Detecting and Exploiting Near-Sortedness for Efficient Relational Query Evaluation

Sagi Ben-Moshe
Technion
sagib@cs.technion.ac.il

Eldar Fischer
Technion
eldar@cs.technion.ac.il

Mani Fischer
HP Labs
mani.fischer@hp.com

Yaron Kanza
Technion
kanza@cs.technion.ac.il

Arie Matsliah
Technion
arie.matsliah@gmail.com

Carl Staelin
HP Labs
carl.staelin@hp.com

ABSTRACT

Many relational operations are best performed when the relations are stored sorted over the relevant attributes (e.g. the common attributes in a natural join operation). However, generally relations are not stored sorted because it is expensive to maintain them this way (and impossible whenever there is more than one relevant sort key). Still, many times relations turn out to be *nearly-sorted*, where most tuples are close to their place in the order. This state can result from "leftover sortedness", where originally sorted relations were updated, or were combined into interim results when evaluating a complex query. It can also result from weak correlations between attribute values. Currently, nearly-sorted relations are treated the same as unsorted relations, and when relational operations are evaluated for them, a generic algorithm is used. Yet, many operations can be computed more efficiently by an algorithm that exploits this near-ordering.

However, to consistently benefit from using such algorithms the system should also refrain from using the wrong algorithm for relations which happen not to be sorted at all. Thus, an efficient *test* is required, i.e., a very fast approximation algorithm for establishing whether a given relation is sufficiently nearly-sorted.

In this paper, we provide the theoretical foundations for improving query evaluation over possibly nearly-sorted relations. First we formally define what it means for a relation to be nearly-sorted, and show how operations over such relations, such as natural join, set operations and sorting, can be executed significantly more efficiently using an algorithm that we provide. If a relation is nearly-sorted enough, then it can be sorted using two sequential reads of the relation, and writing no intermediate data to disk. We then construct efficient probabilistic tests for approximating the degree of the near-sortedness of a relation without having to read an entire file. The role of our algorithms in a database management system setting is illustrated as soon as the theoretical foundation is laid out.

Finally, we outline factors that relate to practical implementations of our algorithms. We show how our test can be enhanced to provide an approximation rather than just a yes-no answer, and discuss its implementability in real-life scenarios where some sparseness may be present in the database files (e.g. if they were created using a B*-tree approach). We also show how our sort can benefit distributed systems and systems that use a solid-state drive, which may very well become prevalent in the near future.

Categories and Subject Descriptors

E.5 [**Files**]: [Sorting/searching]; H.2.4 [**Database Management**]: Systems—*Query processing, Relational databases*

General Terms

Relational databases, Query processing, Property testing, Sorting, Relational operators, Algorithms, Solid State Drives

1. INTRODUCTION

Typically, database query processors handle relations that are stored unsorted on the disk. When a processor needs to access the tuples of a relation in some specific order, this can be done through an appropriate index, if such an index exists, or by initially sorting the relation and then retrieving the tuples. Otherwise the system would have been required to constantly maintain the relations sorted, and this would cause updates and insertions to be inefficient. Additionally, if the setting is such that more than one order is relevant (*e.g.* if there is more than one index) then of course it is impossible to maintain the relation sorted for all of them.

However, when querying the data, there are operations whose evaluation is far more efficient if the relations are sorted. We will refer to such operations as *order-preferring operations*. For example, the natural join of two relations that are sorted on their joint attributes can be done in a single pass over the relations, *i.e.*, a single sequential read of the files (assuming that every set of tuples with common values in the join attributes can fit into computer memory), whereas other join methods, such as nested-loop join, require more than one pass over at least one of the relations [5]. Additional order-preferring operations include the set operations (UNION, INTERSECT and EXCEPT) and operations whose im-

ICDT 2010, March 21–24, 2011, Uppsala, Sweden.

plementation has to use sorting, *e.g.,* grouping, duplicate elimination (DISTINCT) and calculating aggregate functions. An index could help to evaluate some order-preferring operations, but it does not always exist for the attributes that we need, and creating and maintaining an index is costly.

A simple approach to executing order-preferring operations is to first sort the relations and then apply the operation. However, the sorting operation itself is expensive and may require several passes over the relations. In particular, we will show that this approach is not optimal for the *nearly-sorted* relations that we will consider here.

We define nearly-sorted relations with respect to two parameters k and ℓ. The parameter ℓ tells us how rough is the sorting: a larger ℓ means that tuples can be further away from their in-place position, before they are considered to be out of place. The parameter k tells us how many tuples are completely out of place, that is, how many tuples need to be disregarded before the remaining relation conforms with the ℓ parameter.

