement option.

6.1 Experimental Setup

Software. Our adaptive operators are implemented inside PostgreSQL 9.2.1 DBMS. To demonstrate the problem of robustness

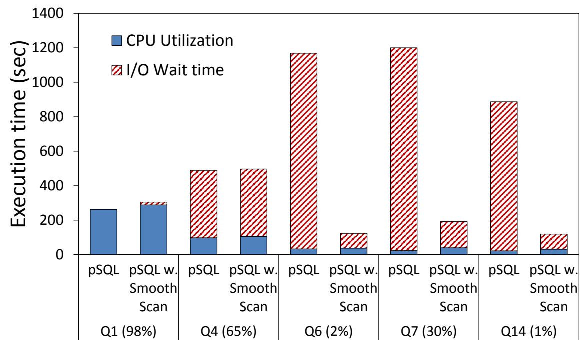


Figure 8: Improving performance of TPC-H with Smooth Scan.

Table 2: I/O Analysis

	Q1		Q4		Q6		Q7		Q14	
	pSql	SS	pSql	SS	pSql	SS	pSql	SS	pSql	SS
#I/O Req.(K)	71	77	225	235	566	95	745	124	416	87
Read data(GB)	8.9	10.2	10.9	12.1	8.7	8.8	11.6	11.6	6.8	8.9

presented in Section 1 we use a state-of-the-art row-store DBMS we refer to as DBMS-X.

Benchmarks. We use two sets of benchmarks to showcase algorithm characteristics: a) for stress testing we use a micro-benchmark, and b) to understand the behavior of the operators in a realistic setting we use the TPC-H benchmark SF 10 [39].

Hardware. All experiments are conducted on servers equipped with 2 x Intel Xeon X5660 Processors, @2.8 GHz (with L1 32KB, L2 256KB, L3 12MB caches), with 48 GB RAM, and 2 x 300 GB 15000 RPM SAS disks (RAID-0 configuration) with an average I/O transfer rate of 130 MB/s, running Ubuntu 12.04.1. In all experiments we report cold runs; we clear database buffer caches as well as OS file system caches before each query execution.

6.2 TPC-H analysis

TPC-H in DBMS-X. In Figure 1 in Section 1, we demonstrated the severe impact of sub-optimal index choices on the overall TPC-H workload. For this experiment, we used the tuning tool provided as part of DBMS-X, with 5GB of space allowance (1/2 of the data set size) to propose a set of indices estimated to boost performance of the TPC-H workload. In queries Q12 and Q19, the presence of indices favors a nested loop join when the number of qualifying tuples in the outer table is significantly underestimated, resulting in a significant increase in random I/O to access tuples from the index ("table look-up"), which in turn results in severe performance degradation (factors 400 and 20 respectively). In both cases the access path operator choice is the only change compared to the original plan, i.e., the join ordering stays the same. Smaller degradation as a result of a suboptimal index choice followed by join reordering occurs in several other queries (Q3, Q18, Q21) resulting in an overall workload performance degradation by a factor of 22.

Improving performance with Smooth Scan. We now demonstrate a significant benefit that Smooth Scan brings to PostgreSQL compared to the optimizer's chosen alternatives when running TPC-H queries. Since PostgreSQL does not have a tuning tool, we create the set of indices proposed by the commercial system from the previous experiment (on the same workload). Figure 8 shows the results for 5 interesting TPC-H queries[6] that cover selectivities from

[6]These queries represent "choke points" for testing data access locality [6].

both ends of spectrum. The query execution plans are given in Appendix A. Q1 and Q6 are single table selection queries, with the selectivity of 98% and 2% respectively. Q4 and Q14 are two-table join queries with two selectivity extremes (65% and 1% respectively) when considering the LINEITEM table. The performance greatly depends on the selectivity of this table, since it is the largest. Lastly, we run Q7, a 6-table join. Since Smooth Scan trades CPU utilization for I/O cost reduction, we show the execution breakdown through CPU utilization and I/O wait time (i.e., the blocking I/O in the critical path of execution). Similarly, in Table 2 we show the number of I/O requests issued by the operators, coupled with the amount of data transferred from the disk.

Figure 8 shows that PostgreSQL with Smooth Scan avoids extreme degradation and achieves good performance for all queries. For instance, while plain PostgreSQL suffers in $Q6$ due to a suboptimal choice of an index scan, PostgreSQL with Smooth Scan maintains good performance preventing a degradation of a factor of 10. $Q6$ selects 2% of the data, which in the case of the index scan causes 566K of random (blocking) I/O accesses over the LINEITEM table (shown in Table 2). By flattening (grouping accesses together) and avoiding repeated accesses, Smooth Scan reduces this number to 95K which resulted in much better performance.

On the other hand, in query $Q1$ with selectivity of 98% the plain PostgreSQL chooses Sort Scan (also called Bitmap Heap Scan), which is an optimal path. However, even in this case Smooth Scan introduces only a marginal overhead; it quickly observes a high selectivity and adjusts the execution by forcing sequential accesses. As a result, Smooth Scan adds an overhead of only 14% over the optimal behavior. This overhead is due to periodical random I/O accesses when following pointers from the index, which increased the number of I/O requests to disk pages from 71K to 77K.

In $Q4$, the selectivity of the LINEITEM table is 65%, and PostgreSQL chooses the full scan as the outer table of a nested loop join with a primary key look-up as the inner input. Although Smooth Scan starts with using the index lookup on the outer table as well, it quickly adjusts its access patterns and adds less than 1% of overhead over the optimal solution.

On the contrary, the selectivity of the LINEITEM table in Q14 is around 1%. Both plain PostgreSQL and our implementation start with an index scan as the outer input, joined with an INLJ with ORDERS (a primary key look-up). Unlike the index scan that issues 416K I/O requests, Smooth Scan issues only 87K requests which translates to a performance improvement of a factor of 8. In both join queries, Smooth Scan does not perform any additional page fetching over the inner tables since for each probe we have a single match; thus there is no need to perform additional adjustments, which Smooth Scan correctly detects.

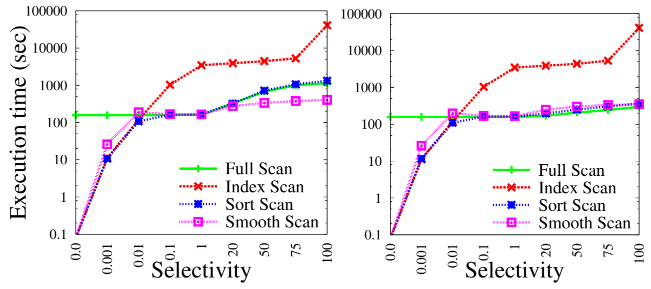
Lastly, an index choice for plain PostgreSQL over the LINEITEM table for a 6-way join in Q7 hurts performance by a factor of 7 compared to the performance of Smooth Scan.

