The VLDB Journal (2018) 27:521–545
https://doi.org/10.1007/s00778-018-0507-8

REGULAR PAPER

Smooth Scan: robust access path selection without cardinality estimation

Renata Borovica-Gajic[1] · Stratos Idreos[2] · Anastasia Ailamaki[3] · Marcin Zukowski[4] · Campbell Fraser[5]

Received: 3 July 2017 / Revised: 11 May 2018 / Accepted: 15 May 2018 / Published online: 29 May 2018

Abstract

Query optimizers depend heavily on statistics representing column distributions to create good query plans. In many cases, though, statistics are outdated or nonexistent, and the process of refreshing statistics is very expensive, especially for ad hoc workloads on ever bigger data.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 core of the problem is the fixed decision on the type of physical operators that comprise a query plan. This paper makes a case for continuous adaptation and morphing of physical operators throughout their lifetime, by adjusting their behavior in accordance with the observed statistical properties of the data at run time. 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 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 optimization decisions on the access paths up front. Additionally, by depending only on the result distribution and eschewing statistics and cardinality estimates altogether, Smooth Scan ensures repeatable execution across multiple query invocations. Smooth Scan implemented in PostgreSQL demonstrates robust, near-optimal performance on micro-benchmarks and real-life workloads, while being statistics oblivious at the same time.

Keywords Access path selection · Cardinality estimation · Robust query execution · Adaptive query processing · DBMS

1 Introduction

Perils of query optimization complexity Query execution performance of database systems heavily depends 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 [54]. To find the best possible plan, query optimizers employ a cost model to estimate performance of viable alternatives. In turn, cost models rely on statistics about the data to estimate the size of intermediate results (cardinality estimates) of each operator in the plan. With the growth in complexity of decision support systems (e.g., templatized queries, UDF) and the advent of dynamic web applications that have databases at their backbone, the optimizer's grasp of reality becomes increasingly loose and it becomes more difficult to produce an optimal plan [17,44]. For instance, to defy complexity and make up for lack of statistics, commercial

Electronic supplementary material The online version of this article (https://doi.org/10.1007/s00778-018-0507-8) contains supplementary material, which is available to authorized users.

✉ Renata Borovica-Gajic
renata.borovica@unimelb.edu.au

Stratos Idreos
stratos@seas.harvard.edu

Anastasia Ailamaki
anastasia.ailamaki@epfl.ch

Marcin Zukowski
marcin.zukowski@gmail.com

Campbell Fraser
campbellbrycefraser@gmail.com

[1] School of Computing and Information Systems, The University of Melbourne, Melbourne, VIC, Australia

[2] School of Engineering and Applied Sciences, Harvard University, Cambridge, MA, USA

[3] School of Computer and Communication Sciences, Ecole Polytechnique Federale de Lausanne and RAW Labs, Lausanne, VD, Switzerland

[4] Snowflake Computing, San Mateo, CA, USA

[5] Google Inc., Seattle, WA, USA

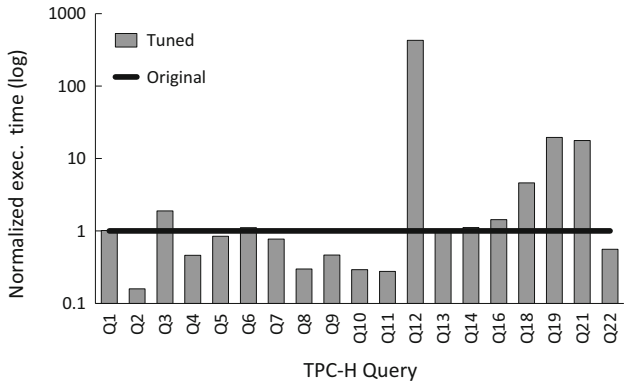

Fig. 1 Non-robust performance due to cardinality misestimates in a state-of-the-art commercial DBMS

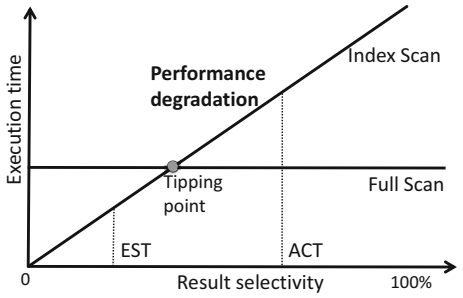

Fig. 2 Access path selection sensitivity to cardinality estimation

database management systems often assume uniform data distributions and attribute value independence, which is in reality hardly the case [29]. As a result, cardinality estimates are frequently off by several orders of magnitude, consequently leading to suboptimal plans and non-robust query performance [9,35,37,59,65,75].

Example of non-robust behavior To illustrate the severe impact of cardinality misestimates and consequent suboptimal access path choices, we provide an example of non-robust performance using a state-of-the-art commercial system, referred to as DBMS-X. We run the TPC-H benchmark [76], having first tuned the system with a set of indexes as indicated by its own tuning tool. Figure 1 shows that for several queries performance degrades significantly after tuning (e.g., up to a factor of 400 for Q12). More details are provided in Sect. 7.2, but for now it suffices to say that the only change compared to the original plan of Q12 is the type of access path operator. This decision prolonged the execution time from 1 min to 11 h.

Tipping points causing robustness problems The performance degradation shown in Fig. 1 is attributed to suboptimal access path choices, where the optimizer favored *index* usage over full table scans for the cases when it *underestimated* the result cardinality sizes. The core of the problem of access path selection lies in the fact that even a small cardinality estimation error may lead to a drastically different result in terms of performance. This effect is shown in Fig. 2 as the *tipping point*. When considering access path selection, the optimizer makes a choice between an index scan and full scan. Figure 2 illustrates how the cost (i.e., execution time) varies for these access path alternatives as a function of result selectivity increase. The tipping point is the estimated cardinality value[1] for which the optimizer makes a decision switch, i.e., for the values below the tipping point the optimizer opts for the index scan and for the values above it opts for the full scan. This means that one tuple difference in cardinality estimation can *swing the decision* between an index scan and a full scan, possibly causing a significant performance drop. It also demonstrates the sensitivity of the optimizer to the quality of estimates. For instance, the choice of index for the estimated selectivity point shown as EST in Fig. 2 will result in a severe performance degradation for the case when the actual selectivity appears to be higher than estimated (e.g., ACT shown in Fig. 2).

A case for robustness in query processing Overall, the sensitivity to the quality of the optimizer's cardinality estimation results in *unpredictable performance*, thereby affecting the *robustness* of the system. In addition, the overall behavior is driven by the *current version* of statistics used by the system, which means that two different deployments over the same data might have different performance results if their statistical summaries that represent data distributions differ. Statistical summaries form a part of metadata catalog populated by the collect statistics command; hence, it is possible for different deployments to be out of sync: although representing the same data(base) their statistical summaries will differ. The last aggravates testing *repeatability* across different servers or even multiple invocations of the same query (if the statistics collection command was issued in between).

Stability and *predictability*, which imply that similar query inputs should have similar execution performance, are of paramount importance for industrial vendors as a path toward respecting service-level agreements (SLA) [64]. 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, and where a manual human effort is highly undesirable [32]. In these cases, the system's ability to efficiently operate in the face of unexpected and especially adverse runtime conditions (e.g., receiving more tuples from an operator than estimated) becomes more important than yielding great performance for one query input while potentially suffering from severe degradation due to a suboptimal plan choice for another [43,45]. We define *robustness in query processing as the ability of a system to efficiently cope with unexpected*

[1] We use the term cardinality value as the value derived from the optimizer's estimate on result selectivity, i.e., $card\_value = |T| * selectivity$, where $|T|$ is the number of tuples in a relation and selectivity is a selectivity factor [73].

and adverse conditions with respect to its input and deliver near-optimal performance for all query inputs.

Run-time reoptimization Past efforts on robustness focus primarily on dealing with the problem at the optimizer level [7,9,33,34,37,38]. 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 reoptimization [3,6,40,58,59,65], although promising, lack the flexibility at the level of access paths. They are limited in their scope, either by completely ignoring intra-operator adaptivity within the access path operator, or by performing binary switching decisions that introduce risks and result in unpredictable performance.

To illustrate the latter, let us re-consider the access path selection problem. A simple solution to *recover* from the suboptimal access path choice would be to switch the strategy (at run time) from an index scan to a full table scan upon detecting a cardinality misestimation or alternatively when the observed cardinality reaches the tipping point between the index and full table scan. Such a case is depicted in Fig. 3 with a 'Reoptimization' line. Reoptimization is typically performed by monitoring the result cardinality and triggering the switch once the observed cardinality exceeds the estimate. Reoptimization bounds the worst case performance and prevents severe performance degradation that could have happened with continuation of the suboptimal access path (the index scan). However, it is not robust. The main problem with reoptimization is that it is based on a binary decision

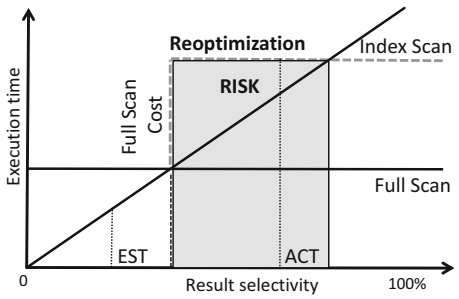

Fig. 3 Reoptimization benefits and risk

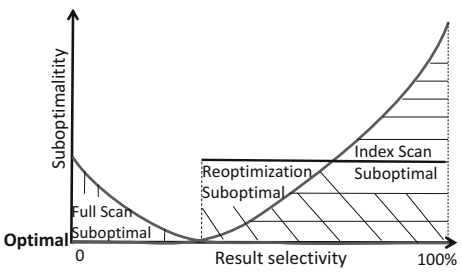

Fig. 4 Access paths suboptimality as a function of selectivity increase

and switches completely when a certain cardinality threshold is reached. This means that even a single extra result tuple can result in drastically different performance if the switch occurs, since after the switch the execution time is prolonged by the time to perform full scan (see Fig. 3). Such a behavior can discourage business analysts who repeat the same query they ran yesterday and observe different query performance, while the only change was addition of a single record to a database table [10].

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 change will be amortized over the remaining query time, render this approach volatile and hence non-robust. For instance, if the actual result selectivity lies anywhere in the gray box shown as 'Risk' in Fig. 3, a better decision would be to continue with the index scan, since the reoptimization overhead (the cost of full scan) cannot be amortized over the rest of the query lifetime. Additionally, since the violation of the optimizer's estimates usually triggers reoptimization, this approach remains sensitive to the version of statistics present in the system, which complicates testing across different query invocations and deployments.

Suboptimality of access paths Figure 4 illustrates the core of the problem with access path selection. The figure depicts how suboptimality changes for access paths as a function of result selectivity increases. The suboptimality is measured as a discrepancy from the optimal solution which lies at the lower bound of the alternative access paths throughout the entire selectivity interval. The actual selectivity points and the suboptimality levels will vary depending on the hardware characteristics. Neither access path is, however, optimal over the entire selectivity interval, making any choice potentially risky for the cases of cardinality misestimation. The full scan is suboptimal until the tipping point, since the index scan is the optimal access path for low selectivity. On the contrary, the index scan is highly suboptimal for high selectivity. Reoptimization is never optimal, but is closer to the optimal compared to the index (in the worst case), making it a viable *patch* to prevent further performance degradation in the case of cardinality underestimation. To reduce variability and performance drops due to suboptimal decisions, we need an access path whose performance stays at the lower cost boundary (i.e., close to the 'Optimal' line), throughout the entire selectivity interval.

Smooth Scan In this paper, we respond to the quest for robust execution at the access path level by introducing a novel class of access path operators designed with the goal of providing robust performance for every query 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, we propose a new class of *morphable operators* that *con-*

tinuously 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.* 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.