Nearly-sorted relations occur in various scenarios. Some examples are *(1)* a relation that has been stored as sorted and has been slightly updated (a small number of arbitrary updates are captured by k, and certain systemic updates are captured by ℓ); *(2)* a relation that has been created by a previous operation (such as a join) performed on sorted relations; *(3)* a relation that was sorted on one attribute (or on a set of attributes) can be nearly-sorted on another attribute due to naturally occurring correlations (*e.g.,* weight to height, apartment size to rental fee and so on).

If the given relation is nearly-sorted then the query processor can execute order-preferring operations very efficiently. Specifically, in Section 4 we show an algorithm that sorts nearly-sorted relations in at most two sequential reading passes. The algorithm requires memory of size roughly $t(2k+\ell)$, where t is the maximal size of a tuple and k, ℓ are the near-sortedness parameters defined above (we also provide a simpler, less efficient algorithm as a warm-up). In contrast, traditional methods (without prior conditions on the relations) require more than just two sequential passes over the relation, even just for sorting (multiway merge sort can usually be done in two "passes", but one of them requires fully random access, which is much more expensive than a sequential pass). Furthermore, our more improved algorithm (Algorithm 2) makes no writes at all to disk apart from writing the output itself, a vast improvement over all previous algorithms (including multiway merge). The reason that we focus first on the sorting operation itself is not only because it is the easiest to explain among database operations, but also because it contains the core algorithm. In Section 5 we explain how to incorporate this algorithm into the execution of other operations, such as natural join (equijoin) and set operations (intersection, etc). The following example illustrates the savings.

EXAMPLE 1. *Consider the natural join of two (large) relations, R_1 and R_2, that are nearly sorted on a single shared attribute, in the sense that at most k tuples of each relation violate the order of the tuples according to this attribute. A naive join of the relations will be to sort each relation and then to apply a sort-based join. For each relation, sorting it will require at least two phases of reading all the tuples of the relation and writing all the tuples to the disk (and possibly more phases if the reads have to be sequential). After sorting, another pass over the two relations will be needed for the join itself (in some cases the last pass of the sorting algorithms can be merged with the join operation).*

However, knowing that the relations are nearly sorted allows to compute the join by merely two sequential readings of the relations and without writing temporary results to disk.

Note that for relations that are much larger than computer memory, a nested-loop join (the sort-less option) will require more than two sequential readings of the relations. Similarly, ad-hoc creation of indexes for the join will be more expensive than two sequential reads of the relations, and in many cases using the indexes will not be efficient either way.

In many cases, applying order-preferring operations on nearly-sorted relations can be done even when the number of tuples which violate the order is quite large.

EXAMPLE 2. *If in Example 1 both R_1 and R_2 have a size of 10 gigabyte and we use a memory of 2 gigabyte for the join, a computation of a join as described in Example 1 can be done even when approximately 10% of the tuples in each relation violate the order.*

When in Example 1 the relations R_1 and R_2 are *not* nearly sorted, the described algorithm will fail due to lack of memory for keeping the order-violating tuples. In such a case, the system would need to use a generic join algorithm, *i.e.,* an algorithm that can join unsorted relations. Actually our algorithm can recover from a failing state and fall back to a less efficient one, but it is still best to efficiently test whether the given relations are nearly sorted and choose the appropriate algorithm in advance. Moreover, in order for the approach to be efficient, this test should only read a small part of the relation.

To test that the relations are nearly sorted we use ideas from the theory of property testing. In general, property testing refers to the following type of problems: Given the ability to perform local inspections (here, reading specific tuples) of a particular object (here, a relation), the goal is to determine whether the object has a predetermined property (here, one related to being sorted), or is far from having the property. The task should be performed by inspecting only a small part of the whole object, where a small probability of failure is allowed. See [3, 10, 11] for surveys on property testing.

In our case, we also require that the computational overhead will be small, and we deal with two parameters, k and ℓ, rather than just measuring the distance from having the property (this distance would conform to our k). Most importantly, we require our tests to be *tolerant*, because we actually want to guarantee acceptance for small enough non-zero values of k and ℓ, and not only guarantee rejection for values that are too large, so as not to miss on any inputs for which our optimization is possible. Also, for practical applications the number of queries has not only to increase slowly

with the input size n, but to depend not too badly on n/k, where the best possible (which we achieve here) is a linear dependence. Additionally, the test is non-adaptive, in that it is able to provide all the locations for the reads ahead of obtaining any answers. This allows modern operating systems to optimize the reading operations and compensate for possible seek times.