Discussion. Our memory structures span a couple of MB in these experiments. For illustration, the Page ID cache for the LINEITEM occupies 140KB (for 1M pages). Although Smooth Scan can transfer from disk larger amounts of data compared to the original access path, its benefit comes from exploiting the access locality and issuing fewer I/O requests. Overall, Smooth Scan provides robust behavior without requiring accurate statistics. It brings significant gains when the original system makes a wrong decision and only marginal overheads when a correct decision can be made.

6.3 Fine-grained Analysis over Selectivity Range

We now use a micro-benchmark to stress test the various access paths. We compare Smooth Scan against Full Scan, Index Scan and Sort Scan to demonstrate the robust behavior of Smooth Scan. All experiments are run on top of our extension of PostgreSQL, thus Full Scan, Index Scan and Sort Scan are the original PostgreSQL access paths. The micro-benchmark consists of a table with 10 integer columns randomly populated with values from an interval $0-10^5$. The first column is the primary key identifier, and is equal to a tuple order number. The table contains 400M tuples, and occupies 25GB of disk space for 3M pages of 8KB size (PostgreSQL's default value). In addition to the primary key, a non-clustered index is created on the second column ($c2$). We run the following query:

```
Q1: select * from relation where c2>= 0 and c2<X%
  [order by c2 ASC];
```



(a) With order by (b) W/o order by

Figure 9: Smooth Scan vs. Alternatives w. and w/o. Order By.

Supporting an interesting order. In this experiment, we show that Smooth Scan serves its purpose of being an index access path; it maintains tuple ordering and hence outperforms other alternatives for queries (or sub-plans) that require the ordering of tuples. Figure 9a shows the performance of all alternative access paths for a query with an order by clause. The performance of Index Scan degrades quickly due to repeated and random I/O accesses. For selectivity 0.1% its execution time is already 10 times higher than Full Scan, reaching a factor of more than a 100 for 100% selectivity. Sort Scan solves the problem of repeated and random accesses, while at the same time fetching only the heap pages that contain results; therefore, it is the best alternative for selectivity below 1%. Nonetheless, its sorting overhead grows and for selectivity above 2.5% it is not beneficial anymore. Smooth Scan is between the alternatives when selectivity is below 2.5%, while it achieves the best performance for the selectivity above this level. This is due to avoiding the overhead of posterior sorting of tuples to produce results respecting the interesting order, from which Full Scan and Sort Scan suffer.

Without an interesting order. Figure 9b shows the performance of the access paths when executing $Q1$ without the order by clause. For selectivity between 0 and 2.5% the behavior of the operators is the same as in the previous experiment. For higher selectivity, however, Full Scan is the best alternative, since it performs a pure sequential access. Both Sort Scan and Smooth Scan, however, manage to maintain good performance. The overhead of Sort Scan is attributed to the pre-sort phase of the tuples obtained from the index; after that the access is nearly sequential (page IDs are monotonically increasing). Smooth Scan does not suffer from the sorting overhead, but it does suffer from a periodical random I/O accesses driven by the index probes, adding less than 20% overhead when compared to Full Scan for 100% selectivity. A different behavior is observed when the experiment is run on an SSD (shown in Figure 15), where Smooth Scan benefits much more compared to Sort Scan (by a factor of 3).

Discussion. Smooth Scan bridges the gap between existing access paths. Its performance does not degrade when selectivity in-

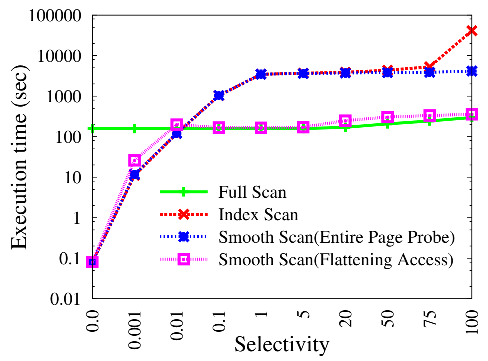
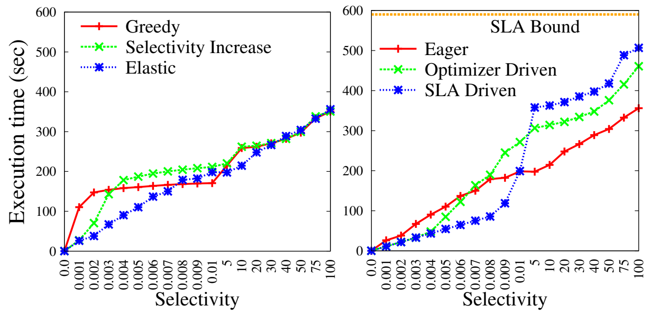


Figure 10: Sensitivity Analysis of Smooth Scan Modes.



(a) Morphing Policies (b) Triggering Points
Figure 11: Impact of Policy and Trigger Choices.

creases, like in the case of Index Scan. This is particularly important in real-life scenarios where a degradation in Index Scan causes performance drops of several orders of magnitude [19]. At the same time, Smooth Scan does not pay the cost of Full Scan to select just a few tuples, which is important for point queries for which Full Scan is not practical. When the order is not imposed the absolute performance of Smooth Scan is comparable to that of Sort Scan; nonetheless, the benefit of Smooth Scan becomes visible when considering its placement in the query plan. Unlike Sort Scan, Smooth Scan adheres to the pipelining model, which is important since the access path operators are executed first and can stall the rest of the stack. When selectivity is below 0.01%, Smooth Scan's Competitive Ratio reaches a factor of 2 over the optimal solution. To put absolute numbers in perspective, in our experiment a maximal overhead of 60 seconds is paid to prevent a worst case performance degradation of 11 hours. In decision support systems that are characterized by long running queries, this overhead is likely to be tolerable as a robustness guarantee for the prevention of severe performance drops that frequently happen due to data correlations and skew.

6.4 Sensitivity analysis of Smooth Scan

We now study the parameters that affect the performance of Smooth Scan such as the impact of its morphing modes, policies, and strategies. We show the bookkeeping overhead and study the Smooth Scan effect on HDDs versus SSDs. For all experiments in this section, unless stated otherwise, we use $Q1$ from the micro-benchmark without an order by clause.

