Smooth Scan: One Access Path to Rule Them All

Renata Borovica
École Polytechnique Fédérale de Lausanne

Stratos Idreos
Harvard University

Anastasia Ailamaki
École Polytechnique Fédérale de Lausanne

Marcin Zukowski
Snowflake Computing

Campbell Fraser
Google

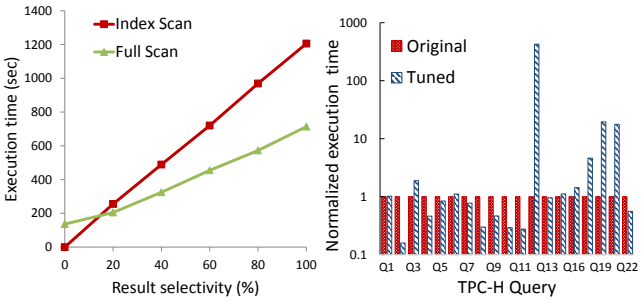
ABSTRACT

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 In many cases, though, statistics are outdated or non-existent, and the process of creating or refreshing statistics is very expensive, especially for ad-hoc workloads on ever bigger data.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

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

1. INTRODUCTION

Perils of Query Optimization Complexity. Query execution performance of database systems depends heavily on query optimization decisions; deciding which (physical) operators to use and in which order to place them in a plan is of critical importance and can affect response times by several orders of magnitude [22]. To find the best possible plan, query optimizers typically employ a cost model to estimate performance of viable alternatives. In turn, cost models rely on statistics about the data. With the growth in complexity of decision support systems and the advent of dynamic web applications, however, cost models' grasp of reality becomes increasingly loose and it becomes more difficult to produce an optimal plan [14]. For instance, to defy complexity and make up for lack of statistics, commercial database management systems often assume uniform data distributions and attribute value independence, which is in reality hardly the case [7]. As a result, database systems are increasingly confronted with suboptimal plans and subpar performance.



(a) Index Scan Vs Full Scan (b) TPC-H

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

Motivating Example. Figure 1 illustrates the severe impact of stale statistics. We use a state-of-the-art commercial system, referred to as DBMS-X (the exact set-up is discussed in Section 6.2.1). Figure 1a depicts the impact of choosing the wrong access path for a simple select query; while an index scan is significantly better when only a small part of the data qualifies, it suffers once more data is selected. The optimizer needs accurate statistics to anticipate the tipping point between alternatives and make the proper choice. In this case, DBMS-X actually uses a sophisticated index-scanning strategy, causing performance degradation of only a factor of 2. Much more severe degradation happens when using the traditional indexing strategy (a factor of 80 as discussed in Section 6). In addition, Figure 1b demonstrates the impact of suboptimal index choices after tuning DBMS-X for TPC-H; the graph shows normalized execution times over non-tuned performance. Even though we use the official tuning tool of DBMS-X, for several queries performance degrades significantly (up to a factor of 400) compared to the original case.

Robust Execution. The core of the problem of suboptimal plans lies in the fact that even a small estimation error may lead to a drastically different result in terms of performance. For instance, one tuple difference in cardinality estimation can *swing the decision* between an index scan and a full scan, possibly causing a significant performance drop. Overall, this results in *unpredictable performance* and makes systems *non-robust*. Stability and predictabil-

ity, that imply that similar query inputs should have similar execution performance, are major goals for industrial vendors towards respecting service level agreements (SLA) [27]. This is exemplified, nowadays, in cloud environments, offering paid-as-a-service functionality governed by SLAs in environments which are much more ad-hoc than traditional closed systems. In these cases, a system's ability to efficiently operate in the face of unexpected and especially adverse[1] run-time conditions becomes more important than yielding great performance for one query input while suffering from severe degradation for another [16]. With respect to predictability, the choice of a full table scan is a preferable access path that should always be favored. Nonetheless, such an approach wastes resources by performing a brute-force access to all data, missing opportunities when there exists a better alternative. We define *robustness in the context of query processing as the ability of a system to efficiently cope with unexpected and adverse conditions, and deliver near-optimal performance for all query inputs.*

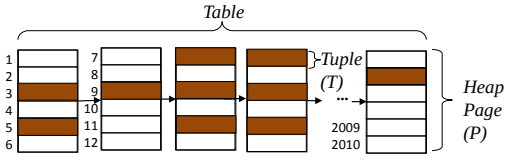
Past work on robustness focuses primarily on dealing with the problem at the optimizer level [9, 8, 4]. Nonetheless, in dynamic cloud environments with constantly changing workloads and data characteristics, judicious query optimization performed up front could bring only partial benefits as the environment keeps changing even after optimization. Thus, here, we take a drastically different route to address the problem at its core.

Smooth Scan. We introduce a paradigm of building a novel class of access path operators designed with the goal of providing robust performance every single time, regardless of the severity of the optimizer's errors in cardinality estimation. Our main intuition is to shift part of the query optimization decisions from the query optimization phase to the query execution phase. The reason is that during query execution, and as the system touches more data, it gets more knowledge about data properties. In fact, the understanding about the underlying data is a continuous process that develops throughout the execution of a query plan. This is why, in order to achieve this conceptual shift in query processing, we need a new class of *morphable operators* that *continuously, adaptively and seamlessly* adjust their execution strategy as the understanding of the data evolves. We introduce Smooth Scan, an operator that morphs from an index look-up into a full table scan, achieving near-optimal performance regardless of the predicate's selectivity. Our aim is to provide graceful degradation and be as close as possible to the performance that could be achieved if all necessary statistics were available and up to date. In addition, morphing relieves the optimizer from choosing an optimal access path a priori, since the execution engine has the ability to adjust its behavior at runtime as a response to the observed operator cardinality.

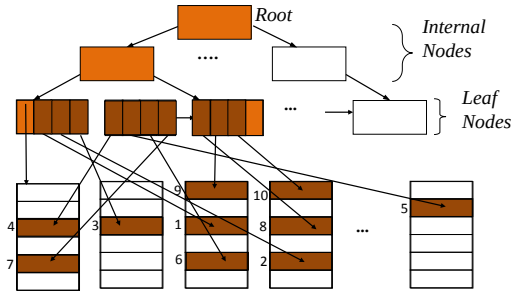
Contributions. Our contributions are as follows:

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

[1]Adverse in this context refers to unfavorable conditions from the aspect of query processing, e.g. receiving more tuples from an operator than estimated, or having less available memory than estimated.



(a) Full Table Scan



(b) Index Scan

Figure 2: Access Paths in a DBMS.

2. BACKGROUND

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

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

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

Sort Scan (Bitmap Scan) represents a middle ground between the previous two approaches. Essentially, Sort Scan still exploits

the secondary index to obtain tuple identifiers (IDs) of all tuples that qualify, but prior to accessing the heap, the qualifying tuple IDs are sorted in an increasing heap page order. That way the poor performance of the random access pattern gets translated into a (nearly) sequential pattern, 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* [29]. An advantage of B-tree indices comes from the fact that tuples are accessed in the sorted order of attributes on which the index is built. Sorting of tuple IDs based on their page placement breaks the natural index ordering that needs to be restored by introducing a sorting operator above the index access (or up in the tree). In addition, the blocking operator so early in the execution plan could stall the rest of the operators; if they require a sorted input, their execution can start only after the second sort finishes.

3. SMOOTH ACCESS PATHS

In this section, we present the concept of smooth access path operators that adjust their execution strategy at run-time to fit the observed data statistics. First, we discuss Switch Scan that switches access path strategy with a binary decision during query processing. Then, we introduce Smooth Scan which instead of making on/off decisions, gradually and adaptively shifts its behavior between access path patterns, avoiding performance drops.

3.1 Switch Scan Operator

The main problem with suboptimal access paths comes from a wrong selectivity estimation. One approach to resolve the problem, is to monitor result cardinality during query execution and *switch* access path strategy when we realize that the initial estimation was wrong. For example, assume that the optimizer instructs that an index scan should be used. The system may monitor the result cardinality during the execution of the select operator. Once the actual cardinality exceeds the expected result cardinality, we can simply throw away all the work performed until that point and restart the execution with a different access path. A more advanced approach would be to try to reuse as much as possible of the existing intermediate results, i.e., by remembering which tuples and pages the previous access path already visited.