Since the entire relation should be considered by the test, even though not all the tuples of the relations are being read, the choice of which tuples to read must be probabilistic. Traditionally, the probability of failure is taken to be at most $\frac{1}{3}$, but it can be made arbitrarily smaller by applying the test several times. In general, to guarantee a failure probability of at most δ, we would need to increase the number of tuples being read by an $O(\log(1/\delta))$ factor. For example, if a test requires $\sqrt{n}$ inspections where n is the number of tuples in the relation, then for a relation of 1,000,000 tuples, 1,000 reads determine the property with probability of success $\frac{2}{3}$. By applying an amplification technique, 6,000 reads provide probability of success $1-(\frac{1}{3})^6$ which is approximately 0.999.

From the property-testing point of view, the testing algorithms that we develop here generalize the tolerant monotonicity tests that were developed in [1, 9]. The algorithms of [1, 9] can only distinguish between almost sorted arrays (in the sense that removing a few elements makes them completely sorted) and those that are far from being sorted. In contrast, our testers work with a more relaxed notion of sortedness, where an unbounded number of elements can be out of order, but not too far from their correct location. As we explain in Remark 2.4 below, there is no simple relation between these monotonicity notions that would allow us to use the testers of [1, 9] as they are. The analysis of the new tests is an important contribution of this paper.

This extended abstract is organized as follows. In Section 2 we provide the framework and define near-sortedness. In Section 3 we outline the suggested strategy. In Section 4 we describe in detail the algorithm for the efficient evaluation of the sorting operation on nearly-sorted relations, where in Section 5 we describe how other operations can be performed efficiently without sorting in advance. In Section 6 we present a test for determining whether relations are nearly-sorted. This is the version that is required for the strategy outlined in Section 3, a tolerant two-parameter test that estimates what can be performed on the relations with respect to the existing memory constraints.

Finally, in Section 7 we discuss additional issues, mainly those pertaining to practical implementation. These include extensions of the test to give more information and to work for files with a bit of sparseness in them, discussion and extensions of the sort operation for more usage scenarios as well as a further discussion of recovering from an overflow, and a few words on the practical experiments (not finished yet) that we are currently conducting.

2. PRELIMINARIES

We consider a relation R as an ordered sequence of tuples $t_1, \ldots, t_n$. For relations that are stored in the database, the order is determined according to the order by which tuples are accessed in a sequential read of the relation. We also refer to R as an array – we denote by $R[i]$ the tuple t_i, and we say that t_i appears in *location* i of R. We denote by $[n]$ the set $\{1, \ldots, n\}$ of the possible indices in R, and by $[n, m]$ we denote the set $\{n, \ldots, m\}$ if $m \geq n$, or the empty set if $m < n$.

A *sort key* of R is a pair $K = (A, \leq_K)$ of an attribute A and an anti-symmetric transitive relation $\leq_K$. A sort key defines a *desired order* for the tuples of R. In the desired order, for every two tuples t_i and t_j, when $\pi_A(t_i) \leq_K \pi_A(t_j)$, the tuple t_i should appear before the tuple t_j. We generally use $R[i] \leq_K R[j]$ to denote $\pi_A(t_i) \leq_K \pi_A(t_j)$. When K is clear from the context, we simply say that the tuple t_i is lesser than or equal to the tuple t_j and denote this by $R[i] \leq R[j]$. We also denote by MIN_VAL_K a tuple with the smallest possible value (with respect to $\leq_K$) of an attribute A. Again, we may use simply MIN_VAL whenever K is clear from the context. MAX_VAL is defined similarly.

The definition of a sort key can be generalized in a natural way to the case where it consists of more than one attribute: $K = ((A_1, \ldots, A_k), (\leq_{K_1}, \ldots, \leq_{K_k}))$, where $A_1, \ldots, A_k$ are attributes in the schema of R. In such a case, the desired order of the tuples is defined using a lexicographic order.

A relation that complies with the desired order defined by the key is called *sorted*.

DEFINITION 2.1 (SORTED RELATION). *A relation R of n tuples is* sorted *according to a sort key K, if for any two indices i, j, where $1 \leq i < j \leq n$, we have $R[i] \leq_K R[j]$.*

A relation R is *k-close to being sorted* when it is possible to remove from it k tuples to achieve a sorted relation. Or alternatively, when there exists a set of at most k tuples so that the relation is fully sorted outside of it.

DEFINITION 2.2 (k-CLOSE TO SORTEDNESS). *A relation R of n tuples is* k-close to being sorted *according to a sort key K, if there exists a set of indices I, where $|I| \leq k$, so that for any two indices $1 \leq i < j \leq n$, where $i \notin I$ and $j \notin I$, we have $R[i] \leq_K R[j]$. If a relation is not k-close to being sorted, then we say that it is* k-far from being sorted.