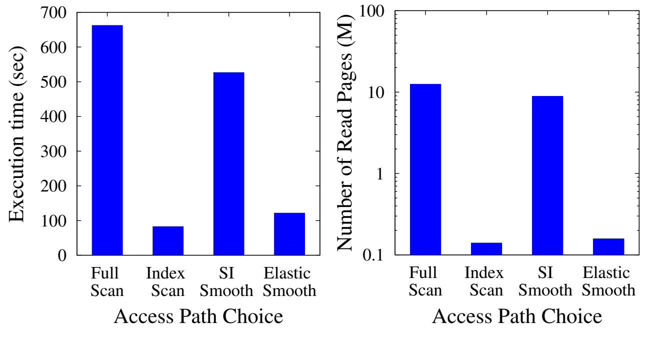
Impact of the Entire Page Probe Mode. The pointer chasing of non-clustered indices when performing a tuple look-up in general hurts performance when the selectivity increases. Figure 10 depicts the improvement that Smooth Scan achieves by removing repeated accesses when executing query $Q1$ from the micro-benchmark. The curve of Smooth Scan denoted as the Entire Page Probe morphs only until Mode 1. Smooth Scan improves by a factor of 10 when compared to Index Scan for selectivity 100%. The performance of Smooth Scan degrades with selectivity increase up to 1%; this is the point where approximately all pages have been read. With 120 tuples per page (64-byte tuples in 8KB pages) and uniform distribution, we expect one tuple from each page to qualify. After that point the execution time stays nearly flat with the increase of 20% for 100% selectivity, showing that the overhead of reading the remaining tuples from a page is dominated by the time needed to fetch a page from disk. The execution time of Smooth Scan when morphing only up to Mode 1, is however still significantly higher (a factor of 14) compared to Full Scan for 100% selectivity. This is due to the discrepancy between random and sequential page accesses; the former being an order of magnitude slower in the case of HDD.

Impact of the Flattening Access Mode. To alleviate the random access problem, Smooth Scan employs Mode 2+ (shown in Figure 10 as the Flattening Access curve). By fetching adjacent pages Smooth Scan amortizes access costs at the expense of extra CPU cost to go through all the fetched data. We perform a sensitivity analysis on the maximum number of adjacent pages up to which we perform the morphing expansion. Our experiments show that 2K pages are optimal (translates to a block size of 16MB); thus we keep this value as the maximum region size for the rest of the experiments. Smooth Scan with Flattening Access is not only much better than Index Scan (by a factor of 115) but also nearly approaches the behavior of Full Scan; in the worst case of selectivity 100% Smooth Scan is only 20% slower than Full Scan.

Impact of Policy Choices. We plot the impact of policy choices in Figure 11a. The Greedy policy morphs with each index probe, and hence converges to the full scan faster than other policies. For lower selectivity the Selectivity Increase and Elastic policies introduce less overhead compared to the Greedy since they fetch fewer adjacent pages, i.e., more pages need to be seen for the morphing size to increase. This particularly holds for the Elastic policy that adjusts the morphing size depending on the selectivity of the fetched regions. Since it is the most adaptive to the changes in the operator selectivity, we favor it in the rest of the experiments.

Impact of Trigger Choices. In Figure 11b, we plot the impact of triggering strategy choices. The Eager strategy starts immediately with Smooth Scan; in this case we plot the Elastic Smooth Scan. The Optimizer Driven strategy starts with the traditional index and changes to Smooth Scan after 15K tuples (the optimizer's estimated cardinality), causing the increase in the execution time for selectivity 0.005%. After the shift to Smooth Scan, for this experiment we continue with the Selectivity Increase Driven policy. The overhead of the Optimizer Driven strategy increases for higher selectivity compared to the Eager strategy and is attributed to a tuple check for each tuple produced with Smooth Scan, and to additional repeated accesses of the same pages accessed before the Smooth Scan behavior is triggered. On the other hand the initial execution time is lower compared to the Eager strategy due to fewer page accesses. Similar behavior is observed with the SLA driven triggering strategy, with a sharper cliff for point 0.009%, since with this strategy we switch immediately to Greedy. For this experiment we have set an upper performance bound equal to the performance of 2 full scans as the SLA constraint; the calculated bound is shown as the orange dotted line in Figure 11b. According to the model the morphing triggering point is 32K tuples, which guarantees the execution time just slightly below the SLA bound for 100% selectivity.

Overall, the Eager strategy strikes a balance in terms of overall performance. However, if we are in an environment where respect-



(a) Execution Time (1% sel.) (b) Number of Read Pages
Figure 12: Handling Skew.

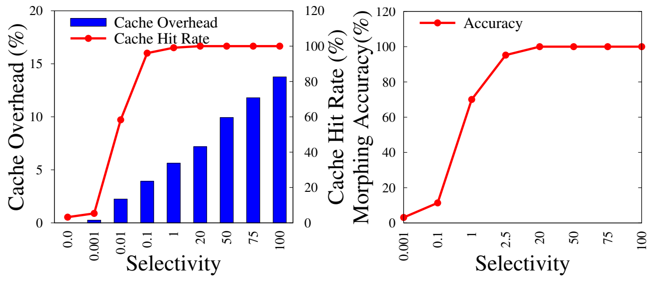
ing SLA is the main priority, or Smooth Scan serves as a means of fixing sub-optimal decisions then an SLA or Optimizer strategy are viable alternatives; we easily turn a strategy knob depending on the applications requirements.

Adjusting to Skew Distribution. Smooth Scan has demonstrated the ability to prevent execution time blow-up due to selectivity increase. Many modern applications, however, exhibit non-uniform data distributions (stock markets, internet networks, etc.). For these applications one execution strategy is not likely to optimally serve the entire table. We show that Smooth Scan can adapt well to skewed distribution of values across pages. We use the Elastic policy and compare it against the Selectivity Increase (SI) policy.

We use a table with 1.5B tuples, 10 integer columns (random values from [0-10^5]) that occupy 100GB, and create a secondary index on the second column ($c2$). First 15M tuples have $c2 = 0$; afterwards another 0.001% of random tuples have value 0. The result selectivity is slightly above 1%, with most of the tuples coming from the pages placed at the beginning of the relation heap, i.e. we read all tuples where $c2 = 0$.

Figure 12a plots the execution time of Index Scan, Full Scan, SI and Elastic Smooth Scan; Figure 12b plots the number of distinct pages fetched to answer the query. From Figure 12b we see that SI Smooth Scan fetches 56 times more pages than Elastic Smooth Scan, and it is 5 times slower. The large number of pages is due to the initial skew; SI Smooth Scan notices the high selectivity increase at the beginning, and in order to reduce the potential degradation it continues fetching big chunks of sequentially placed page, ultimately fetching 8.8M out of 12.5M pages. On the contrary, after the dense region, Elastic Smooth Scan decreases the morphing step, quickly converging back to the access of a single page per probe, ultimately ending up with only 150K pages fetched (Index Scan fetches 140K pages; the severe impact of repeated and random I/O is not seen for Index Scan, since the index key follows the page placement on disk). Elastic Smooth Scan, thus, continues providing near-optimal performance, despite the significant initial skew. This is particularly important for long-running queries over big data, where data distribution tends to be non-uniform [30]. Approaches that employ one execution strategy, or run multiple alternatives shortly and kill all but the winning one are likely to make a mistake and not be able to benefit from this density discrepancy; Elastic Smooth Scan, however, adjusts its behavior to fit the data distribution.