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

3.2 Smooth Scan Operator

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

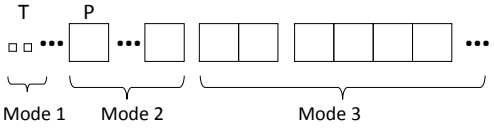


Figure 3: Smooth Scan Mode Expansion Size.

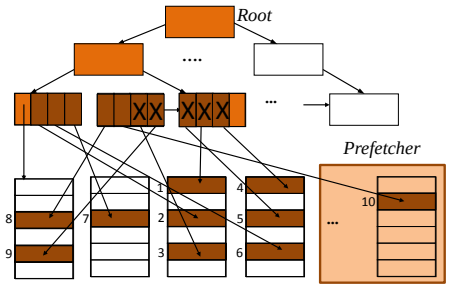


Figure 4: Smooth Scan Access Pattern.

3.2.1 Going Smooth

During its lifetime Smooth Scan can be in three modes whose morphing granularities are depicted in Figure 3. In each mode the operator performs a gradually increasing amount of work as a result of the selectivity increase.

Mode 1: Index Scan. Assuming the existence of a non-clustered 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 (denoted as *T* in Figure 3). 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 2: Entire Page Probe. To avoid repeated page accesses, 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 (denoted as *P* in Figure 3). The high selectivity implies a high probability that the same page will be needed more than once. Therefore, Smooth Scan invests CPU cycles for reading additional tuples from each page without extra I/O cost. Figure 4 depicts the access pattern of a Smooth Scan in this mode. As in Figure 2 the number at the left-hand-side of each tuple indicates the order in which the access path touches this tuple; notice that within each page Smooth Scan accesses tuples sequentially.

Mode 3: Flattening Access. When the result cardinality grows even further, Smooth Scan amortizes the random I/O cost of accessing the various pages 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 read, Smooth Scan prefetches a few more adjacent pages (which can be read sequentially). An example is depicted in Figure 4 as the orange rectangle over the heap pages.

Mode 3+: Gradually Flattening Access. Flattening Access Mode is in fact an ever expanding mode. When it first enters Mode 3, Smooth Scan starts by fetching one extra page for each page it needs to access. However, as result cardinality increases further, Smooth Scan progressively increases the number of pages it prefetches by multiplying it with a factor of 2 (notice the expanding pattern in Figure 3). The rationale behind this decision is that, as

selectivity increases, the I/O increase of fetching even more potentially unnecessary pages could be masked by the CPU processing cost of the tuples that qualify. In this way, as result cardinality increases more, Mode 3 keeps expanding, and conceptually Smooth Scan morphs more aggressively into a full table scan.

3.2.2 Morphing Policies

An important question is what triggers Smooth Scan to morph and change modes during query execution. The parameter which triggers shifts may be driven by several factors.

Optimizer Driven. The first direction is to initiate morphing, i.e., switching from Mode 1 to the next mode, once the result cardinality exceeds the optimizer's estimate. A cardinality violation is a clear indication that the optimizer's estimate is inaccurate and that the chosen access path might be suboptimal. Once Smooth Scan is in Mode 2, one safe approach (with robustness in mind) is to continuously morph with every new page we fetch.

Selectivity Driven. Blindly morphing to the next stage, though, may lead to lost opportunities in certain cases. A more adaptive approach is that Smooth Scan morphs from Mode 2 and on with a selectivity driven policy. With this strategy, Smooth Scan continuously monitors selectivity as it fetches more data, and it morphs every time that selectivity reaches the next level. This can be a design parameter that defines how optimistic or pessimistic Smooth Scan is. For example, assume that the selectivity step is set to 1% of the table size (assuming we know the table size[2]). Then, Smooth Scan monitors the global selectivity (as the ratio between the number of qualifying tuples and the total number of tuples in the table) and every time the result selectivity increases by 1%, Smooth Scan morphs to the next stage. This way we do not introduce too much overhead when selectivity is low, since we need to see more pages before we increase the morphing step. On the contrary, when selectivity is high, the selectivity threshold is reached much faster (and much more frequently), which in turn triggers more aggressive morphing to force sequential access patterns. Hence, morphing is performed at a pace which is purely driven by the data and the query at hand.

Region Density Driven. When considering big data sets, it is unlikely that one execution strategy will be optimal throughout the entire operator lifetime; dense and sparse regions of interest frequently appear in such a context due to skewed data distributions. By being conservative and reacting promptly to the increased result selectivity, Smooth Scan already provides graceful degradation and prevents the execution time blow-up as a consequence of skew. Nonetheless, with Smooth Scan we can go further and actually benefit from this density discrepancy. To use skew as an opportunity and not just a challenge, according to this policy Smooth Scan morphs two-ways, it increases the morphing size when we are in the dense region, and decreases it when the local selectivity over the morphing window decreases. More precisely, we compare the operator selectivity over the so far seen data ($\#qualifying_{tuples}/\#seen_{tuples}$) against the local selectivity over the last morphing range; if the local selectivity factor is higher than the selectivity factor over the seen tuples, that implies we are in a denser region, hence we increase the morphing step in this area by doubling its size. In the counter case, we decrease the morphing size for the next prefetch.

SLA Driven. Another option is that we take action in order to satisfy a given performance threshold, i.e., a service level agreement, SLA. An SLA driven approach is particularly targeted towards environments that need to provide a guarantee on the worst case scenario performance. For example, we would like to make sure that the access operator never exceeds a given time T. In this case, Smooth Scan continuously monitors execution and has a running estimate of the expected total cost (see discussion on the cost models in the next section). The moment it realizes that unless it switches to more conservative modes it will not be able to guarantee the performance targeted by the SLA (calculated for the worst case scenario), it morphs.

Pessimistic Approach. Assuming a worst case scenario, Smooth Scan can start in Mode 2 immediately as of the first tuple. Then it can keep morphing either by a selectivity based approach or by an SLA driven approach or even a fully pessimistic approach can morph to the next stage after each index probe. By omitting Mode 1, we guarantee that the total number of page accesses will be equal to the total number of heap pages in the worst case. Moreover, with this policy there is no need to record tuples produced in Mode 1 (to prevent result duplication), which provides additional benefit and decreases bookkeeping information.

We study the policies in detail during our experimental analysis.

4. INTRODUCING SMOOTH PATHS INTO POSTGRESQL

In this section, we discuss the design details of smooth access operators, and their interaction with the remaining query processing stack. We implement our operators, both Switch Scan and Smooth Scan family in PostgreSQL 9.2.1 DBMS as classical physical operators existing side by side with the traditional access path operators. During query execution, the access path choice is replaced by the choice of Smooth Scan, while the upper layers of query plans generated by the optimizer stay intact.

Switch Scan. Although a separate operator, Switch Scan could conceptually be considered as an instance of Smooth Scan with a threshold driven policy (usually the optimizer's result cardinality estimate) that abruptly switches to Full Scan after reaching the threshold. Therefore, we do not discuss it separately; except when performing the experimental evaluation in Section 6.4.

4.1 Design Details

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

Tuple ID Cache. When switching from Mode 1 (if Mode 1 is employed) to Mode 2, we have to ensure that the same result tuple will not be produced multiple times. This could happen if a result tuple is produced by following the index pointer in Mode 1, and later on we fetch the same page with Mode 2. To address this issue, Smooth Scan keeps a cache of all tuple IDs produced in Mode 1 in a bitmap-like structure, with setting bits from a page to 1 for corresponding tuples produced in Mode 1. Later, while producing tuples in Mode 2 Smooth Scan performs a bit check if the tuple has already been produced.

Page ID Cache. To avoid reading and processing the same heap page twice, 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 simple bit check is performed for every destination heap page. Smooth Scan will access the heap page only if that page has not been accessed before. Otherwise, we just mark the leaf pointer as visited (marked with X in Figure 4) and continue the leaf traversal.