A relation R is *ℓ-globally sorted* according to a sort key K, when for every two tuples in R that do not comply with the order defined by K, the difference between the locations of these two tuples is smaller than ℓ.

DEFINITION 2.3 (ℓ-GLOBALLY SORTED). *Given a positive integer ℓ, a relation R of n tuples is* ℓ-globally sorted *according to K, if for any two indices $i, j \in [n]$, where $i \leq j - \ell$, we have $R[i] \leq_K R[j]$.*

REMARK 2.4. *Notice that a relation is 1-globally sorted if and only if it is 0-close to being sorted (which is equivalent to being sorted). But in general, there is no correspondence between these two notions. It is an easy exercise to construct:* (1) *a relation R which is both 2-globally sorted and $n/2$-far from being sorted;* (2) *a relation R which is not even $(n-2)$-globally sorted, but is 1-close to being sorted.*

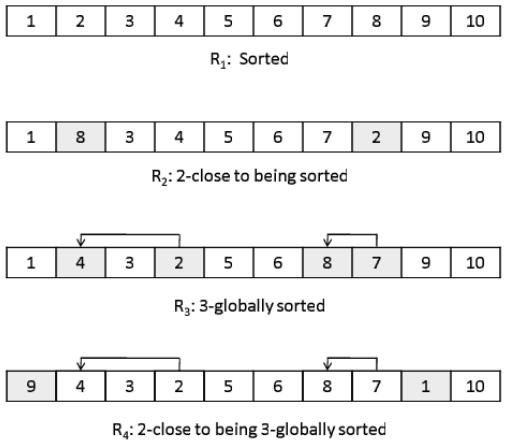

Figure 1: Sorted and nearly-sorted relations

The following final definition combines the relaxation of Definition 2.2 with the relaxation of Definition 2.3, and captures the notion of being nearly-sorted as it was discussed in the introduction.

DEFINITION 2.5 ((k,ℓ)-NEARLY SORTED). *Given a non-negative k and a positive ℓ, we say that R is* k-close to being ℓ-globally sorted *according to K, or* (k,ℓ)-nearly sorted, *if there exists a set of indices I where $|I| \leq k$, so that for any two indices $i,j \in [n]$ where $i \leq j-\ell$, $i \notin I$ and $j \notin I$ we have $R[i] \leq_K R[j]$.*

Here too, the notion of nearly sortedness is, obviously, relative to a particular attribute or sequence of attributes interpreted as the key.

If a relation is not (k,ℓ)-nearly sorted, then we say that it is *k-far from being ℓ-globally sorted*. Note that for any $k' \leq k$ and $\ell' \leq \ell$, a relation that is (k',ℓ')-nearly sorted is also (k,ℓ)-nearly sorted.

Our definitions deviate a little from the standard notion of k-closeness, which requires the tuples with indices in I to be replaced with alternative tuples for which sortedness holds for the entire relation. However, by [2, 4, 12], the two definitions are equivalent for many monotonicity-like properties, including all those defined here.

LEMMA 2.6. *Let $E \subseteq [n]$ be a set of indices, whose dismissal makes R ℓ-globally sorted. There is a way of replacing the tuples in E with new ones (rather than dismissing them) that will make R into an ℓ-globally sorted relation.*

EXAMPLE 3. *In Figure 1 we illustrate the different definitions of nearly-sorted relations. In the sketched relations, the cells contain numbers that refer to the keys of the tuples. Relation R_1 is sorted. Relation R_2 is 2-close to being sorted, because removing 8 and 2 makes it a sorted relation. Relation R_3 is 3-globally sorted, because every two tuples that the difference between their locations is at least 3 comply with the order of the keys. Finally, the relation R_4 is $(2,3)$-nearly sorted, because outside the tuples whose keys are 9 and 1, we actually have a part of R_3.*

In the sequel we show that nearly sorted relations indeed admit an efficient algorithm for sorting or computing other operators. But as mentioned earlier, we also need a way to efficiently decide whether to apply an algorithm tailored for the nearly sorted case. For this we use a *property tester*. Rather than provide the standard definition, we provide here a specific definition for our application; we also change the terminology accordingly (*e.g.,* "probe complexity" here is what testing theory refers to as "query complexity")

DEFINITION 2.7. *Given four integers $k' \leq k$ and $\ell' \leq \ell$, a* $([k',k],[\ell',\ell])$-sortedness test *with* probe complexity q *and* error probability δ *is an algorithm making at most q read operations from the relation (all of them using random access), and satisfying the following.*

- *If the relation is (k',ℓ')-nearly sorted, then the algorithm outputs* ACCEPT *with probability at least $1-\delta$.*
- *If the relation is not even (k,ℓ)-nearly sorted, then the algorithm outputs* REJECT *with probability at least $1-\delta$.*

Note that in the above the parameters may (and will) depend on the size of the relation, n. The parameter δ will usually be held to the constant $1/3$, but making δ smaller with a logarithmic in $1/\delta$ penalty on q will turn out to be an easy amplification procedure.