Auxiliary Data Structures. To avoid repeated page accesses, Smooth Scan in PostgreSQL uses the data structures described in Section 4.1. We now show the bookkeeping overhead of these structures and their hit rate, demonstrated on Q1 from the micro-benchmark with an ORDER BY clause.



(a) Result Cache Analysis (b) Morphing Accuracy
Figure 13: Analysis of Auxiliary Data Structures.

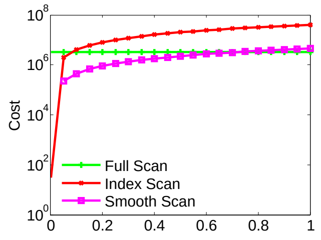
Figure 13a shows that Result Cache adds a maximal overhead of 14% when storing all result matches in the cache (shown as blue bars). At the same, the Result Cache Hit Rate (calculated as the ratio between the number of tuple requests served from the cache and the total number of tuple requests) reaches 100% for 1% selectivity. Figure 13b shows that the morphing accuracy (calculated as the ratio between the number of pages containing result matches and the total number of checked pages with Smooth Scan morphing) gets improved after 1%, reaching 100% for 2.5% selectivity. The overhead of page ID checks remains significantly below 1% in all our experiments, hence we do not show it separately.

Cost Model Analysis. In this experiment we show that the estimates of the analytical model we derived are corroborated with the actually measured performance. Figure 14a and Figure 14b show the cost behavior of Full Scan, Index Scan, and Smooth Scan based on the analytical cost model, as a function of selectivity. The y-axis shows the cost (unit 1 corresponds to one sequential I/O). We model the costs for a table with 400M tuples from the micro-benchmarks. For the page size we take the value of 8KB; for the tuple size we assume 64 bytes (40 bytes of data plus the overhead for the tuple header), and for the key size we use 16 bytes. We assume uniform distribution of result tuples, and approximate the number of random I/O accesses for Mode 2 with $log_2(\#P+1)$. Finally, for seq_{cost} we use 1, for $rand_{cost}$ we use 10, and for cpu_{cost} we use $1/1M$. We separately show the behavior of the operators when selectivity is between 0 and 1%, since for the increasing selectivity both Full Scan and Smooth Scan converge to the same value.

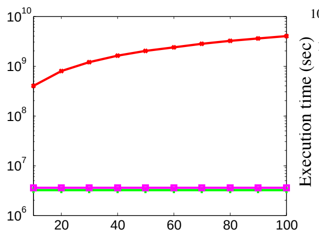
The model suggests that for lower selectivity Smooth Scan behaves like Index Scan, while for higher selectivity it converges to the performance of Full Scan. This is corroborated in our experiments presented in Figure 14c and Figure 14d; they depict the real execution times using the actual data that the model assumed. In both graphs Smooth Scan converges to Full Scan as predicted. The only discrepancy from the model we observe is that Smooth Scan converges faster to Full Scan than estimated. This effect is partly due to the disk controller behavior, grouping many sequential I/O requests from the disk controller queue into one in the case of Full Scan, which puts the performance bar of Full Scan a bit lower than expected. Similar behavior is not observed in the case of Smooth Scan that issues requests for sequential sub-arrays with random jumps in between. Although the same grouping of sequential sub-arrays could happen and equally improve performance, the disk controller did not possess logic to do so.

6.5 Smooth Scan on SSD

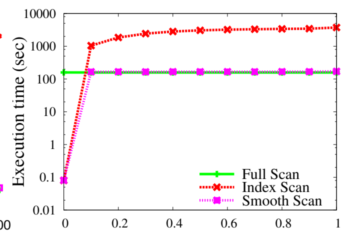
Given the different access costs of solid state disks (SSD), better random access performance, and the forecasts of their potential replacement of HDD [23], we now stress test Smooth Scan on SSD. We use a solid state disk OCZ Deneva 2C Series SATA 3.0 with



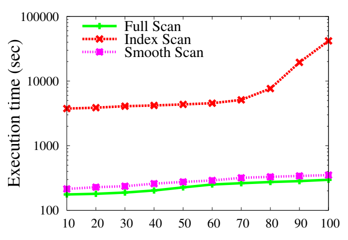
(a) Model: Selectivity 0-1%



(b) Model: Selectivity 1-100%



(c) Measure: Selectivity 0-1%



(d) Measure: Selectivity 1-100%

Figure 14: Comparing the Analytical Model with Execution.

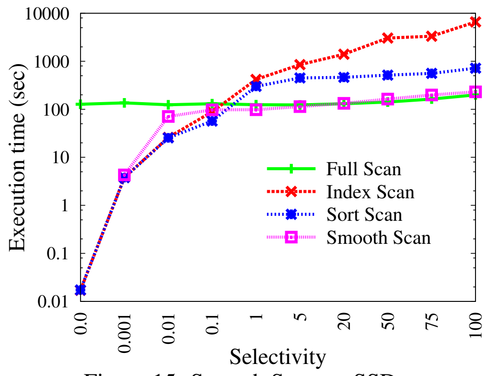


Figure 15: Smooth Scan on SSD.

advertised read performance of 550MB/s (offering 80kIO/s of random reads).

Figure 15 demonstrates that Smooth Scan benefits even more from solid state technology than with hard disks (shown in Figure 9). SSDs are well known for removing mechanical limitations of disks, which enables them to achieve better performance of random I/O accesses. Our analysis for the hardware used in this paper, shows that random I/O accesses are two times slower than sequential accesses on SSD, while this discrepancy reaches a factor of 10 in the case of HDD. This difference makes Index Scan (and Smooth Scan) more beneficial on SSD than on HDD. In our experiments, Index Scan on HDD is beneficial only for selectivity below 0.01%, while on SSD this range increases until 0.1%. For higher selectivity, Index Scan on SSD still loses the battle against other alternatives, since it suffers from repeated accesses and cannot benefit from the flattening pattern compared with other alternatives. Consequently, Index Scan is slower than Smooth Scan by a factor of 30 for 100% selectivity. What is interesting to note is that Sort Scan loses the battle against Smooth Scan for selectivity above 0.1% (even without the imposed order).

Discussion. Smooth Scan favors SSD over HDD, since occasional random jumps when following the index pointers do not hurt performance much, compared to the sorting overhead of Sort Scan to presort tuples. Smooth Scan is faster than Full Scan for selectivity below 20%, and is only 10% slower for 100% selectivity. The smaller gap between random and sequential I/O and the decreased SSD latency, thus makes Smooth Scan a promising solution for the future.

6.6 Switch Scan: A Straw Man Adaptivity

In this section we study the benefit of Switch Scan as a straightforward approach to providing a mid-operator run-time adaptivity. We demonstrate that although a simple solution can help in some cases (SLA), there are consequences behind binary decisions such as performance cliffs or the inability to return once the decision has been made.