This paper extends our previous work [18] with the theoretical analysis on the worst case performance guarantees of Smooth Scan. Worst case performance guarantees are extremely important when considering the algorithm robustness as they show the maximal discrepancy from the optimal solution. In addition, since Smooth Scan trades off CPU for I/O reduction, we present a more detailed cost model that incorporates both I/O and CPU costs. We also present a new robust out-of-core design and implementation of the Smooth Scan algorithm that achieves good performance irrespective of the allowed memory size. Additionally, we enrich our experiments across several dimensions: (i) we report new experiments with respect to the cost model and (ii) statistics collection and include (iii) an in-depth sensitivity analysis on the behavior of the Smooth Scan algorithm on disk and (vi) for various memory sizes. We finally survey related work on adaptive query processing techniques in more detail.

The contributions of the paper are the following:

- We propose a new paradigm of building smooth and morphable access path 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 access path operator called Smooth Scan that morphs between an index access and a full scan as selectivity knowledge evolves at run time.
- We present a theoretical analysis on the worst case performance guarantees of Smooth Scan's alternative policies.
- Using both synthetic benchmarks and TPC-H in a thorough experimental analysis, we show that Smooth Scan, implemented in PostgreSQL, is a viable option for achieving near-optimal performance, by approximating the performance of the optimal access path throughout the entire selectivity interval.

The rest of the paper is structured as follows: Section 2 presents the background on access path selection, describing three main approaches. Section 3 introduces the intra-operator adaptivity at the access path level through the design of the Smooth Scan operator. Section 4 introduces the implementation details of Smooth Scan when incorporated into a mature open-source DBMS (PostgreSQL). Section 5 presents the detailed cost model of Smooth Scan. Section 6 presents a competitive analysis of the worst case performance of Smooth Scan when compared against a theoretical bound. Section 7 demonstrates, through an experimental analysis, that Smooth Scan achieves robust and efficient performance. Finally, Sect. 8 provides a related work discussion, positioning Smooth Scan with respect to existing efforts in (re)optimization, adaptivity and robustness. Section 9 presents our concluding remarks.

2 Background

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

Full (table) scan 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 5a 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 the page 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. Despite its rigorousness, 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.

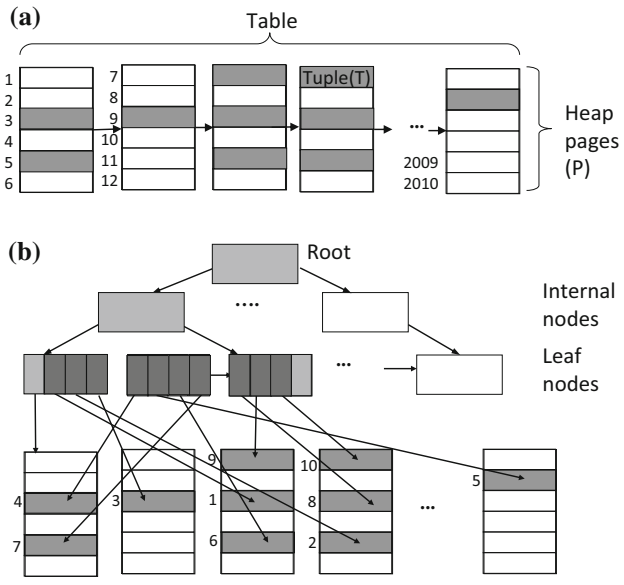

Fig. 5 Access paths in a DBMS. **a** Full (table) scan. **b** index scan

Index scan Secondary indexes are built on top of data pages stored in the heap. Indexes are usually implemented as B^+-trees containing pointers to tuples. Figure 5b depicts a B^+-tree index built on top of the same table we used in Fig. 5a with the leaves of the tree pointing 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. As before, the number placed on the left-hand side of each tuple indicates the order in which it is accessed. 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 deteriorates quickly if many tuples need to be selected. Moreover, as the number of tuples that qualify grows, so does the chance that the index scan visits the same page more than once.

Sort (bitmap) scan represents a middle ground between the previous two approaches. Sort Scan still exploits the secondary index to obtain tuple identifiers (ID) 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 translated into a (nearly) sequential pattern, which is easily detected by disk prefetchers. This can decrease execution time even when the selectivity of the operator grows significantly. However, it 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* [73]. One advantage of B-tree indexes stems 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 is restored by introducing a sorting operator above the index access (or up in the tree). In addition, the introduction of the blocking operator so early in the execution plan may stall the rest of the operators; if they require a sorted input, their execution can start only after the second sort finishes.

3 Intra-operator adaptivity with Smooth Scan

Having discussed in Sect. 1 reasons why performance cliffs are undesirable, this section introduces *Smooth Scan* which avoids performance cliffs while providing robust query execution performance within given cost boundaries. Instead of making binary decisions like the one introduced with reoptimization, Smooth Scan gradually and adaptively shifts its behavior between access path patterns to fit the data distributions, thereby avoiding performance drops.

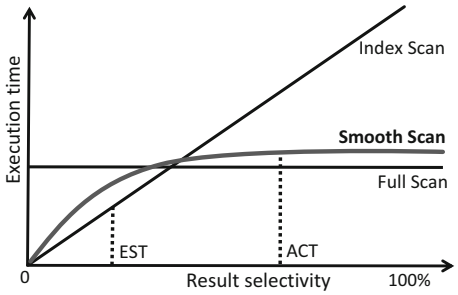

Fig. 6 Targeted behavior of Smooth Scan

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

Figure 6 shows the targeted performance behavior of Smooth Scan as a function of result selectivity increase. As we produce more result tuples, the behavior of Smooth Scan keeps adjusting continuously, eventually approaching the behavior of the full scan when more tuples qualify from the select operator. This continuous adaptation is the key element, which provides near-optimal performance regardless of the severity of cardinality estimation errors.

A critical advantage of Smooth Scan over run-time reoptimization with binary switching is that Smooth Scan does not need to choose a single point of adaptation (i.e., the switch point). As a result, a single point of failure is removed and replaced with incremental refinement actions and decisions. Moreover, we release the optimizer from the burden of choosing an optimal access path a priori, which solves both common problems with access paths: (a) choosing an index when selectivity is underestimated due to attribute correlation, which usually results in performance degradation; (b) choosing a full table scan when selectivity is overestimated due to anti-correlation (see Fig. 4).

3.1 Morphing mechanism

We now describe how Smooth Scan achieves this gradual adaptation. During the operator lifetime, 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 0: Index scan Assuming the existence of index as an access path, Smooth Scan starts with a classical Index Scan as its initial mode. For each tuple from the index, Smooth Scan fetches a single page from the main relation where the look-up key resides and produces one resulting tuple. Additionally, Smooth Scan continuously monitors the result cardinality, and once it exceeds a threshold (see discussion on the policies below), it switches to the Entire Page Probe mode.

Mode 1: Entire page probe A core problem of Index Scan is that a particular disk page can be referenced multiple times, causing repeated (random) I/O accesses (e.g., 3 times for pages 3 and 4 in Fig. 5b). The classical index scan retrieves solely the searched record driven by the index probe while ignoring remaining records from the page, some possibly belonging to the result. The latter potentially results in a need to return to the same page somewhere in the future if more tuples from the same page qualify. 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 [41], Smooth Scan invests CPU cycles for reading additional tuples from each page with minimal overhead. Figure 7 depicts the access pattern of a Smooth Scan in this mode. Like in Fig. 5, 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 Fig. 7 as the gray rectangle over the heap pages.

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 detects result cardinality 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 in 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, Smooth Scan keeps expanding, and conceptually, it morphs more aggressively into a full table scan.

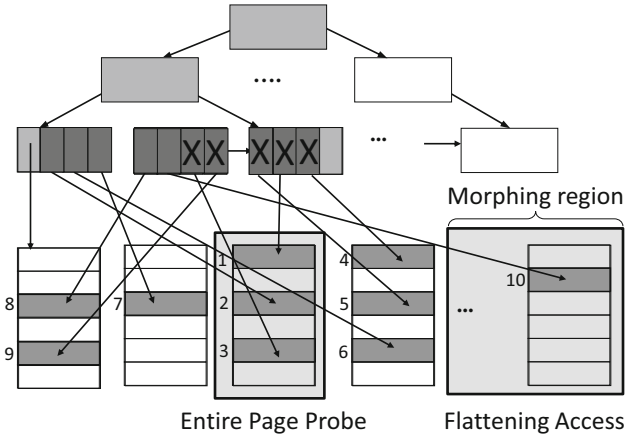

Fig. 7 Smooth Scan access pattern

3.2 Morphing policies

There are several ways in which Smooth Scan can morph between modes.

Greedy policy Assuming a worst case scenario, i.e., a very high result selectivity, Smooth Scan can perform morphing region 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 full scan. The downside is that, in the case of low selectivity, Smooth Scan 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 size when it detects a selectivity increase. In particular, Smooth Scan computes the result selectivity over the last morphing region (the heap pages triggered with the previous index access) and it increases the morphing region size each time the local selectivity over the last morphing region [calculated in Eq. (1)] is greater than the global selectivity over so far seen pages [calculated in Eq. (2)]. The meaning of the parameters can be found in Table 1. If selectivity does not increase, Smooth Scan keeps the previous morphing region size.

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

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

Elastic policy When considering big data sets, it is unlikely that a single execution strategy will be optimal during the entire scan over a big table; dense and sparse regions with respect to the tuple distribution 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 region size over a dense region, while it decreases the morphing region size when it passes the dense 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 a denser region; hence, Smooth Scan doubles the morphing size. In the counter case, Smooth Scan halves the morphing region size for the next heap access. This way, morphing is performed at a pace that is purely driven by the data and the query at hand.

3.3 Morphing triggering point

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 Smooth Scan initiates morphing once the 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 Sect. 3.2.

SLA driven Another option is to take action only when there is 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 Sect. 5). The moment Smooth Scan detects that it will not be able to guarantee the SLA target behavior unless it switches to a more conservative behavior, it triggers morphing.

Eager approach An alternative approach, favored in this work, is to completely replace access paths with Smooth Scan. With this strategy, Smooth Scan eagerly starts morphing immediately as of the first tuple. In this way, Smooth Scan guarantees 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.

Since the bookkeeping overhead of the Eager strategy is minimized, in the experiments performed throughout this paper, Eager is the default strategy unless stated otherwise. We study other strategies in detail in Sect. 7.

4 Integration of Smooth Scan into PostgreSQL

In this section, we discuss the design details of Smooth Scan and its interaction with the remaining query processing stack when incorporated inside an existing mature DBMS. We implement Smooth Scan in PostgreSQL 9.2.1 DBMS as a classical access path operator existing side by side with the traditional access path operators described in Sect. 2. During query execution, the access path choice is replaced by the choice of Smooth Scan, whereas the upper layers of query plans generated by the optimizer remain intact. Thus, one advantage of Smooth Scan is that it can be integrated into existing systems with fewer changes compared to more involved approaches such as [5,6].

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 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 accesses the heap page only if that page has not been accessed before. Otherwise, Smooth Scan skips the leaf pointer (X in Fig. 7) and continues the leaf traversal.

Tuple ID cache If Smooth Scan starts from Mode 0 following the Optimizer or SLA Driven strategy, it has 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 the same page is fetched with Smooth Scan. To address this issue, Smooth Scan keeps a cache of tuple IDs produced with the traditional access in a bitmap-like structure. Later, while producing tuples Smooth Scan performs a bit check whether the tuple has already been produced. The overhead of the Tuple ID Cache, while relatively low, can be avoided if a DBMS maintains a strict $(index_{key}, TID)$ ordering in the secondary index (which some commercial systems do). Then it suffices to remember the last tuple reached with the traditional index and ignore tuples with $(index_{key}, TID)$ lower than that last tuple.

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 constraint, 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. The Result cache is partitioned into a number of smaller caches that can be deleted once all tuples from an instance are produced. By grouping the caches per key value (or key ranges), Smooth Scan can remove all items from a cache as soon as the key value of the cache is traversed.