Result Cache. If an index is chosen to support an interesting or-

[2]The data volume is usually a known variable upon data loading, or can be derived from the metadata quickly. What is unknown in practice, however, and much harder to obtain are data distributions, detecting skew or correlation between attributes.

der (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 the tuples the moment it produces them in Modes 2 and 3. To address this, the additional qualifying tuples found in Modes 2 and 3 (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 data structure where the hash key corresponds to the tuple ID, and the value corresponds to a projected qualifying tuple. 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.

Discussion. Both the Page ID Cache and the Tuple ID Cache are bitmap structures, meaning that their size is significantly smaller than the data set size, therefore they easily fit in memory. To illustrate, their size is usually a couple of MB for hundreds of GB of data. Moreover, in the Tuple ID cache we keep only the tuples acquired in Mode 1, 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. For instance, if the access pattern is such that tuples populated to the Result Cache are soon needed, the cache will return them and shrink back[3]. In the worst case if the cache grows above the allowed memory size, we still have the traditional index access as an overflow mechanism. The index helps us filter out unnecessary data, while the I/O reduction still brings a significant benefit when compared to the original access path choice.

4.2 Interaction with Query Processing Stack

Smooth Scan is an access path targeted primarily at preventing severe degradation in case of suboptimal index decisions taken by the optimizer. Nonetheless, its impact goes much beyond.

Query Optimization. Query optimization is simplified when considering the access path selection problem, since instead of trying to find the tipping point between the two alternatives during query optimization (and frequently making a mistake), the decision is encapsulated inside the operator that adjusts its behavior at runtime to fit the observed data distributions. Therefore, by allowing only a small query execution overhead this burden is completely removed from the optimizer.

Interaction with Other Operators. Smooth Scan is able to completely replace the functionality of both Index and Full Scan. In case Smooth Scan needs to provide tuples in a sorted order by pipelining them to Merge Join (MJ) from the outer input for instance, the variant of Smooth Scan with the result caching will be performed. If instead Index Nested Loops Join (INLJ) is performed, Smooth Scan does not have to respect the order, hence it can produce tuples the moment it finds them[4]. If Smooth Scan serves as an inner input (essentially as a parameterized path) to a join, the results produced per join key could be produced in an arbitrary order. Consequently, the repeated I/O accesses are avoided and random ones are significantly reduced per join key value, which particularly helps in the case of skewed data distributions.

Beyond Traditional Operators. We have seen how Smooth Scan enables graceful degradation of joins, by reducing random and repeated I/O accesses either at the table level or per join key value (e.g. when served as an inner input). Nonetheless, by employing the same concept of smooth morphing and transformation, we could benefit even more at the level of join operators. By performing caching of (qualifying) tuples from the inner input found along the way (i.e., for each page we fetch, we put the remaining tuples in the cache), INLJ morphs into a variant of Hash Join (HJ) overtime, with the index used purely as an overflow mechanism. Similarly, MJ morphs into a symmetric Hash Join [35], frequently used in data streaming environments due to its pipelining nature and amenability for operator reordering at run-time.

Ultimately, by having a morphable join operator, and a morphable access path, the need for a sophisticated query optimizer is obviated, since the only decision left (we believe also reparable) is on the join ordering. We leave a discussion on the join operators as an avenue of future work, and do not employ the proposed optimization in this work.

[3]The deletion is done in a bulk fashion. We organize the cache in a number of smaller caches that could be deleted once all tuples from an instance are produced. By organizing the cache per join key value, we can remove all items once the key value is traversed.

[4]Of course, if the ordering requirement is placed by some of the operators up in the tree, we still employ the first option.

Table 1: Cost model parameters

Parameter	Description
T_S	Tuple size (bytes).
$\#T$	Number of tuples in the relation.
P_S	Page size (bytes).
$\#T_P$	Number of tuples per page.
$\#P$	Number of pages the relation occupies.
K_S	Size of the indexing key (bytes).
sel	Selectivity of the query predicate(s) (%).
$card$	Number of result tuples.
$card_{mX}$	Number of tuples obtained with Mode X.
$m1_{check}$	0 or 1. Is Mode 1 employed?
$rand_{cost}$	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).
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.
$\#P_{res}$	Number of pages containing result tuples.
$\#rand_{io}$	Number of random accesses.
$\#seq_{io}$	Number of sequential accesses.
OP_{io_cost}	Cost of an operator in terms of I/O.
OP_{cpu_cost}	Cost of an operator in terms of CPU.

5. MODELING SMOOTH ACCESS PATHS

To better grasp the behavior of different access path alternatives, and especially to answer the critical questions of which policy and mode we should use and when, in this section we model the operators analytically. Since Smooth Scan trades CPU for I/O cost reduction, we model the cost of the operators both in terms of the number of disk I/O accesses, and their CPU cost. Since one I/O operation maps to an order of million CPU cycles [12], it is expected the overall cost to be dominated by the I/O component; nonetheless, we disclose CPU costs for completeness. Moreover, we make a distinction between the cost of a sequential and random access, since the nature of the accesses drives the overall query performance.

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 of the traditional operators are found in database text books. We assume indices are implemented as B^+-trees, i.e., each parent has k

children, where k is the tree fanout. Equations 1-7 are base formulas used for calculating the cost of all access path operators. The final cost of every operator is a sum of its I/O and CPU costs. We simplify the calculations by assuming every heap (and index) page is filled completely(100%). Similarly, we assume heap pages and index pages are of the same size (P_S). Lastly, to simplify the equations we assume that T_S already includes a tuple overhead (usually padding and a tuple header). In Eq. (3), we calculate the fanout of the B$^+$-tree by adding 20% of space per key for a pointer to a lower level; we further use this value to calculate the height of the tree. For Eq. (4) and (7), we assume that every tuple stored in a heap page has a pointer to it in a leaf page of the index[5].

$$\#T_P = \lfloor \frac{P_S}{T_S} \rfloor \quad (1)$$

$$\#P = \lceil \frac{\#T}{\#T_P} \rceil \quad (2)$$

$$fanout = \lfloor \frac{P_S}{1.2 \times K_S} \rfloor \quad (3)$$

$$\#leaves = \lceil \frac{\#T}{fanout} \rceil \quad (4)$$

$$height = \lceil \log_{fanout}(\#leaves) \rceil + 1 \quad (5)$$

$$card = sel \times \#T \quad (6)$$

$$\#leaves_{res} = \lceil \frac{card}{fanout} \rceil \quad (7)$$

$$OP_{cost} = OP_{io\_cost} + OP_{cpu\_cost} \quad (8)$$

Full Table Scan. A major characteristic of the full table scan is that its cost does not depend on the number of tuples that qualify for the given predicate(s). Thus, regardless of the selectivity of the operator its cost remains constant. As shown in Eq. (9), the I/O cost is equal to the cost to fetch all pages of the relation sequentially (as we expect each table to be stored sequentially on disk). Once we fetch a page, we perform a tuple comparison for all tuples from the page to find the ones that qualify. In Eq. (10) (and further on) we assume that each comparison invokes one CPU operation.

$$FS_{io\_cost} = \#P \times seq_{cost} \quad (9)$$

$$FS_{cpu\_cost} = \#T \times cpu_{cost} \quad (10)$$

Index Scan. The cost of the index access is fully driven by the operator selectivity. In order to fetch the tuples with the (non-clustered) index scan, we traverse the tree once to find the first tuple that qualifies (*height* in Eq. (11)). For the remaining tuples, we continue traversing the leaf pages from the index ($\#leaves_{res} \times seq_{cost}$), and use tuple IDs we found to access the heap pages, potentially triggering a random I/O operation per look-up (*card* in Eq. (11)). While traversing the tree for every index page we perform a binary search in order to find a pointer of interest to the next level ($height \times log_2(fanout)$ in Eq. (12)). For each tuple obtained by following the pointers from the leaf we perform a tuple comparison to see whether it qualifies (the second part of Eq. (12)).

$$IS_{io\_cost} = (height + card) \times rand_{cost} + \#leaves_{res} \times seq_{cost} \quad (11)$$

$$IS_{cpu\_cost} = (height \times log_2(fanout) + card) \times cpu_{cost} \quad (12)$$

Smooth Scan. Having defined the cost of the basic strategies, we move on to defining the cost of Smooth Scan. We calculate the cost of each mode separately. Overall result cardinality is split between the three modes (as shown in Eq. (13)). Like the index scan, the cost of the smooth scan access is driven by selectivity. Assuming uniform distribution of the result tuples (worst case scenario), the number of pages containing the result is calculated in Eq. (14).