The above definition in fact deviates somewhat from the standard testing definition. In the standard theory an *ϵ-test for being ℓ-globally sorted* would correspond to what we defined here as a $([0,\epsilon n],[\ell,\ell])$-sortedness test. Having a $k' > 0$ lower bound makes it relate to the stronger notion of a *tolerant test*, and having two ℓ parameters instead of just one (as we shall see below) will allow us to improve the efficiency of the test considerably, in fact making its probe complexity independent of ℓ.

3. OUTLINE OF THE SUGGESTED QUERY EVALUATION STRATEGY

As mentioned in the introduction, for some useful (small enough with respect to the computer memory) parameters k_0 and ℓ_0 a typical relation R is likely to be (k_0,ℓ_0)-nearly sorted. Based on this, our first ingredient is an efficient "correction" algorithm – Algorithm 2 below, that for any two integers k,ℓ can evaluate the sort operation on a (k,ℓ)-nearly sorted relation more efficiently than a conventional algorithm given sufficient memory. We use the sort operation itself as an example because it is relatively simple to describe, while later we explain how other operations will work by having basically the same sorting procedures plugged in. The algorithm is also capable of reporting whether the evaluation succeeded or not, so we know whether a fall-back procedure is needed.

The second ingredient in our strategy is an extremely efficient probabilistic test (Algorithm 3) that for any integers

k, ℓ can distinguish between the case where R is (k, ℓ)-nearly sorted ("Case Y") and the case where R is not even $(6k, 6\ell)$-nearly sorted ("Case N").

The exact details on the resources that these algorithms require are given in the relevant sections, but for the discussion here we should think of the probabilistic test as the cheapest algorithm, and of the correction algorithm as significantly cheaper than a conventional evaluation algorithm.

Suggested strategy:

1. First we apply the test with parameters k_0, ℓ_0.
2. If the prediction of the test is "Case Y":
   (a) We apply the correction algorithm with parameters $6k_0$ and $6\ell_0$. If the correction algorithm succeeds, then we are done.
   (b) Otherwise, if we were using Algorithm 2 then we can continue running its special fall-back mode (see Remark 4.4 below) or decide to start from scratch with a conventional algorithm, depending on how soon it failed. If we were using the simpler Algorithm 1 then the fall-back is easier – we can just continue with a conventional algorithm from where it failed.
3. If the test predicted "Case N", then we use a conventional evaluation algorithm without attempting the correction algorithm at all.

By sufficiently amplifying the success probability of our test, reaching Step 2b will be rare enough so that the average cost of using the fall-back mode of our algorithm (rather than using a conventional algorithm from the beginning) will be negligible. The fall-back overhead, while best avoided, is generally within the same order of magnitude as that of running the algorithm itself. Also, reaching Step 3 with relations that could have benefited much by our algorithm (again through an error of the testing procedure) will have a negligible average cost. For relations reaching Step 3 that in fact satisfy Case N we only have the additional cost of performing the test (when compared to a query processor that always performs a conventional algorithm); a preliminary experiment (see Section 7.7) suggests that the percentage of those would typically be small enough to have an overall average net gain by using our procedure.

Choosing the right candidate parameters k_0, ℓ_0 can be done empirically, based on past statistics, but this is not necessary. In Section 7.1 we show a probabilistic algorithm that with a slight overhead over the original tolerant tester can output a comprehensive set of candidate pairs (k_i, ℓ_i) (up-to specified precision) for which the given relation R falls under Case Y.

In settings where the relations change slowly over time we can also cache the results of the test for use in future queries, rather than test the relation every time.

4. EXPLOITING NEAR SORTEDNESS

In this section we first present Algorithm 1, which sorts (k, ℓ)-nearly sorted relations in two sequential passes. In the first pass Algorithm 1 acts similarly to the well known Replacement-Selection algorithm (see [6])[1], while it also collects a set of misplaced entries (*i.e.,* a set of at most k tuples that must be removed to make the relation ℓ-globally sorted). In the second pass, the collected tuples are distributed to their final positions. While we have a fail state in the description of Algorithm 1, we later discuss a fall-back mechanism that preserves most of the work already done.