Memory management Both the Page ID and Tuple ID Cache are bitmap structures, meaning that their size is significantly smaller compared to the data set size (they easily fit into memory). To illustrate, their size is usually a couple of KB to MB for hundreds of GB of data. In the Tuple ID cache, we keep only the IDs of the tuples acquired 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. In the worst case, if the cache grows above the allowed memory size, Smooth Scan performs partitioning and overflows partitions into temporary files on disk. Partition ranges are created by looking at the root page of the index to decide on the number of partitions (and their ranges). The reasons are twofold. First, the root page of an index is typically stored in memory; hence, its access will not invoke an unnecessary I/O. Second, the root page of an index contains information about the distribution of key values, since, assuming a B-tree is balanced, data skew will be shown in the way the keys are distributed. For instance, a big gap between two consecutive keys in the root implies a sparse region with respect to the distribution of data with values in that range. Similarly, a small gap implies a dense region. Smooth Scan uses the root keys to decide on the number of partitions and their ranges given the allotted memory size (and the table size). If the number of partitions is higher than the total number of keys in the root page, meaning that consecutive keys create too large partitions, Smooth Scan accesses the second-level index pages to refine the partition ranges.

During run time, Smooth Scan pipelines tuples for the current key immediately as it finds them, while remaining qualifying tuples (for other partition ranges) are stored in their corresponding partitions. If memory becomes scarce, Smooth Scan employs overflow resolution and writes a partition with the highest key values into a temporary file on disk first. Once a particular key (or a partition range) is completely consumed, Smooth Scan can freely discard the partition it belongs to. This shrinking reduces memory pressure during run time. Once Smooth Scan needs to service the data belonging to another partition, it simply accesses the partition (i.e., the temporary file) and returns all tuples belonging to it, enjoying the benefit of spatial locality. Hence, even if the table is much larger compared to the allotted memory, Smooth Scan will still benefit from reducing repeated and random accesses compared to the index scan at the expense of additional sequential access to temporary files.

4.2 Interaction with query processing stack

Smooth Scan is an access path targeted primarily at preventing severe performance degradation due to unexpected selectivity increase, which is a common complaint received by the customer support for major industrial vendors as it is a major source of unpredictability in query performance [43,44,63]. Nonetheless, its impact goes much beyond being just a patch that prevents further performance drops.

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 Smooth Scan. Smooth Scan will then make all decisions on-the-fly during query execution. It should be noted, however, that Smooth Scan is a local optimization that solves the problem of access path selection, while the problem of join ordering still needs to be addressed by the query optimizer.

Interaction with other operators Smooth Scan is able to completely replace the functionality of both index scan and full scan. The output of Smooth Scan is an input to other operators in a query plan. Depending on the next operator in the query tree, a different variation of Smooth Scan may be used. For example, if a Merge Join follows Smooth Scan implying an imposed order among tuples, then the variant of Smooth Scan with the result caching will be used. Since the tuples obtained out of order could not be immediately consumed, they are rather stored in the Result Cache until their key value arrives. If Index Nested Loops Join (INLJ) is an operator on top of the scan and Smooth Scan is employed as the outer input to a join, Smooth Scan does not have constraints on the order of tuples produced from this input; hence, Smooth Scan can consume tuples the moment it finds them and no caching is needed. If the ordering requirement is, however, placed by some of the operators up in the tree, we still employ the first option. If Smooth Scan serves as an inner input (a parameterized path) to a join, the results per requested join key could be produced in an arbitrary order by calling Smooth Scan with that particular key value as a filtering predicate. As a result, the repeated I/O accesses are avoided and random ones are significantly reduced per join key value, which helps in the case of multiple key matches (e.g., PK–FK relationship). Finally, since Hash Join (HJ) does not support an interesting order, this implies that when placed below a HJ,

Smooth Scan can produce the result tuples the moment it finds them.

5 Modeling Smooth Scan

This section provides an analytical model of the access path alternatives. The analytical model serves the purpose of answering the critical questions of which policy and mode Smooth Scan should employ and when. Smooth Scan trades CPU for I/O cost reduction; thus, the proposed model includes the cost of the access path operators both in terms of the number of disk I/O accesses and in terms of the CPU cost. Although it is expected that in most cases I/O dominates overall cost [41], the rapid evolution of modern hardware continuously shifts those balances; thus, we provide a complete model that can be easily adjusted for future faster hardware as well. We make a distinction between the cost of a sequential and random access, since the nature of accesses drives the overall query performance.

$$\#T_{\mathrm{P}} = \left\lfloor \frac{P_{\mathrm{S}}}{T_{\mathrm{S}}} \right\rfloor \tag{3}$$

$$\#P = \left\lceil \frac{\#T}{\#T_{\mathrm{P}}} \right\rceil \tag{4}$$

$$fanout = \left\lfloor \frac{P_{\mathrm{S}}}{1.2 \times K_{\mathrm{S}}} \right\rfloor \tag{5}$$

$$\#leaves = \left\lceil \frac{\#T}{fanout} \right\rceil \tag{6}$$

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

$$card = sel \times \#T \tag{8}$$

$$\#leaves_{res} = \left\lceil \frac{card}{fanout} \right\rceil \tag{9}$$

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 textbooks [69].) Indexes are implemented as B$^+$-trees, with k as the tree fanout. Equations (3)–(9) are base formulas used for all access path operators. 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 Eqs. (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 query its cost remains constant. As shown in Eq. (10), the I/O cost is the cost of fetching all pages of the relation sequentially (as we expect each table to be stored sequentially on disk). Once full scan fetches a page, it performs a tuple comparison for all tuples from the page to find the ones that qualify. Assuming that each comparison invokes one CPU operation, the overall CPU cost is given by Eq. (11).

Table 1 Smooth Scan: cost model parameters

Parameter	Description
T_{S}	Tuple size (bytes)
$\#T$	Number of tuples in the relation
P_{S}	Page size (bytes)
$\#T_{\mathrm{P}}$	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
$m0_{chk}$	Was a traditional index employed first? 0/1
$rand_{cost}$	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
$\#P_{res\_reg}$	Number of pages with result in current region
$\#P_{seen}$	Number of pages seen so far
$\#P_{seen\_reg}$	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

$$FS_{io\_cost} = \#P \times seq_{cost} \tag{10}$$

$$FS_{cpu\_cost} = \#T \times cpu_{cost} \tag{11}$$

Index scan To fetch the tuples, the (non-clustered) index scan traverses the tree once to find the first tuple that qualifies [$height$ in Eq. (12)]. For the remaining tuples, it continues traversing the leaf pages from the index ($\#leaves_{res} \times seq_{cost}$) and uses all tuple IDs that qualify to access the heap pages, potentially triggering a random I/O operation per look-up [$card$ in Eq. (12)]. While traversing the tree, within every internal node page, the index scan performs a

binary search in order to find a pointer of interest to the next level [$height \times \log_2(fanout)$ in Eq. (13)]. For each tuple obtained by following the pointers from the leaf, it then performs a tuple comparison to see whether the tuple qualifies [the second part of Eq. (13)].

$$IS_{io\_cost} = (height + card) \times rand_{cost} + \#leaves_{res} \times seq_{cost} \quad (12)$$

$$IS_{cpu\_cost} = (height \times \log_2(fanout) + card) \times cpu_{cost} \quad (13)$$

Smooth Scan Having defined the cost of the basic access path operators, we move on to define the cost of Smooth Scan. We calculate the cost of Smooth Scan for each mode separately. Overall result cardinality is split between the modes (Eq. (14)). Like the index scan, the cost of the Smooth Scan access is driven by selectivity. Assuming uniform distribution of the result tuples (the worst case cost), the number of pages containing the result is calculated in Eq. (15).

$$card = card_{m0} + card_{m1} + card_{m2} \quad (14)$$

$$\#P_{res} = min(card, \#P) \quad (15)$$

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 in Mode 0 [the multiplier 2 in Eq. (16)], to populate tuple IDs to the Tuple ID cache.

$$SS_{cpu\_cost\_m0} = (height \times \log_2(fanout) + card_{m0} \times 2) \times cpu_{cost} \quad (16)$$

Mode 1: Entire page probe The number of tuples for which Mode 1 is going to be employed is calculated in Eq. (17). Every page is assumed to be fetched with a random access (Eq. (18)). Once Smooth Scan obtains a page, it performs a tuple comparison checking all tuples from the page [the first part of Eq. (19)]. Before fetching a page x, Smooth Scan checks whether x has already been processed; if not, it scans x and adds it to the Page Cache [the second part of Eq. (19)]. Finally, if Smooth Scan started in Mode 0, for each tuple Smooth Scan has to perform a check whether the tuple has already been produced in Mode 0 [the third part of Eq. (19)]. In case Smooth Scan needs to support an interesting order, the Result Cache will be used as a replacement for the Tuple ID cache functionality. In that case, Smooth Scan only marks 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.

$$\#P_{m1} = \min(card_{m1}, \#P) \quad (17)$$

$$SS_{io\_cost\_m1} = \#P_{m1} \times rand_{cost} \quad (18)$$

$$SS_{cpu\_cost\_m1} = (\#P_{m1} \times \#T_P + \#P_{m1} \times 2 + \#P_{m1} \times \#T_P \times m0_{chk}) \times cpu_{cost} \quad (19)$$

Mode 2: Flattening access We calculate the maximum number of pages to fetch with Mode 2 in Eq. (20). 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. (21). The solution of the recurrence equation is shown in Eq. (22). In this case, n is the number of times Smooth Scan expands the morphing region size (i.e., the number of times Smooth Scan performs a random I/O access) and $f(n)$ translates to the number of pages to fetch with Mode 2 ($\#P_{m2}$). The minimum number of random accesses (jumps) to fetch all pages containing the results is given by Eq. (24). 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. (25). When selectivity is low, the number of random I/O accesses is at worst 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 after this number all pages will be accessed.

Since both Eqs. (24) and (25) 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. (26), 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. (27)).

$$\#P_{m2} = \min(card_{m2}, \#P - \#P_{m1}) \quad (20)$$

$$f(i+1) = 2 \times f(i), i = 0..n \quad (21)$$

$$f(0) = 0, f(n) = 2^n, n >= 0 \quad (22)$$

$$\#P_{m2} = \sum_{i=0}^{\#rand_{io}(m2\_min)} 2^i \quad (23)$$

$$\#rand_{io}(m2\_mn) = \log_2(\#P_{m2} + 1) \quad (24)$$

$$\#rand_{io}(m2\_mx) = \min\left(\#P_{m2}, \log_2(\#P + 1)\right) \quad (25)$$

$$SS_{io\_cost\_m2} = \#rand_{io}(m2) \times rand_{cost} + (\#P_{m2} - \#rand_{io}(m2)) \times seq_{cost} \quad (26)$$

$$SS_{cpu\_cost\_m2} = (\#P_{m2} \times \#T_P + \#P_{m2} \times 2 + \#P_{m2} \times \#T_P * m0_{chk}) \times cpu_{cost} \quad (27)$$

Finally, the overall costs are the sums of the operator CPU and I/O costs for all employed modes.

$$SS_{io\_cost} = SS_{io\_cost\_m0} + SS_{io\_cost\_m1} + SS_{io\_cost\_m2}$$
$$SS_{cpu\_cost} = SS_{cpu\_cost\_m0} + SS_{cpu\_cost\_m1} + SS_{cpu\_cost\_m2}$$

6 Competitive analysis

This section provides a competitive analysis comparing Smooth Scan against the optimal access path (referred to as Oracle). The Oracle provides a theoretical bound, modeling the case when all resulting pages are known in advance and accessed sequentially; it mimics the behavior of Sort Scan, while ignoring the sorting cost. We calculate a *competitive ratio* (CR) as the maximum ratio between the cost of Smooth Scan and the Oracle throughout the entire selectivity interval (Eq. (28)).

$$\text{CR} = \max\left(\frac{SS_{cost}(sel)}{Oracle(sel)}\right), \quad sel \in [0, 100\%] \tag{28}$$

The competitive ratio is an important metric when considering robustness, since it shows how far from optimal Smooth Scan can be. The purpose of this analysis is to give insights on which Smooth Scan's policy is most robust, and examines worst case performance guarantees of Smooth Scan. For each of the policies, we consider their worst case result distribution (depicted in Fig. 8) and calculate the CR as a function of table size (the number of pages $\#P$).