$$card = card_{m1} + card_{m2} + card_{m3} \quad (13)$$

$$\#P_{res} = min(card, \#P) \quad (14)$$

Mode 1: Index Scan. The cost to obtain first $card_{m1}$ tuples is identical to the cost of the index scan for the same number of tuples. The only difference is in calculating the CPU cost of the operator in Mode 1, since now in addition to the tuple comparison we add each tuple ID to the Tuple ID cache (which explains the multiplier 2 in Eq. (16)).

$$SS_{io\_cost\_m1} = (height + card_{m1}) \times rand_{cost} + \#leaves_{res} \times seq_{cost} \quad (15)$$

$$SS_{cpu\_cost\_m1} = (height \times log_2(fanout) + card_{m1} \times 2) \times cpu_{cost} \quad (16)$$

Mode 2: Entire Page Probe. We also assume uniform distribution of tuples for which Mode 2 is going to be employed (calculated in Eq. (17)). Every page is assumed to be fetched with a random access (Eq. (18)). Once we obtain a page, we perform a tuple comparison for all tuples from the page (the first part of Eq. (19)). Before fetching the page we check whether the page is already processed, and upon its processing we add it to the Page Cache (the second part of Eq. (19)). Finally, if Mode 1 has been employed for each tuple we have to perform a check whether the tuple has already been produced in Mode 1 (the third part Eq. (19)). In the case we need to support an interesting order, the Result Cache will be used as a replacement for the Tuple ID cache functionality. In that case we only mark the tuple ID as a key in the cache, without copying the actual tuple as a hash value; the probe match without the actual result thus signifies that the tuple has already been produced. Thus, the CPU cost remains (roughly) the same in both cases.

$$\#P_{m2} = \min(card_{m2}, \#P) \quad (17)$$

$$SS_{io\_cost\_m2} = \#P_{m2} \times rand_{cost} \quad (18)$$

$$SS_{cpu\_cost\_m2} = (\#P_{m2} \times \#T_P + \#P_{m2} \times 2 + \#P_{m2} \times \#T_P \times m1_{check}) \times cpu_{cost} \quad (19)$$

Mode 3: Flattening Access. We calculate the maximum number of pages to fetch with Mode 3 in Eq. (20). Notice that pages processed in Mode 2 are skipped in Mode 3. The nature of the morphing expansion in Mode 3 of Smooth Scan is described with Eq. (21). The solution of the recurrence equation is shown in Eq. (22). In our case, n is the number of times we expand the morphing size (which is also the number of times we perform a random I/O access) and $f(n)$ translates to the number of pages to fetch with Mode 3($\#P_{m3}$). In Eq. (23) we calculate the minimum number of random accesses (jumps) to fetch all pages containing the results. We obtain this number from Eq. (24). It is worth noting that 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 could at worst be equal to the number of pages that contain the results. Nonetheless, there is an upper bound to it, equal to the logarithm of the number of pages in total.

SLA policies are mostly targeted towards specifying a cost upper bound. Since both formulas (24) and (25) quickly converge to the same value equal to $log_2(\#P+1)$, we model the worst case and use this value in the remainder of the section. Notice that the worst case scenario equal to the number of all pages is not possible, since the

[5]Some DBMS keep a list of distinct keys in leaves, but for PostgreSQL this assumption holds.

morphing expansion happens following the index access direction. The I/O cost of Mode 3 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 3 is identical to the cost per page in Mode 2 (Eq. (27)).

$$\#P_{m3} = \min(card_{m3}, \#P - \#P_{m2}) \quad (20)$$

$$f(i+1) = 2 \times f(i) \quad (21)$$

$$f(0) = 0, i = 0..n$$

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

$$\#P_{m3} = \sum_{i=0}^{\#rand_{io}(m3\_min)} 2^i \implies$$

$$\#P_{m3} = 2^{\#rand_{io}(m3\_min)} - 1 \quad (23)$$

$$\#rand_{io}(m3\_min) = \log_2(\#P_{m3} + 1) \quad (24)$$

$$\#rand_{io}(m3\_max) = \min(\#P_{m3}, \log_2(\#P + 1)) \quad (25)$$

$$SS_{io\_cost\_m3} = \#rand_{io}(m3) \times rand_{cost} + (\#P_{m3} - \#rand_{io}(m3)) \times seq_{cost} \quad (26)$$

$$SS_{cpu\_cost\_m3} = (\#P_{m3} \times \#T_P + \#P_{m3} \times 2 + \#P_{m3} \times \#T_P * m1_{check}) \times cpu_{cost} \quad (27)$$

The I/O and CPU costs are equal to the sum of the costs of all the modes employed. Finally, the overall cost is the sum of the operator CPU and I/O costs (Eg. (28)).

$$SS_{io\_cost} = SS_{io\_cost\_m1} + SS_{io\_cost\_m2} + SS_{io\_cost\_m3}$$

$$SS_{cpu\_cost} = SS_{cpu\_cost\_m1} + SS_{cpu\_cost\_m2} + SS_{cpu\_cost\_m3}$$

$$SS_{cost} = SS_{io\_cost} + SS_{cpu\_cost} \quad (28)$$

The cost model formulas allow us to predict the cost of Smooth Scan policies, or to decide when is time to trigger a mode change. For instance, for the SLA driven policy we know the overall operator cost defined by an SLA. Based on that cost and Eq. (28), we could calculate the cardinality, i.e., the triggering point of Mode 2 (and 3) calculated for the worst case scenario (selectivity 100%).

6. EXPERIMENTAL EVALUATION

In this section, we present a detailed experimental analysis of Smooth Scan. We demonstrate that Smooth Scan achieves robust performance in a range of synthetic and real workloads without having accurate statistics, while existing approaches fail to do so. Smooth Scan does not strive to achieve the best possible performance. Instead, its goal is to prevent severe degradation due to suboptimal access path choices, and to be competitive with existing access paths throughout the entire selectivity range.

6.1 Experimental Setup

Software. Our adaptive operators are implemented inside PostgreSQL 9.2.1 DBMS, compiled with gcc 4.6.3 64-bit. For illustration purposes in Section 1 we use a state-of-the-art row-store DBMS we refer to as DBMS-X.

Benchmarks. We use two sets of benchmarks to showcase different algorithm characteristics: a) for stress testing we use a micro-benchmark, and b) in order to better understand the behavior of the operators in a realistic setting we use the TPC-H benchmark [33].

Hardware. All experiments are conducted on servers equipped with 2 x Intel Xeon X5660 Processors, @2.8 GHz (with L1 32KB, L2 256KB, L3 12MB caches), with 48 GB RAM, and 2 x 300 GB 15000 RPM SAS disks (RAID-0 configuration) with an average I/O transfer rate of 120 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.

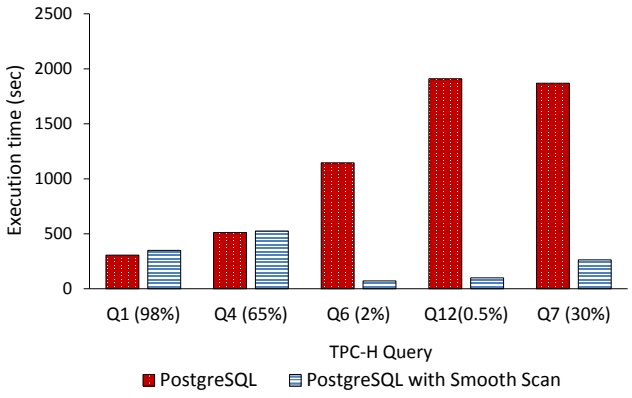


Figure 5: Improving performance of TPC-H with Smooth Scan.

6.2 Smooth Scan vs. Existing Access Paths

In this section, we demonstrate a significant benefit of Smooth Scan against the optimizer's chosen alternatives when running a few representative queries from the TPC-H benchmark (SF 10).