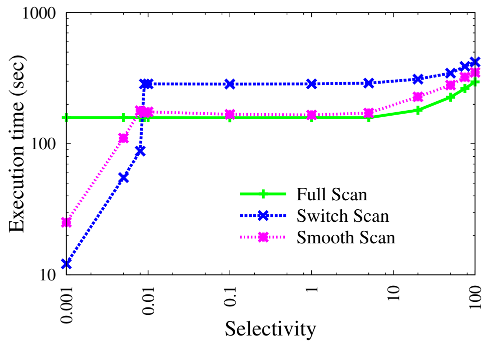


Figure 16: Switch Scan Performance Cliff and Overall Benefit.

Figure 16 shows Switch Scan when executing query $Q1$ from the micro-benchmark. We can observe a performance cliff for 0.009% selectivity, due to the strategy switch. In this example, the optimizer's cardinality estimate is 32K tuples and it decided to employ an index scan. While monitoring the actual cardinality, we observe more than 32K tuples and perform the switch before producing the next result tuple. The execution time to produce 32001 tuples now becomes the execution time of the index seek for 32K tuples plus the execution time of the full table scan. After the switch, Switch Scan performs just like Full Scan, avoiding degradation of more than an order of magnitude when selectivity is 100%. Nonetheless, the moment we opt for the switch, the execution time increases by the time of the full scan, which might not be amortized over the rest of the query's lifetime.

Discussion. Since the decision highly depends on the accuracy of the statistics, this approach is volatile and hence non-robust. Smooth Scan, on the other hand, manages to approach the optimal behavior while being statistics-oblivious.

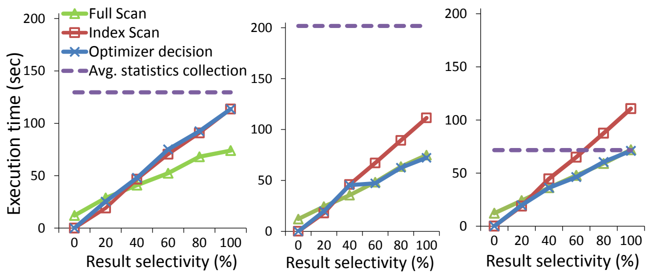
6.7 Need for Intra-operator Adaptivity

An alternative to correcting suboptimal plans with intra-operator adaptivity presented in Section 3 would be to avoid wrong paths in the first place. One could argue this could be achieved by having perfectly accurate statistics representing underlying data; we show, however, that repeatedly collecting statistics is prohibitively expensive, since this effort usually involves a full table access.

Experimental setting. For this experiment we use a table with 40M tuples from the micro-benchmark, with a non-clustered index built on columns $(c2, c3)$. Throughout the experiment we employ the following query:

```
Q2: select * from relation where c2 = X and c3 = X;
```

We perform a constant update of data introducing the skew between columns $c2$ and $c3$ (we update both columns to value X). With this setting we want to simulate a sensor processing environment where data is ingested constantly 24/7, causing a frequent change of data



(a) Basic statistics (b) Single column (c) Column group

Figure 17: Statistics Collection Alternatives in DBMS-X.

statistics. Completely accurate statistics representing underlying data are rarely present in such a system.

Figure 17 shows the statistics collection times on the table, comparing them against the execution time of query $Q2$ run on DBMS-X[7]. The three graphs demonstrate the three levels of database statistics, namely: a) base statistics (the table size, tuple size, number of tuples, etc.); b) single column distribution statistics (histograms on each column separately); c) joint-data distributions (a histogram on the group of columns from the query ($c2$, $c3$)).

Statistics types analysis. Despite being the cheapest alternative, the basic statistics could still lead to the choice of suboptimal plans as shown in Figure 17a since they cannot accurately detect neither skew nor the presence of column correlations. On the other hand, one could observe that obtaining histograms on all columns introduces a higher cost as shown in Figure 17b. Having histograms on all columns could solve the problem of suboptimal decisions in the case of skewed data. Nevertheless, it will still not detect the correlation between different columns (notice the sub-optimal decision for selectivity 40% in Figure 17b). Therefore, whenever a query contains multiple filtering predicates over different columns, joint-data distributions are required. Figure 17c shows the statistics collection time on the group of two columns from the query. Performing this collection once could be tolerated. Calculating all possible joint distributions for the workload consisting of many queries, however, is an unattainable goal, especially since applications today have hundreds of columns in each table [37].

Incomplete statistics hurt performance. Query $Q2$ is a simple query that showcases the problem with existing DBMS. Assuming no accurate statistics exist on the table, the optimizer would fall into a trap of using the non-clustered index regardless of the actual result cardinality. This is happening because the uniformity assumption assumes the selectivity of each predicate to be 10^{-5} (1/100K), while the independence further assumes the overall selectivity to be 10^{-10} ($10^{-5} * 10^{-5}$) [10]. Therefore, the optimizer would always opt for the non-clustered index look-up, severely hurting performance in the case of higher selectivity.

7. RELATED WORK

Smooth Scan draws inspiration from a large body of work on adaptive and robust query processing. We briefly discuss the work more related to our approach, while for a detailed summary the interested reader may refer to [13].

Statistics Collection. Since the quality of plans directly depends on the accuracy of data statistics, a plethora of work has studied techniques to improve the statistics accuracy in DBMS. Modern approaches employ the idea of monitoring execution to exploit this information in future query compilations [1, 8, 37]. In dynamically changing environments, however, statistical information rarely stays unchanged between executions; consider data ingest different devices produce (e.g. smart meters [25], data from Facebook, etc.) for instance. Orthogonal techniques focused on modeling the uncertainty about the estimates durign query optimization [4, 5]. Overall, considering the two-dimensional change in the workload characteristics (frequent data ingest, and ad-hoc queries) in modern applications, and the high price of having up-to-date statistics for all cases in the exponential search space [8, 9], the risk of having incomplete or stale statistics still remains high.

Single-plan Adaptive Approaches. From the early prototypes to most modern database systems, query optimizers determine a single best execution plan for a given query [35]. To cope with environment changes in such systems, some of the early work on adaptive query processing employed reoptimization techniques in the middle of query execution [30, 32, 34]. Since this re-optimization step can introduce overheads in query execution, an alternative technique proposed in the literature is to choose a set of plans at compile time and then opt for a specific plan based on the actual values of parameters at run-time [22, 29]. A middle ground between reoptimization and dynamic evaluation is proposed in [5, 15], where a subset of more robust plans is chosen for given cardinality boundaries. Regardless of the strategy when to adjust behavior, reoptimization approaches suffer from similar binary decisions that we have seen with Switch Scan; once reoptimization is employed, the strategy switch will almost certainly trigger a performance cliff.