6.1 Greedy policy

We first consider Smooth Scan with the Greedy Policy according to which the operator increases the morphing region size after each index access.

Worst case result distribution The worst case result distribution for the Greedy policy is when all additional pages that Smooth Scan obtains with the flattening access pattern do not contain any tuples contributing to the result set. In this case, the number of fault pages (not containing the result tuples) is maximized. This can happen when the next result tuple is always one page ahead of the current morphing region. The order does not have to be such that the page is strictly ahead, but without 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 8a depicts a result distribution for such a case. Squares with striped lines denote pages containing results, while empty squares denote fault pages (i.e., without results). Below each graphics in Fig. 8 describing a different 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 Fig. 8a). 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 Greedy Smooth Scan is when the cardinality is equal to the number of random jumps (Eq. (29)). Equation (30) shows the cost of Smooth Scan for this use case scenario, while Eq. (31) calculates the CR.

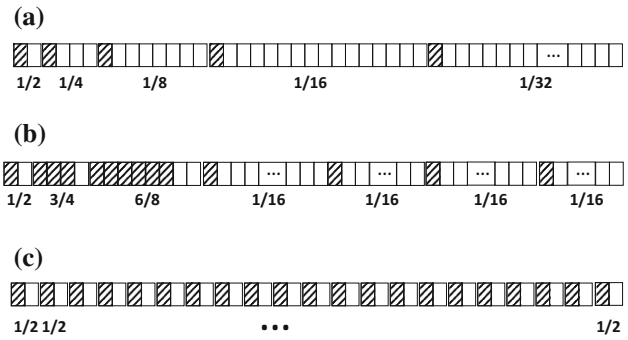

Fig. 8 Worst case result distributions for Smooth Scan alternatives. **a** Greedy Smooth Scan, **b** Selectivity increase Smooth Scan, **c** Elastic Smooth Scan

$$card = \#rand_{io} = \log_2(\#P + 1) \tag{29}$$
$$SS_{cost} = \#rand_{io} \times rand_{cost} + (\#P - \#rand_{io}) \times seq_{cost} \tag{30}$$
$$\text{CR} = \max\left(\frac{SS_{cost}}{Oracle}\right) = \frac{SS_{cost}}{\#rand_{io} \times seq_{cost}} = \frac{\#rand_{io} \times rand_{cost} + \#P \times seq_{cost}}{\#rand_{io} \times seq_{cost}} - 1 \tag{31}$$

The CR of Greedy Smooth Scan against the Oracle for this use case and characteristics of HDD ($rand_{cost} = 10$ and $seq_{cost} = 1$) is depicted in Fig. 9. For 1000 pages the value of CR is equal to 110. The value of CR increases sublinearly with the increase in number of pages, reaching the value of 760 for 10,000 pages. A similar CR is observed for the characteristics of SSD ($rand_{cost} = 2$ and $seq_{cost} = 1$), ranging from 100 to 750.

Discussion From the competitive analysis, we see that Greedy Smooth Scan is not a viable option for low selectivity, since it can be highly suboptimal due to a high number of fault pages that this policy might fetch.

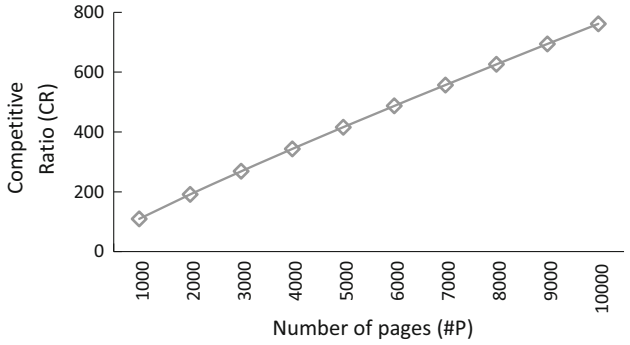

Fig. 9 The CR of Greedy Smooth Scan against Oracle

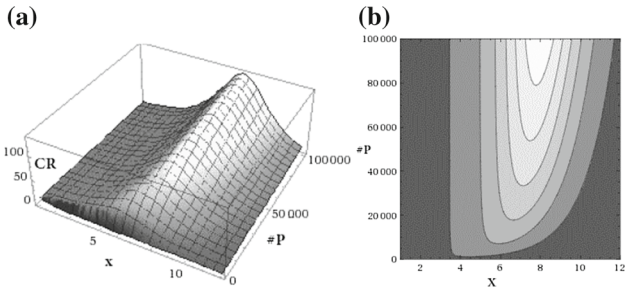

Fig. 10 The CR of Selectivity Increase Smooth Scan when compared against Oracle. **a** CR against Oracle, **b** equidistant contours

6.2 Selectivity increase driven policy

Selectivity increase (SI) Driven Policy increases the morphing region size as a response to the observed selectivity increase.

Worst case result distribution Figure 8b depicts the worst case result distribution for this policy. With the SI driven policy, an initial high selectivity can mislead Smooth Scan to keep a high region size (e.g., in Fig. 8b a morphing region of size 16 is kept throughout the rest of the operator lifetime).

To increase the morphing region size, SI Smooth Scan has to notice the selectivity increase over the last morphing region bigger than the selectivity seen so far [calculated in Eqs. (1) and (2)]. A minimal selectivity sequence that will trigger the morphing region 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. Equation (32) calculates the number of pages containing results needed to trigger such a behavior. After performing the morphing region expansion x times, to maximize the number of fault pages the remaining y morphing regions have a single match. The total number of accesses is shown in Eq. (33). In the following equations, we replace y with Eq. (34) [derived from Eq. (33)]. Since the total cost of Smooth Scan depends on both x and y, and since we can represent y as $f(\#P, x)$, in Fig. 10 we plot the CR of Smooth Scan against the Oracle as a function of x and $\#P$. In addition, Fig. 10b shows a 2D view of the same graph, where the same color in equidistant contours denotes the same value of CR. The peak value has the brightest color. We plot the CR for the HDD characteristics ($rand_{cost} = 10$ and $seq_{cost} = 1$).

$$\begin{aligned} \#P_{res} &= 1 + \sum_{i=0}^{x-2} 3 \times 2^i + \sum_{i=1}^{y} 1 \\ &= 1 + 3 * (2^{x-1} - 1) + y \end{aligned} \tag{32}$$

$$\begin{aligned} \#P &= \sum_{i=1}^{x} 2^i + 2^x * y \\ &= 2 * (2^x - 1) + 2^x * y = 2^x * (2 + y) - 2 \end{aligned} \tag{33}$$

$$y = \frac{\#P + 2}{2^x} - 2 \tag{34}$$

$$\#rand_{io} = x + y$$

$$\begin{aligned} SS_{cost} &= \#rand_{io} \times rand_{cost} \\ &\quad + (\#P - \#rand_{io}) \times seq_{cost} \end{aligned} \tag{35}$$

$$\text{CR} = \frac{SS_{cost}}{\#P_{res} \times seq_{cost}} \tag{36}$$

For this use case, the 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. This is expected since with more pages we can increase the morphing region size to a higher value, for which we need more steps. Nonetheless, the overall trend is similar. Although the CR of SI Smooth Scan is better than the CR of Greedy Smooth Scan, it is still a monotonically increasing sublinear function, which again puts a soft upper bound on the worst case performance of SI Smooth Scan. The same trend is noticed in the case of SSD as well.

Discussion Similar to the Greedy Policy, there are cases when the SI driven policy cannot provide robust performance. With a soft bound on the CR, the discrepancy of SI Smooth Scan from the optimal access path can be quite high, making it an undesirable choice.

6.3 Elastic policy

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

Highest page miss rate In order to increase the morphing region size, Smooth Scan has to notice the same selectivity increase pattern as the one described in Eq. (32). The behavior of Elastic Smooth Scan, however, differs in this case. After

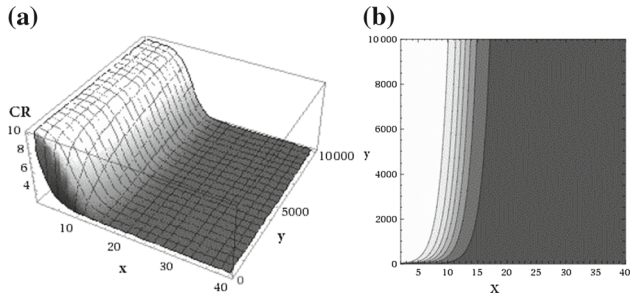

Fig. 11 The CR of Elastic Smooth Scan for the worst case result distribution of SI Smooth Scan. **a** CR against Oracle, **b** equidistant contours

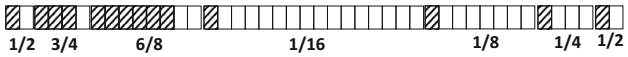

Fig. 12 The highest page miss rate for Elastic Smooth Scan

noticing the selectivity drop, Elastic Smooth Scan progressively decreases the morphing size back until it reaches 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 region size of 1 for the $(y-x)$ remaining tuples (assuming no local selectivity increase is noticed again). Equation (37) calculates the total number of pages accessed.

$$\begin{aligned} \#P &= \sum_{i=1}^{x} 2^i + \sum_{i=0}^{x} 2^i + (y-x) \\ &= 2*(2^x - 1) + 2^{x+1} - 1 + y - x \\ &= 2^{x+2} - 3 + y - x \end{aligned} \tag{37}$$

$$\#rand_{io} = x + y$$

$$\begin{aligned} SS_{cost} &= \#rand_{io} \times rand_{cost} \\ &\quad + (\#P - \#rand_{io}) \times seq_{cost} \end{aligned} \tag{38}$$

$$\mathrm{CR} = \frac{SS_{cost}}{\#P_{res} \times seq_{cost}} \tag{39}$$

Figure 11 plots the CR against the Oracle for the use case from which the Selectivity Increase driven policy suffers. For calculations, we use the characteristics of HDD and plot the CR as a function of x and y [$\#P$ could be derived from Eq. (37)]. The CR is a monotonically decreasing function that from an initial value of 10 for one random access, converges to a value of 2 for $x > 10$ (a more realistic case). From this analysis, 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 upper bound; hence, it is a more robust choice.

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 12 depicts such a distribution. We calculate the CR for this scenario. Our analysis shows the CR against the Oracle of 2.45 for 100 pages that decreases to the value of 2.0001 for 3M pages.

Worst case result distribution The previous analysis showed the worst case scenario with respect to the number of fault page reads. Nonetheless, for Elastic Smooth Scan, 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 Fig. 8c). 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 (except for the first page). Therefore, Smooth Scan will perform $\#P/2$ random accesses, and the same amount of sequential accesses (to fetch adjacent pages).

$$\#rand_{io} = \frac{\#P}{2} \tag{40}$$

$$\begin{aligned} SS_{cost} &= \#rand_{io} \times rand_{cost} \\ &\quad + (\#P - \#rand_{io}) \times seq_{cost} \end{aligned} \tag{41}$$

$$\begin{aligned} CR &= \frac{SS_{cost}}{\frac{\#P}{2} \times seq_{cost}} \\ &= \frac{\frac{\#P}{2} \times (rand_{cost} + seq_{cost})}{\frac{\#P}{2} \times seq_{cost}} = 11 \end{aligned} \tag{42}$$

The CR is calculated in Eq. (42). For characteristics of HDD, with $rand_{cost} = 10$ and $seq_{cost} = 1$, the CR reaches the value of 11 when compared against the Oracle, and is constant regardless of the table size. The same ratio decreases in the case of SSD ($rand_{cost} = 2$ and $seq_{cost} = 1$), reaching the value of 6. When compared to Full Scan (which is the optimal *existing* access path in this case), the CR of Elastic Smooth Scan is equal to 5.5 for HDD, and 3 for SSD.