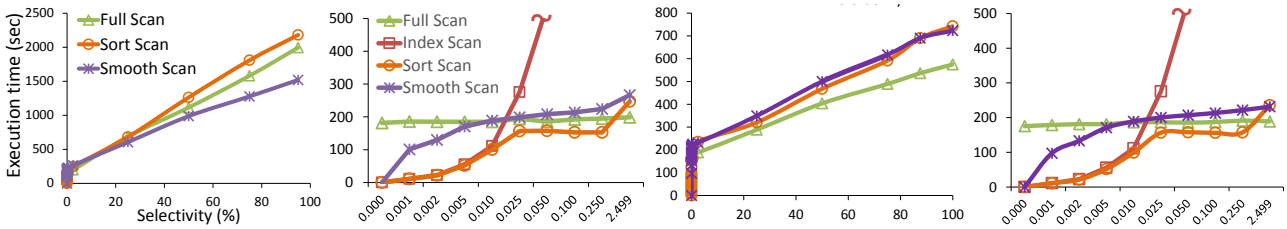
6.2.1 TPC-H Analysis

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

Improving performance of TPC-H with Smooth Scan. In this section, we demonstrate the benefit that Smooth Scan brings to PostgreSQL, illustrated when running TPC-H. Since PostgreSQL does not have a tuning tool, we create the set of indices proposed by the commercial system from the previous experiment (for the same data set). Figure 5 shows the results for 5 representative TPC-H queries. Q1 and Q6 are single table selection queries, with the selectivity at both sides of the spectrum (98% in the former, and 2% in the latter case). Q4 and Q12 are two-table join queries, again with two selectivity extremes (65% and 0.5% respectively) when considering the LINEITEM table[6]. Finally, to give a hint what happens in the case of multiple joins (since real world use case scenarios usually imply such cases), we run Q7 which is a 6-table join.

With Smooth Scan, PostgreSQL demonstrates robust performance. It does not suffer from extreme degradation and achieves good performance for all queries. For instance, while plain PostgreSQL

[6]The performance greatly depends on the selectivity of this table, since it is the largest.



(a) With order by (b) With order by (sel. 0-2.5%) (c) W/o order by (d) W/o order by (sel. 0-2.5%)

Figure 6: Smooth Scan vs. Alternative Access Paths w/o Order By.

suffers in $Q6$ due to a suboptimal choice of an index scan, PostgreSQL with Smooth Scan maintains good performance avoiding a degradation by a factor of 16. $Q6$ selects 2% of the data, which in the case of the index scan causes a significant number of random accesses over the LINEITEM table, which Smooth Scan successfully flattens over time resulting in much better performance. On the other hand, in query $Q1$ with selectivity of 98% a full scan is the optimal access path and plain PostgreSQL chooses it correctly. However, even in this case Smooth Scan introduces only a marginal overhead; it quickly realizes that the full scan is the right access path and adjusts accordingly, adding only an overhead of 14% over the optimal behavior. In $Q4$, the selectivity of the LINEITEM table is 65%, and PostgreSQL chooses full scan correctly, and places it as the outer table of a nested loop join with a primary key lookup as the inner input. Although Smooth Scan starts with an index, it quickly adjusts behavior adding only 4% of overhead when compared to the optimal solution. On the contrary, the selectivity of the LINEITEM table in Q12 is around 0.5%, but still two orders of magnitude more than the optimizer estimated. Both plain PostgreSQL and our implementation start with an index scan as the outer input, joined with a INLJ with ORDERS (a primary key lookup). Smooth Scan quickly adjusts it behavior preventing a degradation by a factor of 20. Also, 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, therefore there is not need to perform additional adjustments, which Smooth Scan correctly detects. Finally, a suboptimal index choice for plan PostgreSQL over the LINEITEM table for a 6-way join in Q7, degraded performance by a factor of 7; which Smooth Scan successfully prevented.

Memory requirements. Our memory structures span a couple of MB in these experiments. For illustration, Q1 has the highest selectivity touching 59M tuples, with the original estimate of roughly 20M. Therefore, if we decided to track all 20M tuples (employing the Optimizer Driven Policy), our Tuple ID cache will occupy 2.5MB. The Page ID cache is the only mandatory structure that will be employed regardless of the policy choice. In this experiment, the LINEITEM Page ID cache occupies 140KB (for 1M pages).

Overall, Smooth Scan gives PostgreSQL a robust behavior without requiring accurate statistics, bringing significant gains in cases where the original system makes a wrong decision and only marginal overheads when a correct decision can be made.

6.2.2 *Fine-grained Analysis over Selectivity Range*

We now use a micro-benchmark to stress test the various access paths. We compare Smooth Scan against Full Scan, Index Scan and Sort Scan[7] to demonstrate the robust behavior of Smooth Scan. All experiments are on top of our extension of PostgreSQL, thus Full Scan, Index Scan and Sort Scan are the original PostgreSQL access paths. The micro-benchmark consists of a table with 10 integer columns randomly populated with values from an interval $0 - 10^5$. The first column is the primary key identifier, and is equal to a tuple order number. The table contains 400M tuples, and occupies 25GB of disk space (3M pages[8]). In addition to the primary key, a non-clustered index is created on the second column ($c2$). For this experiment we run the following query:

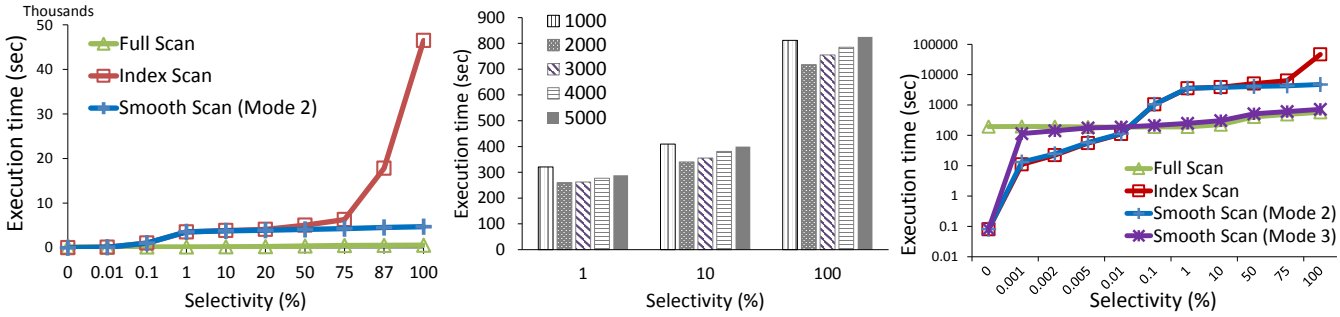
```
Q1: select * from relation where c2 >= 0 and c2< X%
  [order by c2 ASC];
```

Supporting an interesting order. In this experiment we show that Smooth Scan serves its purpose of being an index access path; it maintains tuple ordering and hence significantly outperforms other alternatives for queries (or sub-plans) that require the ordering of tuples. Figure 6a and 6b show such a use-case. The graphs show the performance of all alternative access paths for a query with an order by statement. For readability, we plot the operators separately over the entire selectivity range (in Figure 6a) and for selectivity between 0 and 2.5% (in Figure 6b). The performance of Index Scan degrades quickly due to repeated and random I/O accesses. For selectivity 0.1% its execution time is already 10 times higher than Full Scan, reaching a factor of 80 for 100% selectivity. Therefore, we omit Index Scan from Figure 6a. Sort Scan solves the problem of repeated and random accesses, while at the same time fetching only the heap pages that contain results; therefore, it is the best alternative for selectivity below 1%. Nonetheless, its sorting overhead grows and for selectivity above 2.5% it is not beneficial anymore. Smooth Scan is between the alternatives when selectivity is below 2.5%, while it actually achieves the best performance for the selectivity above this level. This is due to the overhead of posterior sorting of tuples to produce results that respect the interesting order (order by) from which both Full Scan and Index Scan suffer.