While sometimes useful in itself, the main function of this algorithm is to serve as a warm-up for the more efficient Algorithm 2, whose analysis is based on the analysis of Algorithm 1. The main change in Algorithm 2 is that it defers all writing to the second pass, managing to do away with the writing of an intermediate file. Its fall-back mode is on the other hand less efficient than that of Algorithm 1, so a proper testing procedure for being (k, ℓ)-nearly sorted is more important there.

Algorithm 1 (Sorts a (k, ℓ)-nearly sorted relation R.)

```
create two binary heaps S, G
insert the first k + ℓ + 1 tuples (R[1], ..., R[k + ℓ + 1]) into S
i_write ← 1
for i_read = |S| + 1 to n do {first pass}
  if S = ∅ then
    FAIL
  end if
  last_written ← min{x ∈ S}
  write last_written to TMP[i_write]
  S ← (S \ {last_written})
  i_write ← i_write + 1
  if R[i_read] ≥ last_written then
    insert R[i_read] into S
  else
    insert R[i_read] into G
  end if
end for
append all tuples in S to TMP, in sorted order
i_write ← 1
for i_read = 1 to n − |G| do {second pass}
  x ← min{y ∈ G}
  if x > TMP[i_read] then
    write TMP[i_read] to OUT[i_write]
  else
    write x to OUT[i_write]
    G ← (G \ {x}) ∪ {TMP[i_read]}
  end if
  i_write ← i_write + 1
end for
append all tuples in G to OUT, in sorted order
```

THEOREM 4.1. *If R is a (k, ℓ)-nearly sorted relation, then Algorithm 1 does not reach its fail state and the result relation is strictly sorted. Furthermore, Algorithm 1 makes only two sequential passes over R, uses memory of size $O(k + \ell)$, and makes $O(n \cdot \log(\ell + k))$ computing operations.*

[1]In fact, our algorithm starts out identically to the Replacement-Selection algorithm, but the analysis given here shows that, specifically for nearly sorted relations, much stronger properties hold. The fact that empirically Replacement-Selection does well with nearly sorted relations was already mentioned in [7].

PROOF. The fact that only two passes are made as well as the bounds on memory size and the number of computing operations are clear from the description of Algorithm 1. Now we prove that the algorithm sorts any relation which is (k, ℓ)-nearly sorted.

It is easy to see that if the algorithm did not fail in the first pass, then the intermediate result (that is written in TMP) is sorted. If this is the case, then the second pass is just the standard merge between two sorted lists, and hence the resulting relation (written to OUT) will be fully sorted. So, we only need to prove that if R is (k, ℓ)-nearly sorted, then Algorithm 1 cannot fail. Observe that in every stage of the first pass $|S| + |G|$ equals $k + \ell + 1$, and therefore showing that $|G|$ never exceeds k implies that S stays nonempty, preventing the algorithm from failing.

Let $\mathcal{E}(R)$ denote the collection of subsets $E \subseteq [n]$ of at most k indices, such that for every $E \in \mathcal{E}(R)$, if we restrict R to indices $[n] \setminus E$ then we get an ℓ-globally sorted relation.

Let in addition $D = \{j \in [n] : \forall E \in \mathcal{E}(R),\ j \in E\}$ denote the set of indices that *must* be removed from R in order to make it ℓ-globally sorted by at most k removals. Observe that $|D| \leq k$. With a slight abuse of notation, let us also denote by D the set of tuples that appear at indices D of the relation R.

We claim that in every iteration of the first pass, $G \subseteq D$ and hence $|G| \leq k$. To see this, notice that whenever a tuple at index i is inserted into G, it is strictly smaller than *last_written*. Using this observation, we prove the claim $G \subseteq D$ by induction on i.

For $i \leq k+\ell+1$ the claim trivially holds (since these indices are unconditionally inserted into S). Now let $i > k + \ell + 1$. By the induction hypothesis, before treating $R[i]$ we had $G \subseteq D$. If $R[i] \geq$ *last_written* then G remains the same and we are done. Otherwise, since $R[i] <$ *last_written* there are at least $|S| = k+\ell+1-|G| \geq \ell+1$ tuples in S that are strictly larger than $R[i]$. All these tuples originally appeared before $R[i]$. Consequently, at least $|S| - \ell \geq 1$ of these tuples appeared in indices lower than $i - \ell$. Assume that $i \notin D$ and let $E \in \mathcal{E}(R)$ be such that $i \notin E$. Then all the corresponding $|S| - \ell \geq 1$ indices should be in E, because they form a violation of ℓ-global sortedness together with i. By the induction hypothesis, $G \subseteq D \subseteq E$ and hence $|E| \geq |G| + |S| - \ell \geq k + 1$, contradicting the fact that $|E| \leq k$. Hence i must be in D, concluding the proof. □