Discussion Overall, Elastic Smooth Scan proves to be the most robust solution. This policy provides a *firm* upper bound on suboptimality with the maximum theoretical CR of 11 and 6 in the case of HDD and SSD, respectively, regardless of the table size. We thus choose Elastic Smooth Scan as the default policy in our experiments.

Additionally, our analysis shows that a morphing increase factor greater than 2 leads to a higher CR. For instance, for the previous analysis, the morphing increase factor of 10 for HDD gives a competitive ratio of 19. Therefore, we use the factor of 2 as the morphing increase factor for the Smooth Scan implementation.

7 Experimental evaluation

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

7.1 Experimental setup

Software Smooth Scan is implemented inside PostgreSQL 9.2.1 DBMS. To demonstrate the problem of robustness presented in Sect. 1, we use a state-of-the-art commercial 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 [76].

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 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. The memory setting thus does not impact our findings.

7.2 TPC-H analysis

TPC-H in DBMS-X In Fig. 1 in Sect. 1, we demonstrated the severe impact of suboptimal 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 indexes estimated to boost the performance of the TPC-H workload. In queries Q12 and Q19, the presence of indexes 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., 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 the 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 indexes proposed by the commercial system from the previous experiment (on the same workload).

Figure 13 shows the results for 5 interesting TPC-H queries that typically trigger robustness issues in databases. These queries represent "choke points" for testing data access locality [13]. They cover: (i) range predicates, (ii) a LIKE statement, (iii) an equality over string predicates, which are all known to be problematic predicates that cause cardinality misestimates, consequently leading to poor query performance. We show the performance of: (i) single table selections with selectivities from both sides of the spectrum (very high and very low), (ii) two-table joins covering both selectivity extremes, and (iii) a more complex query involving a 6-table join and predicates over multiple attributes as an exemplary case of complex decision-making workloads. Low and high selectivity experiments are chosen as extreme cases for Smooth Scan—showing that it can solve the problem of suboptimal index selection in the case of cardinality misestimates such as for Q12 and Q19 of DBMS-X (low selectivity), but also demonstrating the negligible overhead in the cases when the optimal choice can be made by a DBMS (high selectivity).

The brackets on the right-hand side of the query ID show the selectivity of the query. 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, which has selectivity of 30%. 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, together with the amount of data transferred from the disk. The query execution plans are given in Appendix [15].

Figure 13 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 I/O accesses over the LINEITEM table (shown in Table 2). By flattening (i.e., accessing adjacent pages) and avoiding repeated accesses, Smooth Scan reduces this number to 95K which results in much better performance.

Table 2 I/O analysis of TPC-H queries

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

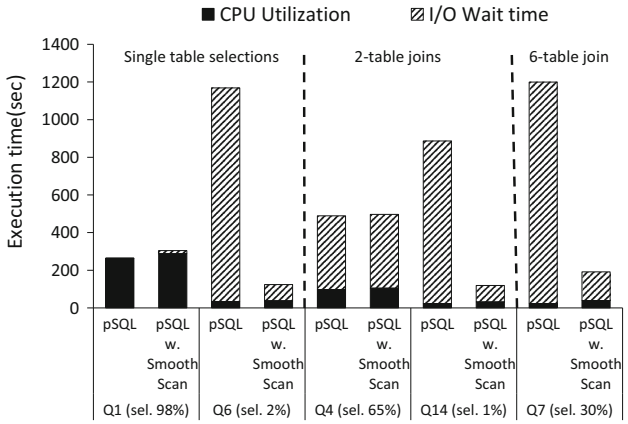

Fig. 13 Improving performance of TPC-H queries with Smooth Scan

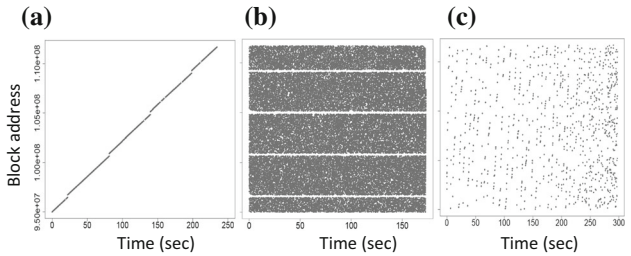

Fig. 14 Profiling I/O access of Q1: **a** full/sort scan, **b** index scan, **c** Smooth Scan

On the other hand, in query $Q1$ with selectivity of 98% the plain PostgreSQL chooses Sort Scan (also called Bitmap Heap Scan), which is the optimal path. However, even in this case Smooth Scan introduces only a marginal overhead; it quickly realizes that the result selectivity is high 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 for disk pages from 71K to 87K.

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 look-up on the outer table as well, it adjusts its access patterns quickly morphing its behavior toward sequential scan 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%, but still a factor of 2 more than what the optimizer estimated. 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). This query is an illustration for cases when performance of DBMS-X degraded severely (e.g., by a factor of 400). Furthermore, this is a major source of performance degradation in databases—a suboptimal index choice due to cardinality underestimates, typically as a consequence of attribute value independence assumption employed by most DBMSs [9,35,37,59,65,75]. Unlike the index scan that issues 416K I/O requests for this query, 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 morphing region expansion, 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. This degradation is due to the choice of the index over a range predicate with selectivity of 30%. The result cardinality of the range predicate was 18M as opposed to 300K which was the estimated value. Smooth Scan detects higher selectivity and naturally morphs toward a more desirable access pattern.

Discussion The memory structures of Smooth Scan 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 larger amounts of data from disk compared to the original access path (see Table 2), 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 suboptimal decision and only marginal overheads over optimal decisions.

Profiling I/O access To better grasp the behavior of Smooth Scan on disk, we profile the access of TPC-H Q1 with the iosnoop tool. Figure 14 depicts the disk accesses of all access paths when running Q1. The figure shows which device address (shown on the Y axis) is requested at what point in time (shown on the X axis). In the case of both full scan and sort scan, all pages are requested and accessed consecutively from the first until the last one. This is due to high

Table 3 Histogram of block transfer sizes

Block size	8 KB	16 KB	32 KB	64 KB	128 KB
# of transfers	3079	288	174	168	83263

selectivity of Q1 (98%), i.e., all pages have matching tuples. The index scan suffers from accessing pages repeatedly over time. Compared to the index scan, Smooth Scan requests fewer pages over time (and no repeated accesses occur). In the case of Smooth Scan the access is not purely sequential, as it is driven by occasional index probes which invoke random I/O accesses. This is evident in the last stages of execution where accesses to a single page are typically issued to fetch the data of interest, without expanding the morphing region. This is due to the fragmentation in accessed areas on disk, since the neighboring pages have already been processed in the past. Due to this reason and the fact that the result selectivity is high (98%), the highest number of data transfers was issued to the maximum block size of 128 KB[2] (83,263 transfers) followed by a single page 8KB requests (3079 transfers), as presented in Table 3.