Multi-plan Adaptive Approaches. Some of the early techniques with multi-plan approaches employed competition to decide between alternative plans [2, 24]. Essentially, multiple access paths for a given table are executed simultaneously for a short time and the one that wins is used for the rest of the query plan. In contrast, Smooth Scan does not perform any work that is thrown away later (while all the work done for every access method except the winning one is discarded in the approach of competing plans).

Adaptive and Robust Operators. With workloads being less steady and predictable, coarse-grained index tuning techniques are becoming less useful with the optimal physical design being a moving target. In such environments, adaptive indexing techniques emerged, with index tuning being a continuous effort instead of a one time procedure. Partial indexing [36,38,42] broke the paradigm of building indices on a full data set, by partitioning data into interesting and uninteresting tuples, while indexing only the former. Similarly, but more adaptively using the workload as a driving force, database cracking and adaptive merging techniques [20,26,27] lower the creation cost of indices and distribute it over time by piggybacking on queries to refine indices. Lastly, SMIX indices are introduced as a way to combine index accesses with full table scans, by building covered values trees(CVT) on tuples of interest [40]. Despite bringing adaptivity in index tuning, none of the techniques addresses the index accesses from the aspect of query processing, and hence stayed susceptible to the optimizer's mistakes. The closest to our motivation of achieving robustness in query processing is G-join [18], an operator that combines strong points of join alternatives into one join operator; we, however, consider access path operators and adapt and morph from one operator alternative to another as knowledge about data evolves.

Improving IO Access. Index-lookups cause poor disk performance due to random-access latency. Asynchronous IO with prefetching [16,31] improves performance of such pattern but still suffers from repeated page reads and small access size. Partial sorting of tuples [14, 16] can improve access locality and size, but unless

[7]We have measured statistics collection time on a commercial system, since this system supports a wider spectrum of possibilities than PostgreSQL.

the entire input is sorted, repeated page reads are still possible.

8. CONCLUSION

With the increase in complexity of modern workloads and the technology shift towards cloud environments, robustness in query processing is gaining momentum. With current systems remaining sensitive to the quality of statistics, however, the runtime performance of queries may fluctuate severely even after marginal changes in the underlying data. For a productive user experience, the performance for every query must be robust, i.e., close to the expected performance, even with stale, or insufficient statistics.

This paper introduces Smooth Scan, a *statistics-oblivious* operator that continuously morphs between the two access path extremes: an index look-up and a full table scan. As Smooth Scan processes data during query execution, it understands the properties of the data and morphs its behavior to preferred access path. We implement Smooth Scan in PostgreSQL and through both synthetic benchmarks and TPC-H we show that it achieves near-optimal performance throughout the entire range of possible selectivities.

Acknowledgments

We would like to thank the organizers of the Dagstuhl seminar 12321 on "Robust query processing" for the introduction of robustness issues in query processing. In particular, we thank Goetz Graefe for his support throughout this work. We thank Danica Porobic for the discussions on the cost model part of the work, and Anshuman Dutt, the DIAS lab members and the reviewers for their constructive feedback. This work is partially funded by the European Union Seventh Framework Programme (ERC-2013-CoG), Grant No 617508 (ViDa).

9. REFERENCES

[1] A. Aboulnaga and S. Chaudhuri. Self-tuning Histograms: Building Histograms Without Looking at Data. In *SIGMOD*, 1999.
[2] G. Antoshenkov and M. Ziauddin. Query processing and optimization in Oracle Rdb. *PVLDB*, 5(4):229–237, 1996.
[3] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *SIGMOD*, 2000.
[4] B. Babcock and S. Chaudhuri. Towards a robust query optimizer: a principled and practical approach. In *SIGMOD*, 2005.
[5] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization. In *SIGMOD*, 2005.
[6] P. A. Boncz, T. Neumann, and O. Erling. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *TPCTC*, 2013.
[7] R. Borovica, I. Alagiannis, and A. Ailamaki. Automated physical designers: what you see is (not) what you get. In *DBTest*, 2012.
[8] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. A pay-as-you-go framework for query execution feedback. *PVLDB*, 1(1), 2008.
[9] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Exact cardinality query optimization for optimizer testing. *PVLDB*, 2(1), 2009.
[10] S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *TODS*, 9(2):163–186, 1984.
[11] H. D., P. N. Darera, and J. R. Haritsa. On the production of anorexic plan diagrams. In *VLDB*, 2007.
[12] H. D., P. N. Darera, and J. R. Haritsa. Identifying robust plans through plan diagram reduction. *PVLDB*, 1(1):1124–1140, 2008.
[13] A. Deshpande, I. Zachary, and V. Raman. Adaptive Query Processing. In *Foundations and Trends in Databases*, 2007.
[14] D. J. DeWitt, J. F. Naughton, and J. Burger. Nested Loops Revisited. In *PDIS*, 1993.
[15] A. Dutt and J. Haritsa. Plan Bouquets: Query Processing without Selectivity Estimation. In *SIGMOD*, 2014.
[16] M. Elhemali, C. A. Galindo-Legaria, T. Grabs, and M. M. Joshi. Execution Strategies for SQL Subqueries. In *SIGMOD*, 2007.
[17] G. Graefe. Modern B-Tree Techniques. *Found. Trends databases*, 3(4):203–402, 2011.
[18] G. Graefe. New Algorithms for Join and Grouping Operations. *Comput. Sci.*, 27(1):3–27, 2012.
[19] G. Graefe, A. C. König, H. A. Kuno, V. Markl, and K.-U. Sattler. Robust Query Processing (Dagstuhl Seminar 10381). In *Robust Query Processing*, 2011.
[20] G. Graefe and H. Kuno. Adaptive indexing for relational keys. *ICDEW*, 0:69–74, 2010.
[21] G. Graefe, H. A. Kuno, and J. L. Wiener. Visualizing the robustness of query execution. In *CIDR*, 2009.
[22] G. Graefe and K. Ward. Dynamic query evaluation plans. In *SIGMOD*, 1989.
[23] J. Gray. Tape is Dead, Disk is Tape, Flash is Disk, RAM Locality is King. Presented at CIDR, 2007.
[24] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah. Adaptive Query Processing: Technology in Evolution. *IEEE Data Engineering Bulletin*, 23:2000, 2000.
[25] IBM. Managing big data for smart grids and smart meters. White Paper, http://goo.gl/n1Ijtd, 2012.
[26] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *CIDR*, 2007.
[27] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. *PVLDB*, 4:586–597, 2011.
[28] Y. E. Ioannidis. Query Optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.
[29] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric query optimization. *PVLDB*, 6(2):132–151, 1997.
[30] Z. G. Ives, A. Y. Halevy, and D. S. Weld. Adapting to source properties in processing data integration queries. In *SIGMOD*, 2004.
[31] S. Iyengar, S. Sudarshan, S. K. 0002, and R. Agrawal. Exploiting Asynchronous IO using the Asynchronous Iterator Model. In *COMAD*, 2008.
[32] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.
[33] L. Mackert and G. Lohman. R* optimizer validation and performance evaluation for local queries. In *SIGMOD*, 1986.
[34] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust Query Processing through Progressive Optimization. In *SIGMOD*, 2004.
[35] 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.
[36] P. Seshadri and A. N. Swami. Generalized Partial Indexes. In *ICDE*, 1995.
[37] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *VLDB*, 2001.
[38] M. Stonebraker. The Case for Partial Indexes. *SIGMOD Record*, 18:4–11, 1989.
[39] TPC. TPC-H Benchmark. `http://www.tpc.org/tpch/`.
[40] H. Voigt, T. Kissinger, and W. Lehner. SMIX: self-managing indexes for dynamic workloads. In *SSDBM*, 2013.
[41] A. Wilschut and P. Apers. Dataflow Query Execution in a Parallel Main-memory Environment. In *PDIS*, 1991.
[42] E. Wu and S. Madden. Partitioning techniques for fine-grained indexing. In *ICDE*, 2011.