REMARK 4.2. *In an actual implementation we can (and should) replace the failure mode in Algorithm 1 with a fall-back to a traditional sorting algorithm. For example, instead of failing, the algorithm can just reset last_written and start writing another run (monotone subsequence), repeating this as many times as is necessary. In other words, we just fall-back to the Replacement Selection algorithm for creating runs that are as long as possible. Next, instead of moving to the second pass, a traditional merge-sort can be performed.*

Implementing a fall-back is necessary in part because of the probabilistic nature of the testing algorithms. In Section 7.3 below we touch upon the expected amount of overhead when the k parameter in the near-sortedness of the input is somewhat larger than available memory, bounding it when the error in the estimation of k is not too large (which will usually be the case).

Now we present a sorting algorithm which is an improvement over Algorithm 1 as it saves significantly on write operations, and in fact writes nothing to disk apart from the output itself. Apart from the clear saving in write operations in itself, not writing any intermediate result to disk makes the algorithm easier to combine with other database operations, because its output can be piped directly to the algorithm processing the next operation. The only disadvantage in the improved algorithm is that sometimes not all of the work already done upon reaching a failure mode is recoverable, so it is all the more important to use the test of Section 6 first. The analysis of the improved algorithm is based on the analysis of Algorithm 1.

THEOREM 4.3. *If R is a (k, ℓ)-nearly sorted relation, then Algorithm 2 does not reach its fail states and the resulting relation OUT is strictly sorted. Furthermore, Algorithm 2 makes only two sequential passes over R, uses memory of size $O(k + \ell)$, and makes $O(n \cdot \log(k + \ell))$ computing operations; also, it never writes any intermediate file, only the sorted output.*

PROOF. The fact that only two passes are made as well as the bounds on memory size and the number of computing operations are clear from the description of Algorithm 2. To prove that the algorithm sorts any relation which is (k, ℓ)-nearly sorted, we use the proof of Theorem 4.1.

We note that both passes of our algorithm in fact mimic the first pass of Algorithm 1, so that the analysis there still holds (in fact unless our algorithm has reached the first fail state and was then made to resume normal operation, it will never reach the second fail state, whether the input is (k, ℓ)-nearly sorted or not). The first pass of our algorithm is identical to the first pass of Algorithm 1, with the only difference being that nothing is written to disk. In the end we are left with G, which holds up to k tuples of the relation.

The second pass again generally follows the first pass of Algorithm 1, only here tuples are not inserted to G, because already in the beginning G contains all tuples that would have been passed to it. Additionally, this pass follows the writing pattern of the first pass of Algorithm 1, only here we merge the (already known in advance) content of G into the output stream. Therefore this essentially combines the writing actions of the two passes of Algorithm 1, resulting in a fully sorted output. □

REMARK 4.4. *The clear improvement in this algorithm is that no intermediate result is ever written to disk. Apart from the saving in write operations itself, it is easier to combine this algorithm with other database operations, because its output can be piped directly to the algorithm processing the next operation.*

Algorithm 2 (Improved sort for a (k, ℓ)-nearly sorted relation R.)

create two binary heaps S, G
insert the first $k + \ell + 1$ tuples $(R[1], \ldots, R[k + \ell + 1])$ into S
$last\_handled \leftarrow$ MIN_VAL
for $i_{read} = |S| + 1$ to n **do** {first pass}
  if $S = \emptyset$ **then**
    FAIL
  end if
  $last\_handled \leftarrow \min\{x \in S\}$
  $S \leftarrow (S \setminus \{last\_handled\})$
  if $R[i_{read}] \geq last\_handled$ **then**
    insert $R[i_{read}]$ into S
  else
    insert $R[i_{read}]$ into G
  end if
end for
empty S
let $G[1], \ldots, G[|G|]$ be the sorted order of G's elements
insert the first $k + \ell + 1$ tuples $(R[1], \ldots, R[k + \ell + 1])$ into S
$i_{write} \leftarrow 1$, $i_g \leftarrow 1$
$last\_handled \leftarrow$ MIN_VAL
for $i_{read} = |S| + 1$ to n **do** {second pass}
  if $S = \emptyset$ **then**
    FAIL
  end if
  $last\_handled \leftarrow \min\{x \in S\}$
  while $i_g \leq |G|$ and $G[i_g] \leq last\_handled$ **do**
    write $G[i_g]$ to $OUT[i_{write}]$
    $i_g \leftarrow i_g + 1$, $i_{write} \leftarrow i_{write} + 1$
  end while
  write $last\_handled$ to $OUT[i_{write}]$
  $i_{write} \leftarrow i_{write} + 1$
  $S \leftarrow (S \setminus \{last\_handled\})$
  if $R[i_{read}] \geq last\_handled$ **then**
    insert $R[i_{read}]$ into S
  end if
end for
append $\{G[i_g], \ldots, G[|G|]\} \cup S$ to OUT, in sorted order