7.3 Fine-grained analysis over full selectivity interval

This section provides the performance comparison of Smooth Scan against Full Scan, Index Scan and Sort Scan. In order to demonstrate the robust behavior of Smooth Scan, a micro-benchmark is used to stress test various aspects. 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 the tuple order number. The table contains 400M ($4 * 10^8$) tuples and occupies 25GB of disk space for 3M ($3 * 10^6$) pages each of which is 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];
```

Supporting an interesting order In this experiment, we show that Smooth Scan maintains tuple ordering and hence outperforms other alternatives for queries (or subplans) that require the ordering of tuples. Figure 15a 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 the execution of 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 to restore the proper ordering 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 attributed to avoiding the overhead of posterior sorting of tuples to produce results in the interesting order, from which Full Scan and Sort Scan suffer.

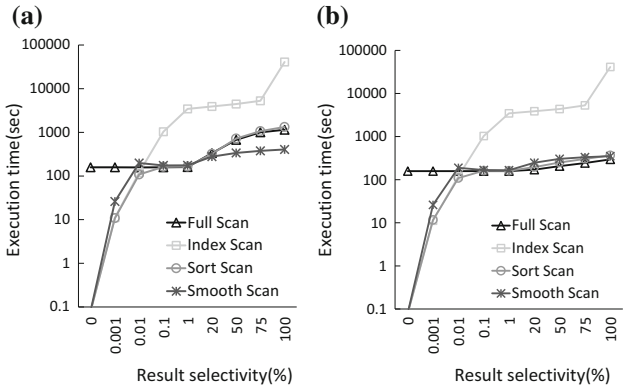

Fig. 15 Smooth Scan versus alternative access paths for a query with and without an *order by* clause. **a** With order by, **b** without order by

Without an interesting order Figure 15b 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 as page IDs are monotonically increasing. Smooth Scan does not suffer from the sorting overhead, but it does suffer from a periodical random I/O access 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 Fig. 23), 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 increases, like in the case of Index Scan. This is particularly important in real-life scenarios where a degra-

[2] 128 KB block requests were consecutive up to 16 MB of size, which is the maximum expansion region of Smooth Scan.

dation in Index Scan causes performance drops of several orders of magnitude [44]. 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 impractical. 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 query plan. In the experiments, Smooth Scan's CR reaches a maximum value of 2 over the optimal solution, for the case when selectivity is below 0.01%. To put absolute numbers in perspective, in our experiment a maximal overhead of 60 s is paid to prevent a worst case performance degradation of 11 h. In decision support systems that are characterized by long-running queries, this overhead is likely tolerable as a robustness guarantee for the prevention of severe performance drops that happen due to data correlation and skew.

7.4 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 on HDD versus SSD. 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 indexes when performing a tuple look-up in general hurts performance when selectivity increases. Figure 16 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 performance 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 when 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 Fig. 16 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 fetched data. 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.

Maximum morphing region size We perform a sensitivity analysis on the maximum number of adjacent pages up to which Smooth Scan performs the morphing region expansion. The experiment varies this number from 1000 up to 5000 pages, shown in Fig. 17 the query execution times for 3 cases, when selectivity is 1, 10 and 100%. We performed a fine-grained analysis over the entire selectivity range, and results followed the same trend; hence, for clarity we show only these 3 selectivity points. The experiments show that

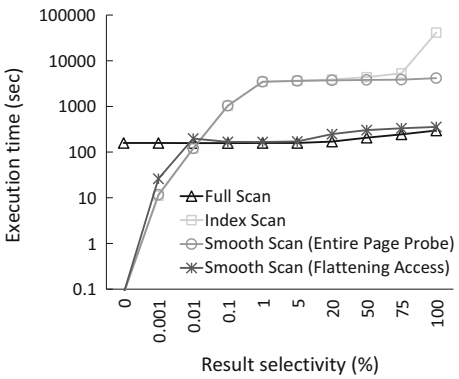

Fig. 16 Sensitivity analysis of Smooth Scan modes

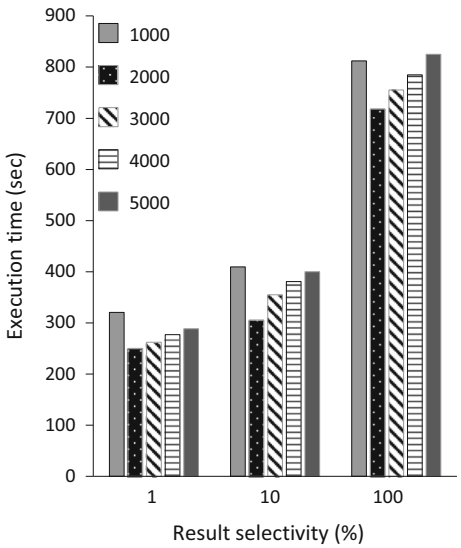

Fig. 17 Max. region size (# pages)

2000 pages are optimal, which translates to the morphing region size of 16MB. Thus, we keep 2000 as the maximum morphing region size for the rest of the experiments.

Impact of policy choices We plot the impact of policy choices in Fig. 18. The Greedy policy morphs with each index probe and hence converges to Full Scan faster than other policies. For lower selectivity, the Selectivity Increase and Elastic policies introduce less overhead compared to Greedy since they fetch fewer adjacent pages, i.e., more pages need to be seen for the morphing region 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 responsive to the changes in selectivity, we favor it in the rest of the experiments.

Impact of trigger choices Figure 19 plots 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 an SLA constraint; the calculated bound is shown as the dashed line in Fig. 19. 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.

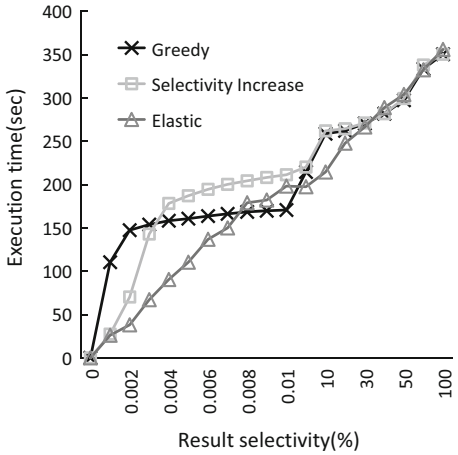

Fig. 18 Morphing policies

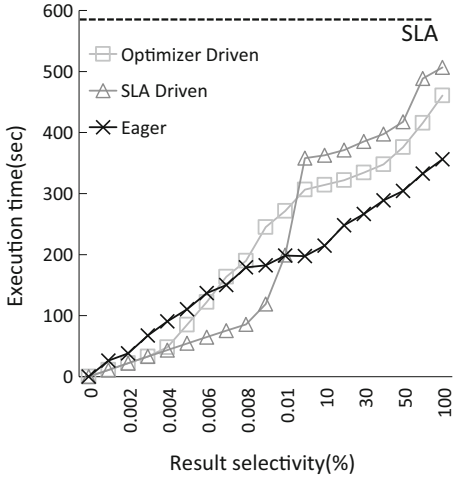

Fig. 19 Triggering choices

Overall, the Eager strategy strikes a balance in terms of overall performance; hence, we favor it as the default strategy in the remaining experiments. However, when in an environment where respecting SLAs is the main priority, or Smooth Scan serves as a means of fixing suboptimal decisions then the SLA or Optimizer driven strategies are viable alternatives. We can 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 tested on uniform distributions of result tuples. Many modern applications, however, exhibit non-uniform data distributions (e.g., stock markets, Internet networks [23,49]). 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 $c2 = 0$; afterward, another 0.001% of random tuples has 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 20a plots the execution time of Index Scan, Full Scan, Selectivity Increase and Elastic Smooth Scan; Fig. 20b plots the number of distinct pages fetched to answer the query. From Fig. 20b, one can see that Selectivity Increase 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; Selectivity Increase Smooth Scan notices the high selectivity increase at the beginning,

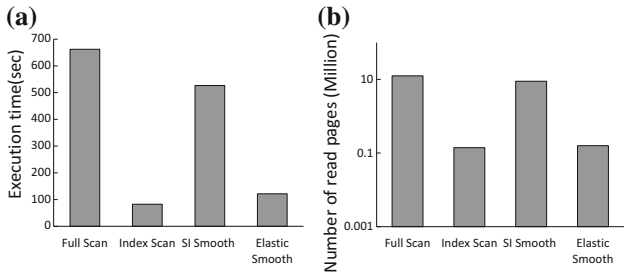

Fig. 20 Handling skew. **a** Execution time (1% sel), **b** number of read pages

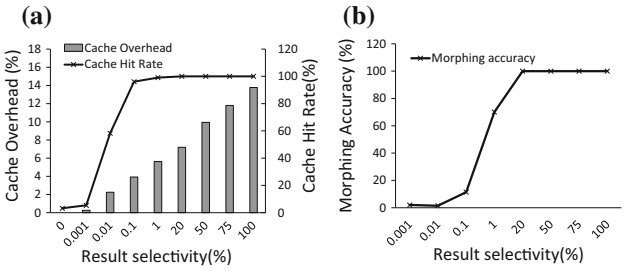

Fig. 21 Analysis of auxiliary data structures. **a** Result cache analysis, **b** morphing accuracy

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. This number is close to the number of pages accessed by Index Scan that fetched 140K pages. The severe impact of random I/O is not seen for Index Scan, since for this experiment the index key follows the page placement on disk.

From the experiment, one could observe that Elastic Smooth Scan continues to provide near-optimal performance, despite the significant initial skew. This is particularly important for long-running queries over big data, where data distributions tend to be non-uniform [58]. Approaches that employ one execution strategy, or run multiple alternatives shortly and stop 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, seamlessly adjusts its behavior to fit the data distribution.

The overhead of auxiliary data structures To avoid repeated page accesses, Smooth Scan in PostgreSQL uses the data structures described in Sect. 4. We now show the bookkeeping overhead of these structures and their usability rate, demonstrated on Q1 from the micro-benchmark with an ORDER BY clause.

Figure 21a shows that Result Cache adds a maximum 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 21b 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, 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.

Memory sensitivity of result cache Since Result Cache is the largest data structure, we perform a sensitivity analysis

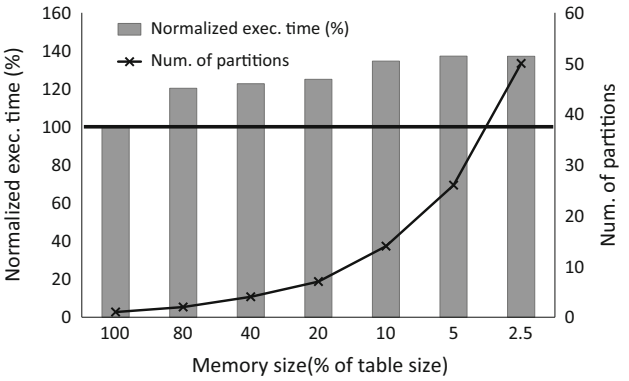

Fig. 22 Memory sensitivity of Result Cache

of Result Cache to the memory size. We run Q1 from the micro-benchmark, varying the Result Cache size from 2.5% of the table size to 100% of the table size. The table size is 25GB with 400M tuples stored, and the query has selectivity 100% throughout the entire experiment.

To see the overhead when partitions are spilled on disk, Fig. 22 plots the normalized execution time with respect to the execution time when Result Cache completely resides in memory, i.e., when no spilling occurs. As one can see from the graph, Smooth Scan is quite resilient to the memory size, adding only 37% of overhead when the memory size is 2.5% of the table size, i.e., it occupies only 625MB, compared to the case when all data stays in memory (i.e., no partitioning occurs). For the memory size of 2.5%, Smooth Scan builds 50 partitions in total shown by the black line in Fig. 22. Moreover, Smooth Scan is quite resilient to the number of partitions created. For instance, Smooth Scan adds only 3% of additional overhead for creating 50 partitions compared to 14 partitions for the case when the memory size is 10% of the table size. The biggest overhead increase of 20% is between 100 and 80% and is attributed to disk access. Once partitions start spilling to disk (in all other cases except 100% they do), the performance remains steady across a different number of partitions, because Smooth Scan enjoys the benefit of spatial locality when fetching partitions from disk.

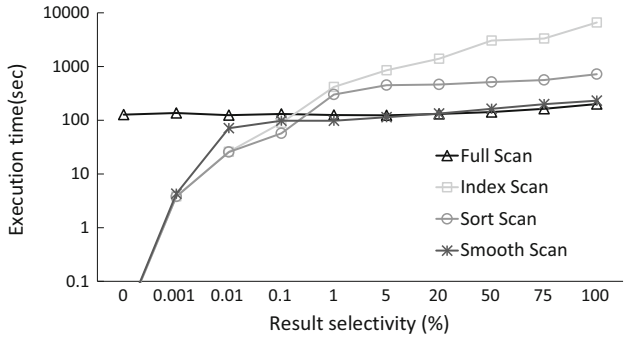

Fig. 23 Smooth Scan on SSD

7.5 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 [47], we now stress test Smooth Scan on SSD. We use a solid state disk OCZ Deneva 2C Series SATA 3.0 with advertised read performance of 550MB/s (offering 80kIO/s of random reads). We use query $Q1$ from the micro-benchmark without an order by clause and compare Smooth Scan against the existing access operators.

Figure 23 demonstrates that Smooth Scan benefits even more from solid state technology than from hard disks (shown in Fig. 15). SSD is 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 to 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), since the pre-sort overhead to obtain page IDs cannot be masked due to faster I/O performance.

Discussion Smooth Scan favors SSD over HDD, since occasional random jumps when following the index pointers do not hurt performance as much, compared to the sorting overhead of Sort Scan to pre-sort 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.

7.6 Cost model analysis

In this experiment, we show that the estimates of the analytical model derived in Sect. 5 correspond closely to the measured performance. Figure 24 compares the execution time and number of I/O requests of Full Scan, and Smooth Scan against the analytical cost model, shown as a function of result selectivity increase.

We model the costs for a table with 400M tuples from the micro-benchmark. For the page size we take the value of 8KB (default PostgreSQL page); for the tuple size we assume 68 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 of Smooth Scan with $\log_2(\#P + 1)$. Finally, for seq_{cost} we use 1, for $rand_{cost}$ we use 10, and for cpu_{cost} we use 10^{-6} (i.e., one I/O translates to 1M CPU cycles). Our disk has I/O transfer rate of 130MB/s, which for the block size of 128KB (the OS setting) gives the throughput of 1000 blocks per second. Thus, when transforming the analytical model into execution time, we use 1ms as the block transfer latency.

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 the experiment presented in Fig. 24a, where 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 that groups many sequential I/O requests from the disk controller queue into one in the case of Full Scan. This consequently 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 subarrays with random jumps in between. Although the same grouping of sequential subarrays could happen and equally improve performance, the disk controller did not possess logic to do so. For high selectivity, both Full Scan and Smooth Scan slightly exceed the estimates of the model. This is due to the overhead of tuple construction of PostgreSQL, that is not part of the model, but dominates the CPU cost in the case of high selectivity.