APPENDIX

A. TPC-H QUERY PLANS

This section shows the TPC-H query execution plans for the experiment presented in Section **??**. For each query, we show the original query execution plan of PostgreSQL and the plan after introducing Smooth Scan into PostgreSQL. The proposed plans are obtained by the ”explain analyze” command of PostgreSQL. For clarity, we omit details such as the optimizer’s cost and time information, while we enclose the cardinality information. The first bracket at the operator level denotes the optimizer’s estimated cardinality, while the second bracket with the prefix ”actual” contains the actual cardinality information measured at run-time. As expected both plans, the original and the plan where any decision on the access path is replaced by placing Smooth Scan, return the same number of records.

A.1 Q1

PostgreSQL:

```
Sort  (rows=13334)(actual rows=4 loops=1)
   Sort Key: l_returnflag, l_linestatus
   Sort Method: quicksort  Memory: 25kB
   -> HashAggregate  (rows=13334) (actual rows=4 loops=1)
         -> Seq Scan on lineitem  (rows=19995351) (actual rows=59047103 loops=1)
               Filter: (l_shipdate <= '1998-08-28'::timestamp)
               Rows Removed by Filter: 938949
```

PostgreSQL with Smooth Scan:

```
Sort  (rows=13334)(actual rows=4 loops=1)
   Sort Key: l_returnflag, l_linestatus
   Sort Method: quicksort  Memory: 25kB
   -> HashAggregate  ( rows=13334) (actual rows=4 loops=1)
         -> Index Smooth Scan using idx1202121036090 on lineitem  (rows=19995351) (actual rows=59047103 loops=1)
               Index Cond: (l_shipdate <= '1998-08-28'::timestamp)
               Rows Removed by Filter: 938949
```

A.2 Q4

PostgreSQL:

```
Sort  (rows=1) (actual rows=5 loops=1)
   Sort Key: orders.o_orderpriority
   Sort Method: quicksort  Memory: 25kB
   -> HashAggregate  (rows=1)(actual rows=5 loops=1)
         -> Nested Loop  (rows=37500) (actual rows=525621 loops=1)
               -> HashAggregate  (rows=67) (actual rows=13753474 loops=1)
                     -> Seq Scan on lineitem  (rows=19995351) (actual rows=37929348 loops=1)
                           Filter: (l_commitdate < l_receiptdate)
                           Rows Removed by Filter: 22056704
               -> Index Scan using sql120209155202560 on orders  (rows=1)  (actual  rows=0 loops=13753474)
                     Index Cond: (o_orderkey = lineitem.l_orderkey)
                     Filter: ((o_orderdate >= '1994-07-01'::date) AND (o_orderdate <'1994-10-01'))
                      Rows Removed by Filter: 1
```

PostgreSQL with Smooth Scan:

```
Sort  (rows=1) (actual rows=5 loops=1)
   Sort Key: orders.o_orderpriority
   Sort Method: quicksort  Memory: 25kB
   -> HashAggregate  (rows=1)(actual rows=5 loops=1)
         -> Nested Loop  (rows=37500) (actual rows=525621 loops=1)
               -> HashAggregate  (rows=67) (actual rows=13753474 loops=1)
                     -> Index Smooth Scan using sql120209154437510 on lineitem
                     (rows=19995351) (actual rows=37929348 loops=1)
                           Filter: (l_commitdate < l_receiptdate)
                           Rows Removed by Filter: 22056704
               -> Index Smooth Scan using sql120209155202560 on orders  (rows=1) (actual  rows=0 loops=13753474)
                     Index Cond: (o_orderkey = lineitem.l_orderkey)
                     Filter: ((o_orderdate >= '1994-07-01'::date) AND (o_orderdate <'1994-10-01'))
                     Rows Removed by Filter: 1
```

A.3 Q6

PostgreSQL:

```
Aggregate  (rows=1) (actual rows=1 loops=1)
   -> Index Scan using idx1202121036090 on lineitem  (rows=500) (actual rows=1195577 loops=1)
         Index Cond: ((l_shipdate >= '1996-01-01'::date) AND (l_shipdate < '1997-01-01')
                                  AND (l_discount >= 0.02) AND (l_discount <= 0.04))
         Filter: (l_quantity < 25::numeric)
         Rows Removed by Filter: 1293437
```

PostgreSQL with Smooth Scan:

```
 Aggregate  (rows=1) (actual rows=1 loops=1)
   ->  Index Smooth Scan using idx1202121036090 on lineitem  (rows=500) (actual rows=1195577 loops=1)
         Index Cond: ((l_shipdate >= '1996-01-01'::date) AND (l_shipdate < '1997-01-01')
                                  AND (l_discount >= 0.02) AND (l_discount <= 0.04))
         Filter: (l_quantity < 25::numeric)
         Rows Removed by Filter: 1293437
```