Without an interesting order. When tuple ordering does not have to be maintained, the index does not actually need to be used (unless the query is highly selective); other access paths such as Full Scan or Sort Scan can help more. This is illustrated in Figure 6c and Figure 6d; The figures show the performance of the operators when executing $Q1$ but this time without the order by clause. For selectivity between 0 and 2.5% the behavior of the operators is the same as with the case when ordering is imposed. Nonetheless, for higher selectivity Full Scan is now the best alternative, since it performs a pure sequential access. This corroborates the common wisdom that Full Scan should be used when selectivity is high. 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 almost sequential (page IDs are always monotonically increasing). Smooth Scan does not suffer from the sorting

[7]Sort Scan or Bitmap Scan in PostgreSQL implementation differs a bit from the high level description. PostgreSQL sorts only page IDs, while tuples per page are stored in separate bitmap structures; the sorting procedure is thus two-step, filling in the bitmap structures and sorting the page IDs prior to page fetching.

[8]Every page has a size of 8KB (PostgreSQL's default value).



(a) Impact of Mode 2 (b) Maximum Morphing Expansion Size (c) Impact of Mode 3

Figure 7: Sensitivity Analysis of Smooth Scan Modes.

overhead, but it does suffer from a periodical random I/O accesses driven by the index probes, adding 20% overhead when compared to Full Scan for 100% selectivity.

Marrying alternatives with Smooth Scan. Smooth Scan bridges the gap between existing access paths. Its performance does not degrade severely when selectivity increases, like in the case of Index Scan. At the same time, it avoids the sorting overhead of Sort Scan, and also it does not pay a huge cost to select just a few tuples which is the case with Full Scan. This is especially important for point queries for which Full Scan is practically not applicable, while Index Scan could degrade quickly if the dataset happen to be skewed (or big). In addition, by respecting tuple ordering driven by the index it adheres to the pipelining model. Most importantly, Smooth Scan is statistics-oblivious and it can provide near-optimal performance in the case of severe cardinality misestimation.

6.3 Sensitivity analysis of Smooth Scan

In this section, we study in detail low level parameters that affect the performance of Smooth Scan such as the impact of its morphing modes, policies, and its ability to handle skew. Finally, we show that our analytical model matches the real performance and we also study the Smooth Scan effect on hard disks versus SSDs.

Impact of the Entire Page Probe Mode. In its leaf pages, a non-clustered index keeps pointers to tuples stored in the heap.Thus, each index access (unless completely covering) is followed by a tuple look-up. This pointer chasing in general hurts performance, and depending on the layout of pages on the heap, could cause up to the number of tuples per page repeated accesses of the same page.

Figure 7a depicts the improvement that Smooth Scan achieves by removing repeated accesses when executing query $Q1$ from the micro-benchmark. Here Smooth Scan morphs only to Mode 2 (Entire Page Probe). Smooth Scan improves by a factor of 10 when compared to Index Scan for selectivity 100%. The performance of Smooth Scan degrades only slightly with selectivity and only up to 1%; this is the point where approximately all pages have been read[9]. After that point the execution time stays nearly flat with the increase of 30% 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.

Nonetheless, the execution time of Smooth Scan when morphing only up to Mode 2, is still significantly higher (8 times) compared to Full Scan for 100% selectivity. This is due to the discrepancy between random and sequential page accesses; the former being at least an order of magnitude slower in the case of disk storage.

[9]With 120 tuples per page (64-byte tuples stored in a 8KB page) and uniform distribution, we may expect one tuple from each page to qualify for selectivity 1%.

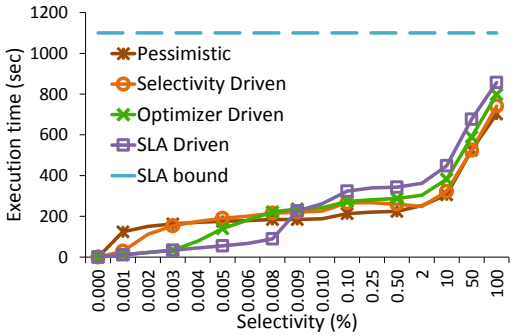
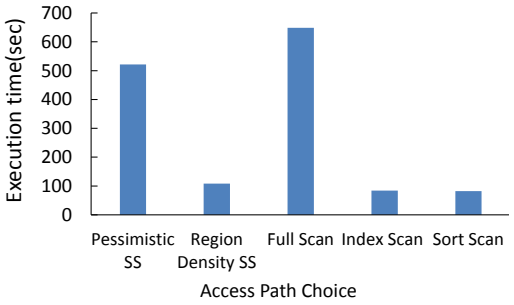


Figure 8: Impact of Policy Choices.

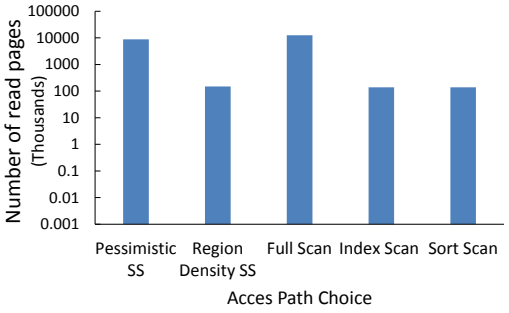
Impact of the Flattening Access Mode. To alleviate the random access problem in a statistics-oblivious way, Smooth Scan morphs to Mode 3(+). In this phase, Smooth Scan starts fetching an increasing number of data pages for every index probe. By fetching adjacent pages it amortizes access costs at the expense of extra CPU cost to go through all the fetched data. Figure 7b shows a sensitivity analysis of the maximum number of adjacent pages up to which we perform the expansion. In this experiment we vary the maximum number of adjacent pages from 1K up to 5K pages. Our experiments show a factor of 2K pages as optimal (translates to the block size of 16MB); thus we keep this value as the maximum expansion step in the rest of the experiments.

Figure 7c shows the results for the same set-up as in the previous experiment (Figure 7a); now, Smooth Scan is set with a maximum expansion step of 2K pages and with the Pessimistic Policy. This time Smooth Scan is not only much better than Index Scan (factor 65) but also nearly approaches the behavior of Full Scan in the worst case of selectivity 100% (it is only 20% slower). Morphing to Mode 3 thus provides even more robust performance.

Impact of Policy Choices. To better understand the trade-offs lying behind different policy choices we plot them in Figure 8. The (Fully) Pessimistic policy in this example starts in Mode 3(+) immediately, therefore it converges to Full Scan the fastest. The Optimizer Driven policy starts in Mode 1 and changes to Mode 2 after 15K tuples, which is the optimizer estimated cardinality, causing the increase in the execution time for selectivity 0.003%. After the shift to Mode 2, the Optimizer Driven policy uses $2 * 10^{-5}$ % which translates to 100 tuples as the selectivity threshold. One could observe that the overhead of the Optimizer Driven policy increases for higher selectivity compared to the Pessimistic policy. The overhead in this case is attributed to a tuple check for each tuple produced in Modes 2 and 3. Similar behavior is observed with the SLA driven policy, with a bit sharper cliff for point 0.009%, since with this policy we switch immediately to Mode 3(+). For this experiment



(a) Execution Time (1% selectivity)



(b) Number of Read Pages (1% selectivity)

Figure 9: Handling Skew.

we have set an upper performance bound equal to the performance of 2 full scans as the SLA constraint. According to the model the switching point between Modes 1 and 3 is 32K tuples. For the Selectivity Driven policy we set a selectivity threshold of 10^{-4}%, equal to 400 tuples. For lower selectivity this policy introduces less overhead compared to the pessimistic one since it fetches fewer adjacent pages, i.e., more pages need to be seen before we morph.

Although there is no clear winner between the policies, they are nicely clustered depending on the overall target in terms of performance. The Selectivity Driven policy strikes a nice balance in terms of overall performance. However, if we are in an environment where respecting SLAs is the main priority then an SLA driven or a Pessimistic policy can be viable alternatives.

Adjusting to Skew Distributions. In previous sections, Smooth Scan has demonstrated the ability to prevent execution time blow-up, due to the increased number of qualifying tuples in the result. We now show that Smooth Scan can adapt very well even to the skewed data distributions. For this experiment, we use the Region Density policy and compare it against the Pessimistic policy. For completeness, we also report numbers for the traditional operators.