Figure 24b compares the estimated and measured I/O performance of Full Scan and Smooth Scan. Smooth Scan again exhibits close to estimated behavior in terms of the number of I/O requests. While for Full Scan the analytical model has a 2% of relative error, in the case of Smooth Scan for 100% selectivity the relative error is 11%, i.e., the model suggested I/O increase of 5% compared to Full Scan, while measured experiments capture 16% of I/O increase. When it comes to

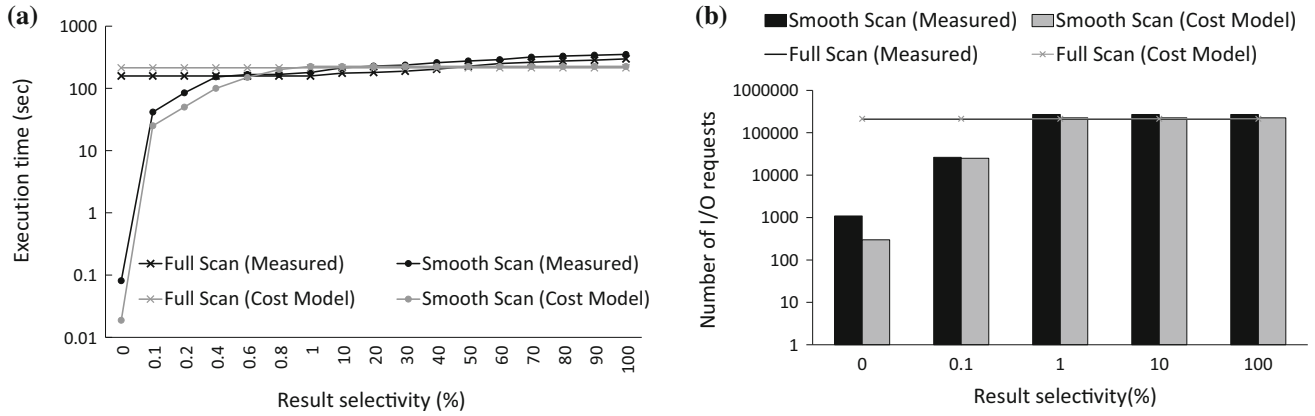

Fig. 24 Comparing the analytical model against measured performance. **a** Execution time, **b** I/O analysis

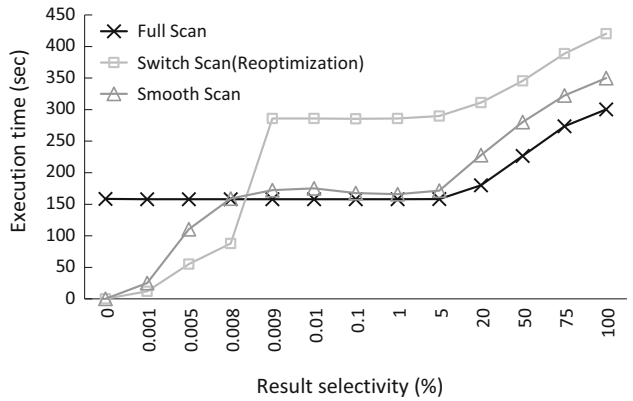

Fig. 25 Performance cliff and benefit of reoptimization

the CPU cost, both the analytical model and real execution observe a negligible CPU overhead due to Smooth Scan's operations. Since the CPU cost is less than 0.1% of the total cost, we do not present CPU measures separately.

7.7 The benefit of mid-operator reoptimization

We now study the benefit of mid-operator reoptimization as an alternative to preventing performance degradation. We demonstrate that although a simple solution can help in some cases (such as fulfilling SLA constraints for instance), there are consequences behind binary decisions such as performance cliffs or the inability to return once the decision has been made.

Figure 25 shows the benefit of mid-operator reoptimization implemented through an operator we refer to as Switch Scan. Switch Scan is implemented in PostgreSQL, existing side by side with the remaining access path operators. Switch Scan starts with following an index scan. During run time, it monitors the operator's selectivity and upon detecting the selectivity estimation violation, to prevent further degradation, it switches the access path strategy to full scan. Although pretty simplistic, Switch Scan bounds the worst case execution time to the time of obtaining X tuples (the optimizer's cardinality estimation) with the index look-up plus the time to perform the full table scan, which could still be significantly lower than the time to fetch all the tuples with the index look-up.

We report results of executing query $Q1$ from the micro-benchmark. In the case of Switch Scan, one 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, Switch Scan detects more than 32K tuples and performs 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 Switch Scan opts 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.

The performance hit together with the uncertainty whether the overhead incurred at the time of a change will actually be amortized over the remaining query time is perceived as *lacking in robustness*. In this example, if it were to receive only 32001 qualifying tuples (but not knowing it at the time), Switch Scan would pay the overhead that could not be amortized over the rest of the query life time and hence is unjustified. Moreover, since the decision depends on the accuracy of the statistics, this approach is highly volatile. Smooth Scan, on the other hand, manages to approach near-optimal performance throughout the entire selectivity interval as shown in Fig. 25, while being statistics oblivious.

7.8 Statistics collection overheads

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

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 $c3$ (i.e., 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 statistics. Completely accurate statistics are rarely present in such a system.

Figure 26 shows the statistics collection times on the table, comparing them against the execution time of query $Q2$ run on DBMS-X. We have measured statistics collection time on a commercial system, since this system supports a wider spectrum of possibilities than PostgreSQL. We compare the performance of Bitmap Scan, Full Scan and the optimizer's choice against the time to collect statistics. 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 ($c2$, $c3$)].

Despite being the cheapest alternative, the basic statistics could still lead to the choice of suboptimal plans as shown in Fig. 26a, since they cannot accurately detect neither skew nor the presence of column correlations. In the case of basic statistics presence, the optimizer kept the original access path choice (i.e., Bitmap Scan) throughout the entire selectivity range. On the other hand, one could observe that obtaining histograms on all columns introduces a higher cost as shown in Fig. 26b. 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 suboptimal decision for selectivity 40% in Fig. 26b). Therefore, whenever a query contains multiple filtering predicates over different columns, joint-data distributions are required. Figure 26c 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 [75].

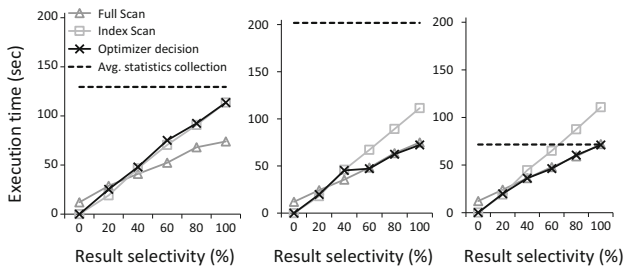

Fig. 26 Statistics collection in DBMS-X as an alternative to run-time adaptivity: **a** basic statistics, **b** single-column histograms, **c** joint-distribution histograms

Query $Q2$ is a simple query that showcases the problem with cardinality estimation. Assuming no accurate statistics exist on the table, the optimizer would fall into a trap of using the 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} ($10^{-5} * 10^{-5}$) [29]. Therefore, the optimizer would always opt for the index look-up, severely hurting performance in the case of higher selectivity [37,58,59,61–63,65].

8 Related work

The volatility of query optimizers does not only affect the quality of plans, but it might significantly decrease the overall user experience. Anecdotal evidence from the industrial leaders states that the angriest calls are from customers unsatisfied with their query performance [44,63]. With queries being increasingly complex, statistics being less available and more expensive to gather and data being even stored remotely, it is clear that the traditional optimize-then-execute query paradigm is becoming insufficient [12,37,56–59,62,65,67]. This has led to the need for having *adaptive query processing* techniques, where runtime feedback is used to monitor the current query execution strategy with a purpose of correcting the choice and providing a better query response time [8,35,50,81]. Adaptive query processing is an active area of database research that comes in several flavors [35], ranging from runtime statistics refinement, dynamic plan change through shuffling or reoptimization, and robust or multiple plans selection to the fine-grained adjustment within operators.

Run-time statistics refinement Missing or imprecise statistical information could be obtained at run time usually with low overhead, if the statistics collection procedure gets piggybacked on the query execution. Learned statistical information then can be *injected* [24] back in the

planning procedure and exploited by the current or future queries [2,20,21,25–27,74,75]. A step further is to explicitly trigger subplans to collect statistical information for particular parts of the plan search space (i.e., sensitive query fragments) [1,51,53,68] in order to remove uncertainty. In such cases, the execution and statics collection are usually interleaved, where newly gathered knowledge helps proposing better plans. Despite improving the quality of plans, there are environments where statistical information cannot be fully gathered (e.g., remote data sources, frequent data ingest [52], streaming, etc.). In such environments, plans need to be changed dynamically at run time.

Change through subplan shuffling Subplan shuffling is employed to deal with unexpected data arrival delays, usually due to effects of network transfer from remote sources typical for data integration systems. The employed techniques minimize the idle time during query processing by rescheduling the order of the subplans of the original plan [3,56,57,77]. The latter ultimately results in join reordering of the original plan. A step further is to fully interleave the scheduling and execution phase and trigger scheduling every time a data item becomes unavailable or a subplan finishes [19]. The highest level of adaptivity is achieved in Ingres [80] and with Eddies [6,70] where the order among the existing (predetermined) operators is reassessed and changed based on the data arrival and the observed selectivities of operators of the query plan.

Change through reoptimization Unlike shuffling, reoptimization performs full query optimization usually upon detecting a cardinality estimate violation [9,40,59,62,65]. When performing reoptimization, a special attention has to be paid to the treatment of intermediate results (already done work) that could be fully exploited or discarded [82]. It is also important to know when is a possible time to perform reoptimization to ensure the correctness of results [40,62].

Multiple plan choices Multi-plan techniques have been explored in the database community for the past decade. Multi-plan approaches choose a set of possible plans and execute them either in parallel [4,5] or each one on a disjoint subset of data [12,22,58,67,82]. Special cases of multi-plan choices are parametric [55], and dynamic plans [31,46], where from a set of plans determined at compile time, a specific plan or operator implementation is chosen based on the value of parameter markers obtained at run time. Similarly, Plan Bouquets [37] choose from a discretized space of parametric optimal plans the subset based on observed selectivity at run time, while Proactive Reoptimization proposes a set of *switchable plans* that could be safely interchanged without losing already processed work [9].

Robust plan selection Robust plans take into account the uncertainty of the optimization process and choose plans more resilient to the cardinality misestimates [7,30]. The plan search space can be pruned leaving only a subset of plans more resilient to the optimizer misestimates [33,34].

Adaptive operators All mentioned approaches that perform dynamic plan changes are examples of inter-operator adaptivity, where the adaptation mechanism is employed between operators, i.e., it mostly pertains to the operator order. Adaptive operators, on the other hand, are more fine-grained as they encapsulate the adaptation mechanism within their own algorithm [16,48,78,79].

Robustness and adaptation to data characteristics at the intra-operator level are considered in [4,5,11,28,42,66]. Despite a lot of efforts in fixing suboptimal decisions, little attention has been paid to the access path selection problem. Nonetheless, a suboptimal decision at the level of access paths has a highly detrimental effect on the overall query performance [14], since the access paths touch most of the data before any filtering has been applied.

Improving IO access Index-lookups cause poor disk performance due to random access latency. Asynchronous IO with prefetching [39] improves performance of such pattern but still suffers from repeated page reads. Partial sorting of tuples [36,39] can improve access locality and size, but unless the entire input is sorted, repeated page reads are still possible.

In this paper, we fill the need for adaptation at the access path level by introducing a hybrid adaptive access path called Smooth Scan. Smooth Scan guarantees nearly optimal performance throughout the entire range of possible selectivities, thereby preventing poor execution cases as a consequence of suboptimal decisions. Unlike [4,5], however, Smooth Scan does not waste any resources by doing double work, nor does it require a serious change of the database architecture. Moreover, since the high risk of having incomplete statistics in the case of ever-increasing data sets still remains, Smooth Scan is completely statistics oblivious.