A.4 Q7

PostgreSQL:

```
Sort  (rows=4) (actual rows=4 loops=1)
   Sort Key: n1.n_name, n2.n_name, (date_part('year'::text, (lineitem.l_shipdate)::timestamp without time zone))
   Sort Method: quicksort  Memory: 25kB
   ->  HashAggregate  (rows=4) (rows=4 loops=1)
         ->  Nested Loop  (rows=16) (actual rows=58258 loops=1)
               Join Filter: ((customer.c_nationkey = n2.n_nationkey)
                            AND (((n1.n_name = 'EGYPT'::bpchar) AND (n2.n_name = 'CHINA'::bpchar))
                        OR ((n1.n_name = 'CHINA'::bpchar) AND (n2.n_name = 'EGYPT'::bpchar))))
                        Rows Removed by Join Filter: 2846110
               ->  Nested Loop  (rows=2999) (actual rows=1452184 loops=1)
                     ->  Nested Loop  (rows=2999) (actual rows=1452184 loops=1)
                           ->  Hash Join  (rows=2999) (actual rows=1452184 loops=1)
                                 Hash Cond: (lineitem.l_suppkey = supplier.s_suppkey)
                                 ->  Index Scan using idx1202121036090 on lineitem
                                        (rows=299930) (actual rows=18230325 loops=1)
                                       Index Cond: ((l_shipdate >= '1995-01-01'::date)
                                                AND (l_shipdate <= '1996-12-31'::date))
                                 ->  Hash  (rows=1000) (actual rows=7969 loops=1)
                                       Buckets: 1024  Batches: 1  Memory Usage: 483kB
                                       ->  Hash Join  (rows=1000) (actual rows=7969 loops=1)
                                             Hash Cond: (supplier.s_nationkey = n1.n_nationkey)
                                             ->  Seq Scan on supplier (rows=100000)(actual rows=100000 loops=1)
                                             ->  Hash  (rows=2)(actual rows=2 loops=1)
                                                   Buckets: 1024  Batches: 1  Memory Usage: 1kB
                                                   ->  Seq Scan on nation n1  (rows=2) (actual rows=2 loops=1)
                                                         Filter: ((n_name = 'EGYPT'::bpchar)
                                                         OR (n_name = 'CHINA'::bpchar))
                                                          Rows Removed by Filter: 23
                           ->  Index Scan using sql120209155202560 on orders
                                 (rows=1) (actual rows=1 loops=1452184)
                                  Index Cond: (o_orderkey = lineitem.l_orderkey)
                     ->  Index Only Scan using idx1202121037110 on customer
                        (rows=1) (actual  rows=1 loops=1452184)
                           Index Cond: (c_custkey = orders.o_custkey)
                           Heap Fetches: 1452184
               ->  Materialize  (rows=2) (actual rows=2 loops=1452184)
                     ->  Seq Scan on nation n2  (rows=2) (actual rows=2 loops=1)
                           Filter: ((n_name = 'CHINA'::bpchar) OR (n_name = 'EGYPT'::bpchar))
                           Rows Removed by Filter: 23
```

PostgreSQL with Smooth Scan:

```
Sort  (rows=4) (actual rows=4 loops=1)
   Sort Key: n1.n_name, n2.n_name, (date_part('year'::text, (lineitem.l_shipdate)::timestamp without time zone))
   Sort Method: quicksort  Memory: 25kB
   ->  HashAggregate  (rows=4) (rows=4 loops=1)
         ->  Nested Loop  (rows=16)  (actual rows=58258 loops=1)
               Join Filter: ((customer.c_nationkey = n2.n_nationkey)
                        AND (((n1.n_name = 'EGYPT'::bpchar) AND (n2.n_name = 'CHINA'::bpchar))
                        OR ((n1.n_name = 'CHINA'::bpchar) AND (n2.n_name = 'EGYPT'::bpchar))))
                        Rows Removed by Join Filter: 2846110
               ->   Nested Loop  (rows=2999) (actual rows=1452184 loops=1)
                     ->  Nested Loop  (rows=2999) (actual rows=1452184 loops=1)
                           ->  Hash Join  (rows=2999) (actual rows=1452184 loops=1)
                                 Hash Cond: (lineitem.l_suppkey = supplier.s_suppkey)
                                 ->  Index Smooth Scan using idx1202121036090 on lineitem
                                        (rows=299930) (actual rows=18230325 loops=1)
                                       Index Cond: ((l_shipdate >= '1995-01-01'::date)
                                                AND (l_shipdate <= '1996-12-31'::date))
                                 ->  Hash  (rows=1000) (actual rows=7969 loops=1)
                                       Buckets: 1024  Batches: 1  Memory Usage: 483kB
                                       ->  Nested Loop  (rows=1000) (actual rows=7969 loops=1)
```

```
                                      -> Index Smooth Scan using sql120209154306430
                                      on nation n1 (rows=2)(actual rows=2 loops=1)
                                            Filter: ((n_name = 'EGYPT'::bpchar)
                                            OR (n_name = 'CHINA'::bpchar))
                                            Rows Removed by Filter: 23
                                      -> Index Only Scan using idx1202121034380 on supplier
                                         (rows=500)(actual rows=3984 loops=2)
                                            Index Cond: (s_nationkey = n1.n_nationkey)
                                            Heap Fetches: 7969
                        -> Index Smooth Scan using sql120209155202560 on orders
                             (rows=1) (actual rows=1 loops=1452184)
                              Index Cond: (o_orderkey = lineitem.l_orderkey)
                    -> Index Only Scan using idx1202121037110 on customer
                       (rows=1) (actual rows=1 loops=1452184)
                         Index Cond: (c_custkey = orders.o_custkey)
                         Heap Fetches: 1452184
                -> Materialize (rows=2) (actual rows=2 loops=1452184)
                    -> Index Smooth Scan using sql120209154306430 on nation n2 (rows=2) (actual rows=2 loops=1)
                         Filter: ((n_name = 'CHINA'::bpchar) OR (n_name = 'EGYPT'::bpchar))
                         Rows Removed by Filter: 23
```

A.5 Q14

PostgreSQL:

```
Aggregate (rows=1) (actual rows=1 loops=1)
  -> Hash Join (rows=299930) (actual rows=747437 loops=1)
       Hash Cond: (lineitem.l_partkey = part.p_partkey)
       -> Index Scan using idx1202121036090 on lineitem (rows=299930) (actual rows=747437 loops=1)
            Index Cond: ((l_shipdate >= '1996-04-01'::date) AND (l_shipdate <'1996-05-01'::timestamp))
       -> Hash (rows=2000000) (actual rows=2000000 loops=1)
         Buckets: 262144 Batches: 1 Memory Usage: 125000kB
            -> Seq Scan on part (rows=2000000) (actual rows=2000000 loops=1)
```

PostgreSQL with Smooth Scan:

```
Aggregate (rows=1) (actual rows=1 loops=1)
  -> Hash Join (rows=299930) (actual rows=747437 loops=1)
       Hash Cond: (lineitem.l_partkey = part.p_partkey)
       -> Index Smooth Scan using idx1202121036090 on lineitem (rows=299930) (actual rows=747437 loops=1)
            Index Cond:((l_shipdate >= '1996-04-01'::date) AND (l_shipdate <'1996-05-01'::timestamp))
       -> Hash (rows=2000000) (actual rows=2000000 loops=1)
         Buckets: 262144 Batches: 1 Memory Usage: 125000kB
            -> Index Smooth Scan using sql120209154306520 on part (rows=2000000)(actual rows=2000000 loops=1)
```