We use a table with 1.5B tuples, 10 integer columns (randomly populated from the interval [0-10^5]) that occupy 100GB of disk space. Similarly to the previous micro-benchmarks, the first column is a primary key, and we create a secondary index on the second column ($c2$). The skew is created as follows. First 15M tuples have $c2$ value set to 0; afterwards another 0.001% of random tuples are set to 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. The query we execute is:

```
Q2: select * from relation where c2 = 0;
```

Figure 9a plots the execution time of Index Scan, Sort Scan, Full Scan, Pessimistic Smooth Scan and Smooth Scan with the Region Density morphing applied (Region Density Smooth Scan); Figure 9b plots the number of distinct pages fetched to answer the query. From Figure 9b we can see that Pessimistic Smooth Scan fetches 58 times more pages than Region Density Smooth Scan, and it is 5 times slower. The large number of pages is due to the initial skew; Pessimistic Smooth Scan notices the high selectivity increase at the beginning, and in order to reduce the potential degradation it continues fetching big chunks of sequentially placed page, ultimately fetching 8.8M out of 12.5M pages. On the contrary, after the dense region, Region Density Smooth Scan decreases the morphing step, quickly converging back to the classical access of only a single page per probe, ultimately ending up with only 150K pages fetched, in contrast with Index Scan[10] and Sort Scan that fetch 140K pages. Therefore, it continues providing near-optimal performance, despite the significant initial skew.

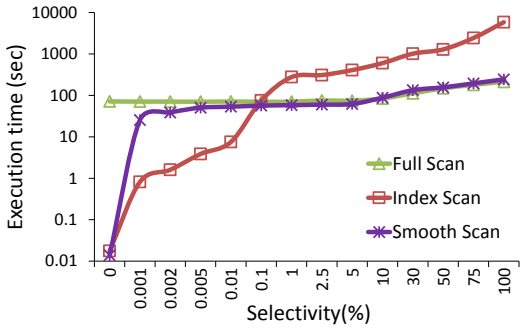


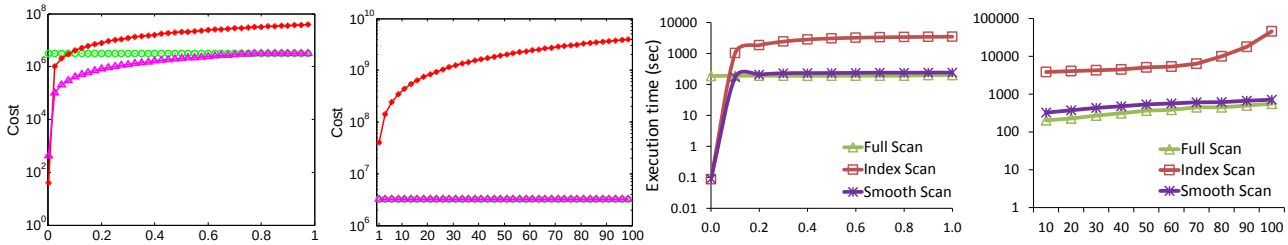
Figure 10: Smooth Scan on SSD.

SSD Analysis. Next, we test the behavior of Smooth Scan in solid state disks (SSD). Given the different access costs and better random access performance it is interesting to see how behavior changes when we replace disk with SSDs. 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). For the experiment we use query $Q1$ from the micro-benchmark; we compare Smooth Scan against Index Scan and Full Scan.

Figure 10 demonstrates that Smooth Scan benefits even more from solid state technology than with hard disks. The overall behavior of the three access paths is similar in both cases; when ran on HDD (Figure 7c) and when ran on SSD (Figure 10). Nonetheless, there are subtle differences. Solid state devices are well known for removing mechanical limitations of disks, which enables them to achieve better performance of random I/O accesses. Our analysis for the hardware used in this paper, shows that random I/O accesses are two times slower than sequential accesses on SSD, while this discrepancy reaches a factor of 10 in the case of a hard disk. This difference makes Index Scan (and consequently 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 the flattening pattern compared with other alternatives. As a result, Index Scan is slower than other alternatives by a factor of 25 for 100% selectivity. Smooth Scan favors SSD over HDD, since occasional random jumps when following pointers from the index do not hurt performance much. As a consequence, Smooth Scan is actually faster than Full Scan for selectivity below 20%, and is only 10% slower compared with Full Scan for 100% selectivity.

Cost Model Analysis. The cost model of smooth paths allows us to predict the performance of different policy choices or to trigger mode shifts to meet SLA requirements. In this experiment we show that the estimates of the analytical model we derived are corroborated with the actually measured performance. Figure 11a and Figure 11b show the cost behavior of Full Scan (green circles), Index

[10]The severe impact of repeated and random I/O is not seen in this experiment, since the index key follows the page placement on disk.



(a) Model: Selectivity 0-1% (b) Model: Selectivity 1-100% (c) Measure: Selectivity 0-1% (d) Measure: Selectivity 1-100%

Figure 11: Comparing the Analytical Model with Execution.

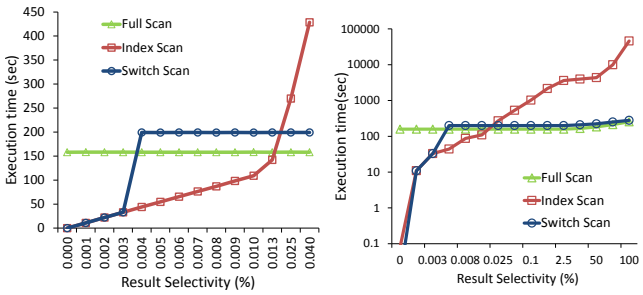
Scan (red stars), and Smooth Scan (purple triangles) based on the analytical cost model, as a function of selectivity. The y-axis shows the cost (unit 1 corresponds to one sequential I/O). We model the costs for a table with 400M tuples from the micro-benchmarks. For the page size we take the value of 8KB; for the tuple size we assume 64 bytes (40 bytes of data plus the overhead for the tuple header), and for the key size we use 16 bytes. We assume uniform distribution of result tuples (worst case), and approximate the number of random I/O accesses for Mode 3 with $log_2(\#P+1)$, where P is the number of pages in total. Finally, for seq_{cost} we use 1, for $rand_{cost}$ we use 10, and for cpu_{cost} we use $1/1M$. We separately show the behavior of the operators when selectivity is between 0 and 1%, since for the increasing selectivity both Full Scan and Smooth Scan converge to the same value and hence overlap on the graph.

The model suggests that for lower selectivity Smooth Scan behaves like Index Scan, while for higher selectivity it converges to the performance of Full Scan. This is corroborated in our experiments presented in Figure 11c and Figure 11d; they depict the real execution times using the actual data that the model assumed. In both graphs Smooth Scan converges to Full Scan as predicted. The only discrepancy from the model we observe is that Smooth Scan converges faster to Full Scan than estimated. This effect is due to the disk controller behavior, grouping many sequential I/O requests from the disk controller queue into one in the case of Full Scan, which puts the performance bar of Full Scan a bit lower than expected. Similar behavior is not observed in the case of Smooth Scan that issues requests for sequential sub-arrays with random jumps in between. Although the same grouping of sequential sub-arrays could happen and equally improve performance, the disk controller did not possess logic to do so. With the selectivity increase and the morphing expansion this affect vanishes since the sequential sub-arrays are bigger, meaning that Smooth Scan is able to fill in the queue with pure sequential requests as well.

6.4 Switch Scan: A Straw Man Adaptivity

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

Figure 12 shows the analysis of the benefit and overhead of Switch Scan when executing query $Q1$ from the micro-benchmark, as a function of selectivity. In Figure 12a, one can observe a performance cliff for 0.004% selectivity, due to the strategy switch. In this example, the optimizer has estimated cardinality of the result to be 15K tuples (out of 400M) and therefore decided to employ an index scan. While monitoring the actual cardinality of this predicate, we observe that we have more than 15K tuples and perform the switch before we produce result tuple 15001. The execution



(a) Switch point (b) Switch Scan benefit

Figure 12: Switch Scan Performance Cliff and Overall Benefit.

time to produce 15001 tuples now becomes the execution time of the index seek to produce 15K 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% (Figure 12b). Nonetheless, the moment we opt for the switch we increase the execution time equal to the time of the full scan, which might not be amortized over the rest of the query lifetime.