9 Concluding remarks and the future ahead

With the increase in complexity of modern workloads and the technology shift toward cloud environments, robustness in query processing is gaining momentum. Still current systems remain sensitive to the quality of statistics. As a result, the run-time execution of queries may fluctuate severely as a result of marginal changes in the underlying data. For a productive user experience, the performance of every query must be robust, i.e., close to the expected performance, even with missing, stale, or insufficient statistics.

This paper introduces Smooth Scan, a *statistics-oblivious* access path 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 the

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 over the entire range of possible selectivities.

We believe that the impact of techniques presented in this paper could reach far beyond traditional (relational) DBMS, as similar access patterns with the same trade-off between the random and sequential I/O are observed in NoSQL database solutions [71,72]. Additionally, recent research has shown that access path selection is equally important for column stores and in memory analytics systems [60]. Similarly, it would be worth considering the adjustments of Smooth Scan to storage tiering hierarchy where data is spread across multiple tiers with different access latency properties.

Acknowledgements We would like to thank the organizers of the Dagstuhl seminar 12321 on "Robust query processing" for the inspirational sessions and the introduction of robustness issues in query processing. In particular, we thank Goetz Graefe for his support throughout this work.

References

1. Abdel Kader, R., Boncz, P., Manegold, S., van Keulen, M.: ROX: run-time optimization of XQueries. In: SIGMOD (2009)
2. Aboulnaga, A., Chaudhuri, S.: Self-tuning histograms: building histograms without looking at data. In: SIGMOD (1999)
3. Amsaleg, L., Franklin, M.J., Tomasic, A., Urhan, T.: Scrambling query plans to cope with unexpected delays. In: DIS (1996)
4. Antoshenkov, G.: Dynamic query optimization in Rdb/VMS. In: ICDE (1993)
5. Antoshenkov, G., Ziauddin, M.: Query processing and optimization in Oracle Rdb. PVLDB **5**(4), 229–237 (1996)
6. Avnur, R., Hellerstein, J.M.: Eddies: continuously adaptive query processing. In: SIGMOD (2000)
7. Babcock, B., Chaudhuri, S.: Towards a robust query optimizer: a principled and practical approach. In: SIGMOD (2005)
8. Babu, S., Bizarro, P.: Adaptive query processing in the looking glass. In: CIDR (2005)
9. Babu, S., Bizarro, P., DeWitt, D.: Proactive re-optimization. In: SIGMOD (2005)
10. Barber, R., Bendel, P., Czech, M., Draese, O., Ho, F., Hrle, N., Idreos, S., Kim, M., Koeth, O., Lee, J., Li, T.T., Lohman, G.M., Morfonios, K., Müller, R., Murthy, K., Pandis, I., Qiao, L., Raman, V., Szabo, S., Sidle, R., Stolze, K.: Blink: not your father's database! In: BIRTE (2011)
11. Bellamkonda, S., Li, H.G., Jagtap, U., Zhu, Y., Liang, V., Cruanes, T.: Adaptive and big data scale parallel execution in oracle. PVLDB **6**(11), 1102–1113 (2013)
12. Bizarro, P., Babu, S., DeWitt, D., Widom, J.: Content-based routing: different plans for different data. In: PVLDB, pp. 757–768 (2005)
13. Boncz, P.A., Neumann, T., Erling, O.: TPC-H analyzed: hidden messages and lessons learned from an influential benchmark. In: TPCTC (2013)
14. Borovica, R., Alagiannis, I., Ailamaki, A.: Automated physical designers: what you see is (not) what you get. In: DBTest (2012)
15. Borovica-Gajic, R.: Toward timely, predictable and cost-effective data analytics. Ph.D. thesis (2016)
16. Borovica-Gajic, R., Appuswamy, R., Ailamaki, A.: Cheap data analytics using cold storage devices. PVLDB **9**(12), 1029–1040 (2016)
17. Borovica-Gajic, R., Graefe, G., Lee, A.: Robust performance in database query processing (dagstuhl seminar 17222). Dagstuhl Rep. **7**(5), 169–180 (2017)
18. Borovica-Gajic, R., Idreos, S., Ailamaki, A., Zukowski, M., Fraser, C.: Smooth scan: statistics-oblivious access paths. In: ICDE (2015)
19. Bouganim, L., Fabret, F., Mohan, C., Valduriez, P.: Dynamic query scheduling in data integration systems. In: ICDE (2000)
20. Bruno, N., Chaudhuri, S.: Efficient creation of statistics over query expressions. In: ICDE (2003)
21. Bruno, N., Chaudhuri, S., Gravano, L.: STHoles: A multidimensional workload-aware histogram. In: SIGMOD (2001)
22. Cao, L., Rundensteiner, E.A.: High performance stream query processing with correlation-aware partitioning. PVLDB **7**(4), 265–276 (2013)
23. Cao, L.B.P.: Web caching and Zipf-like distributions: evidence and implications. In: INFOCOM (1999)
24. Chaudhuri, S.: Query optimizers: time to rethink the contract? In: SIGMOD (2009)
25. Chaudhuri, S., Narasayya, V., Ramamurthy, R.: A pay-as-you-go framework for query execution feedback. PVLDB **1**(1), 1141–1152 (2008)
26. Chaudhuri, S., Narasayya, V.R.: Automating statistics management for query optimizers. In: ICDE (2000)
27. Chen, C.M., Roussopoulos, N.: Adaptive selectivity estimation using query feedback. In: SIGMOD (1994)
28. Chen, S., Ailamaki, A., Gibbons, P.B., Mowry, T.C.: Inspector joins. In: VLDB (2005)
29. Christodoulakis, S.: Implications of certain assumptions in database performance evaluation. TODS **9**(2), 163–186 (1984)
30. Chu, F.: Least expected cost query optimization: What can we expect. In: Proceedings of the ACM Symposim on Principles of Database Systems, pp. 293–302 (2002)
31. Cole, R.L., Graefe, G.: Optimization of dynamic query evaluation plans. In: SIGMOD (1994)
32. Curino, C., Jones, E., Popa, R.A., Malviya, N., Wu, E., Madden, S., Balakrishnan, H., Zeldovich, N.: Relational cloud: a database service for the cloud. In: CIDR (2011)
33. D., H., Darera, P.N., Haritsa, J.R.: On the production of anorexic plan diagrams. In: VLDB (2007)
34. Harish, D., Darera, P.N., Haritsa, J.R.: Identifying robust plans through plan diagram reduction. PVLDB **1**(1), 1124–1140 (2008)
35. Deshpande, A., Ives, Z., Raman, V.: Adaptive query processing. Found. Trends Databases **1**(1), 1–140 (2007). https://doi.org/10.1561/1900000001
36. DeWitt, D.J., Naughton, J.F., Burger, J.: Nested loops revisited. In: PDIS (1993)
37. Dutt, A., Haritsa, J.: Plan bouquets: query processing without selectivity estimation. In: SIGMOD (2014)
38. Dutt, A., Narasayya, V.R., Chaudhuri, S.: Leveraging re-costing for online optimization of parameterized queries with guarantees. In: SIGMOD (2017)
39. Elhemali, M., Galindo-Legaria, C.A., Grabs, T., Joshi, M.M.: Execution strategies for SQL subqueries. In: SIGMOD (2007)
40. Eurviriyanukul, K., Paton, N.W., Fernandes, A.A.A., Lynden, S.J.: Adaptive join processing in pipelined plans. In: EDBT (2010)
41. Graefe, G.: Modern B-tree techniques. Found. Trends Databases **3**(4), 203–402 (2011)
42. Graefe, G.: New algorithms for join and grouping operations. Comput. Sci. **27**(1), 3–27 (2012)
43. Graefe, G., Guy, W., Kuno, H.A., Paulley, G.N.: Robust query processing (dagstuhl seminar 12321). Dagstuhl Rep. **2**(8), 1–15 (2012)
44. Graefe, G., König, A.C., Kuno, H.A., Markl, V., Sattler, K.U.: Robust query processing (dagstuhl seminar 10381). In: Robust Query Processing (2011)

45. Graefe, G., Kuno, H.A., Wiener, J.L.: Visualizing the robustness of query execution. In: CIDR (2009)
46. Graefe, G., Ward, K.: Dynamic query evaluation plans. In: SIGMOD (1989)
47. Gray, J.: Tape is dead, disk is tape, flash is disk. RAM locality is king, Presented at CIDR (2007)
48. Haas, P.J., Hellerstein, J.M.: Ripple joins for online aggregation. In: SIGMOD (1999)
49. Harris, L.: Stock price clustering and discreteness. Rev. Financ. Stud. **4**(3), 389–415 (1991)
50. Hellerstein, J.M., Franklin, M.J., Chandrasekaran, S., Deshpande, A., Hildrum, K., Madden, S., Raman, V., Shah, M.A.: Adaptive query processing: technology in evolution. IEEE Data Eng. Bull. **23**, 2000 (2000)
51. Herodotou, H., Babu, S.: Xplus: a SQL-tuning-aware query optimizer. PVLDB **3**(1–2), 1149–1160 (2010)
52. IBM: Managing big data for smart grids and smart meters. White paper. http://goo.gl/n1Ijtd (2012)
53. Ilyas, I.F., Markl, V., Haas, P., Brown, P., Aboulnaga, A.: CORDS: automatic discovery of correlations and soft functional dependencies. In: SIGMOD (2004)
54. Ioannidis, Y.E.: Query optimization. ACM Comput. Surv. **28**(1), 121–123 (1996)
55. Ioannidis, Y.E., Ng, R.T., Shim, K., Sellis, T.K.: Parametric query optimization. PVLDB **6**(2), 132–151 (1997)
56. Ives, Z.G.: Efficient Query Processing for Data Integration. University of Washington, Seattle (2002)
57. Ives, Z.G., Florescu, D., Friedman, M., Levy, A., Weld, D.S.: An adaptive query execution system for data integration. In: SIGMOD (1999)
58. Ives, Z.G., Halevy, A.Y., Weld, D.S.: Adapting to source properties in processing data integration queries. In: SIGMOD (2004)
59. Kabra, N., DeWitt, D.J.: Efficient mid-query re-optimization of sub-optimal query execution plans. In: SIGMOD (1998)
60. Kester, M.S., Athanassoulis, M., Idreos, S.: Access path selection in main-memory optimized data systems: should I scan or should I probe? In: SIGMOD (2017)
61. Leis, V., Gubichev, A., Mirchev, A., Boncz, P.A., Kemper, A., Neumann, T.: How good are query optimizers, really? PVLDB **9**(3), 204–215 (2015)
62. Li, Q., Shao, M., Markl, V., Beyer, K.S., Colby, L.S., Lohman, G.M.: Adaptively reordering joins during query execution. In: ICDE (2007)
63. Lohman, G.: Is query optimization a "solved" problem? In: ACM SIGMOD Blog (2014)
64. Mackert, L., Lohman, G.: R* optimizer validation and performance evaluation for local queries. In: SIGMOD (1986)
65. Markl, V., Raman, V., Simmen, D., Lohman, G., Pirahesh, H., Cilimdzic, M.: Robust query processing through progressive optimization. In: SIGMOD (2004)
66. Müller, I., Sanders, P., Lacurie, A., Lehner, W., Färber, F.: Cache-efficient aggregation: hashing is sorting. In: SIGMOD (2015)
67. Nehme, R.V., Rundensteiner, E.A., Bertino, E.: Self-tuning query mesh for adaptive multi-route query processing. In: EDBT, pp. 803–814 (2009)
68. Neumann, T., Galindo-Legaria, C.A.: Taking the edge off cardinality estimation errors using incremental execution. In: DBIS (2013)
69. Ramakrishnan, R., Gehrke, J.: Database Management Systems, 3rd edn. McGraw-Hill, New York (2003)
70. Raman, V., Deshpande, A., Hellerstein, J.M.: Using state modules for adaptive query processing. In: ICDE (2003)
71. Schindler, J.: I/O characteristics of NoSQL databases. PVLDB **5**(12), 2020–2021 (2012)
72. Schindler, J.: Profiling and analyzing the I/O performance of NoSQL DBs. In: SIGMETRICS (2013)
73. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: SIGMOD (1979)
74. Srivastava, U., Haas, P.J., Markl, V., Kutsch, M., Tran, T.M.: ISOMER: consistent histogram construction using query feedback. In: ICDE (2006)
75. Stillger, M., Lohman, G.M., Markl, V., Kandil, M.: LEO–DB2's learning optimizer. In: VLDB (2001)
76. TPC: TPC-H benchmark. http://www.tpc.org/tpch/
77. Urhan, T., Franklin, M.J., Amsaleg, L.: Cost-based query scrambling for initial delays. In: SIGMOD (1998)
78. Viglas, S., Naughton, J.F., Burger, J.: Maximizing the output rate of multi-way join queries over streaming information sources. In: VLDB (2003)
79. Wilschut, A., Apers, P.: Dataflow query execution in a parallel main-memory environment. In: PDIS (1991)
80. Wong, E., Youssefi, K.: Decomposition—a strategy for query processing. ACM Trans. Database Syst. **1**(3), 223–241 (1976)
81. Yin, S., Hameurlain, A., Morvan, F.: Robust query optimization methods with respect to estimation errors: a survey. SIGMOD Rec. **44**(3), 25–36 (2015)
82. Zhu, Y., Rundensteiner, E.A., Heineman, G.T.: Dynamic plan migration for continuous queries over data streams. In: SIGMOD (2004)