7. RELATED WORK

Smooth Scan draws inspiration from a large body of work on adaptive and robust query processing. We briefly discuss the work more related to our approach, while for a detailed summary the interested reader may refer to [10].

Statistics Collection. Since the quality of plans directly depends on the accuracy of data statistics, a plethora of work has studied techniques to improve the statistics accuracy in DBMS. Modern approaches employ the idea of monitoring execution to exploit this information in future query compilations [1, 31, 5]. In dynamically changing environments, however, statistical information rarely stays unchanged between executions; consider data ingest different devices produce (e.g. smart meters [19], data from Facebook, etc.) for instance. Orthogonal techniques focused on modeling the uncertainty about the estimates durign query optimization [3, 4]. Overall, considering the two-dimensional change in the workload characteristics (frequent data ingest, and ad-hoc queries) in modern applications, and the high price of having up-to-date statistics for all cases in the exponential search space [5, 6], the risk of having incomplete or stale statistics still remains high.

Single-plan Adaptive Approaches. From the early prototypes to most modern database systems, query optimizers determine a single best execution plan for a given query [29]. To cope with environment changes in such systems, some of the early work on adaptive query processing employed reoptimization techniques in the middle of query execution [26, 28, 24]. Since this re-optimization step can introduce overheads in query execution, an alternative tech-

nique proposed in the literature is to choose a set of plans at compile time and then opt for a specific plan based on the actual values of parameters at run-time [17, 23]. A middle ground between reoptimization and dynamic evaluation is proposed in [4], where a subset of more robust plans is chosen for given cardinality boundaries. Regardless of the strategy when to adjust behavior, reoptimization approaches suffer from similar binary decisions that we have seen with Switch Scan; once reoptimization is employed, the strategy switch will almost certainly trigger a performance cliff.

Multi-plan Adaptive Approaches. Single-plan approaches in query processing are usually not a viable solution in the case of data streaming environments. Some of the early techniques with multi-plan approaches employed competition to decide between alternative plans [2, 18]. Essentially, multiple access paths for a given table are executed simultaneously for a short time and the one that wins is used for the rest of the query plan. This approach could be considered as a very first that employed intra-operator adaptivity. In contrast, Smooth Scan does not perform any work that is thrown away later (while all the work done for every access method except the winning one is discarded in the approach of competing plans).

Adaptive and Robust Operators. With workloads being less steady and predictable, coarse-grained index tuning techniques are becoming less useful with the optimal physical design being a moving target. In such environments, adaptive indexing techniques emerged, with index tuning being a continuous effort instead of a one time procedure. Partial indexing [30, 32, 36] broke the paradigm of building indices on a full data set, by partitioning data into interesting and uninteresting tuples, while indexing only the former. Similarly, but more adaptively using the workload as a driving force, database cracking and adaptive merging techniques [20, 15, 21] lower the creation cost of indices and distribute it over time by piggybacking on queries to refine indices. Lastly, SMIX indices are introduced as a way to combine index accesses with full table scans, by building covered values trees(CVT) on tuples of interest [34]. Despite bringing adaptivity in index tuning, none of the techniques addresses the index accesses from the aspect of query processing, and hence stayed susceptible to the optimizer's mistakes.

The closest to our motivation of achieving robustness in query processing is G-join [13], an operator that combines strong points of join alternatives into one join operator. Orthogonal techniques have similarly been proposed to prevent the degradation of INLJ [25, 11]. We, however, consider access path operators and morph from one operator alternative to another as knowledge about data evolves, while other operators benefit consequently.

8. CONCLUSION

With the increase in complexity of modern workloads and the technology shift towards cloud environments, robustness in query processing is gaining its momentum. Still current systems stay sensitive to the quality of statistics. As a result, the runtime execution of queries may fluctuate severely as a result of marginal changes in the underlying data. When considering robust performance, stable and expected or close to expected performance has to be provided for every query, even with stale or non-existent statistics.

This paper introduces a new class of *morphable* operators that are statistics-oblivious. In particular, we present Smooth Scan, an operator that continuously and adaptively morphs between the two access path extremes: from an index look-up into a full table scan. As Smooth Scan sees more data during query execution, it understands more about the properties of the data and morphs its behavior to fit 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 throughout the entire selectivity interval by being either competitive or significantly outperforming other access path alternatives.

9. REFERENCES

[1] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *SIGMOD*, 1999.

[2] G. Antoshenkov and M. Ziauddin. Query processing and optimization in oracle rdb. *PVLDB*, 5(4):229–237, 1996.

[3] B. Babcock and S. Chaudhuri. Towards a robust query optimizer: a principled and practical approach. In *SIGMOD*, 2005.

[4] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization. In *SIGMOD*, 2005.

[5] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. A pay-as-you-go framework for query execution feedback. *PVLDB*, 1(1), 2008.

[6] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Exact cardinality query optimization for optimizer testing. *PVLDB*, 2(1), 2009.

[7] S. Christodoulakis. Implications of certain assumptions in database performance evauation. *TODS*, 9(2):163–186, 1984.

[8] H. D, P. N. Darera, and J. R. Haritsa. On the production of anorexic plan diagrams. In *VLDB*, 2007.

[9] H. D., P. N. Darera, and J. R. Haritsa. Identifying robust plans through plan diagram reduction. *PVLDB*, 1(1):1124–1140, 2008.

[10] A. Deshpande, I. Zachary, and V. Raman. Adaptive Query Processing. In *Foundations and Trends in Databases*, 2007.

[11] M. Elhemali, C. A. Galindo-Legaria, T. Grabs, and M. M. Joshi. Execution Strategies for SQL Subqueries. In *SIGMOD*, 2007.

[12] G. Graefe. Modern b-tree techniques. *Found. Trends databases*, 3(4):203–402, 2011.

[13] G. Graefe. New algorithms for join and grouping operations. *Comput. Sci.*, 27(1):3–27, 2012.

[14] G. Graefe, A. C. König, H. A. Kuno, V. Markl, and K.-U. Sattler. 10381 Summary and Abstracts Collection – Robust Query Processing. In *Robust Query Processing*, 2011.

[15] G. Graefe and H. Kuno. Adaptive indexing for relational keys. *ICDEW*, 0:69–74, 2010.

[16] G. Graefe, H. A. Kuno, and J. L. Wiener. Visualizing the robustness of query execution. In *CIDR*, 2009.

[17] G. Graefe and K. Ward. Dynamic query evaluation plans. In *SIGMOD*, 1989.

[18] J. M. Hellerstein, M. J. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah. Adaptive Query Processing: Technology in Evolution. *IEEE Data Engineering Bulletin*, 23:2000, 2000.

[19] IBM. Managing big data for smart grids and smart meters. White Paper, http://goo.gl/n1Ijtd, 2012.

[20] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *CIDR*, 2007.

[21] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. *PVLDB*, 4:586–597, 2011.

[22] Y. E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.

[23] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric query optimization. *PVLDB*, 6(2):132–151, 1997.

[24] Z. G. Ives, A. Y. Halevy, and D. S. Weld. Adapting to source properties in processing data integration queries. In *SIGMOD*, 2004.

[25] S. Iyengar, S. Sudarshan, S. K. 0002, and R. Agrawal. Exploiting Asynchronous IO using the Asynchronous Iterator Model. In *COMAD*, 2008.

[26] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, 1998.

[27] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for local queries. In *SIGMOD*, 1986.

[28] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust Query Processing through Progressive Optimization. In *SIGMOD*, 2004.

[29] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[30] P. Seshadri and A. N. Swami. Generalized partial indexes. In *ICDE*, 1995.

[31] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. Leo - db2's learning optimizer. In *VLDB*, 2001.

[32] M. Stonebraker. The case for partial indexes. *SIGMOD Record*, 18:4–11, 1989.

[33] TPC. Tpc-h benchmark. `http://www.tpc.org/tpch/`.

[34] H. Voigt, T. Kissinger, and W. Lehner. Smix: self-managing indexes for dynamic workloads. In *SSDBM*, 2013.

[35] A. N. Wilschut and P. M. G. Apers. Dataflow Query Execution in a Parallel Main-memory Environment. In *PDIS*, 1991.

[36] E. Wu and S. Madden. Partitioning techniques for fine-grained indexing. In *ICDE*, 2011.