In the case where algorithm reaches the fail state in the first pass, the natural instinct is to continue with the algorithm by writing the contents of G to a temporary file on the disk, clearing it, and then populating S with $k + \ell + 1$ new tuples from the input (without resetting last_handled). However, this will not work – while out of place tuples that come "too late" (i.e. well after their position according to the ordered relation) will not pose a problem here, tuples that come "too early" may set the value of last_handled so high that the rest of the relation will land in G.

There is still a fall back procedure that is more beneficial than just restarting the sort using a different algorithm when the error in our estimation of k is not too large (e.g., whenever the true k still satisfies $k = o(n)$). See Section 7.4.

5. EVALUATING OTHER OPERATORS

In this section we discuss efficient evaluation of some order-preferring operations over nearly-sorted relations. For all operations here we use at our core Algorithm 2, which is the one more suited for integration into a larger operation. For the purpose here we do not describe the fall-back procedures in the case where we reach a fail state, as these would be the expected ones. We outline the algorithms without formal details whenever these follow from standard procedures in relational query evaluation.

Intersection and other set operations. Consider the computation of an intersection of two relations, R_1 that is (k_1, ℓ_1)-nearly sorted and R_2 that is (k_2, ℓ_2)-nearly sorted, where for both relations the entire schema is the sort key in some order. Suppose that the memory is large enough to hold $2k_1 + \ell_1$ tuples of R_1, $2k_2 + \ell_2$ tuples of R_2, and buffers for the input and output.

In the first stage, we perform the first pass of Algorithm 2 on R_1 and R_2 (in some cases, for example if they are on different disks, this is best done in parallel), and obtain the corresponding heaps G_1 for R_1 and G_2 for R_2. Recall that the first pass of Algorithm 2 does not produce any output.

Then, we go in parallel over R_1 and R_2, again, and perform a procedure similar to the second pass of the sorting algorithm on each of them, but with the following change: Instead of writing the sorted output to disk, we pipe it to the algorithm that performs intersection using the merge procedure for two sorted relations. In fact, we do not keep running the second pass of the sorting algorithm over R_1 and R_2 unconditionally, but use it as an iterator – we buffer sorted subsequences of R_1 and R_2, and whenever a buffer is exhausted by the merge-intersection procedure we run more iterations of the loop in the sorting algorithm for the corresponding relation so as to fill the buffer again.

Accommodating bag (multiset) intersection is an easy extension of the above procedure, one just needs to keep track also of the original locations in the files of the tuples that are stored in G_1 and G_2. If we were using Algorithm 1 instead, we would have needed to run over both the result of the intersection from first pass and the original relations to make sure that we got the correct number of duplicate entries.

Using the above idea (with Algorithm 2) for set union, or set or bag difference, works in much the same way: running sorting algorithm instances in parallel on R_1 and R_2 and piping the output of the second stage to the corresponding merging algorithm. Bag union by itself is not an order preferring operation, but if it is part of a larger expression involving order preferring operations then it may still be better to combine it with the sorting procedure so that its output would be sorted.

Natural join (equijoin). The computation of a natural join is performed similarly to the computation of an intersection, if there are not too many tuples that agree on the value of the common attributes in the join. Suppose that at most m_1 tuples of R_1 can agree at one time on the values of the attributes common to R_1 and R_2, while at most m_2 tuples of R_2 can agree on them. In this case for the merging join algorithm during the second stage to work, we would need a memory big enough to hold a total of $(2k_1 + \ell_1)$ tuples of R_1, $(2k_2 + \ell_2)$ tuples of R_2, and additionally either m_1 tuples of R_1 or m_2 tuples of R_2 (as well as sufficient buffers). If this does not hold, then we would need to accommodate for saving and retrieving the state of the second stage sorting algorithm over (say) R_2. Then, given a large subsequence of tuples from R_2 that agree on a common attribute, we can save the state at the beginning of the sequence, and reset the algorithm to this state for every subsequence of tuples from R_1 that need to be joined with it.

6. TESTING FOR NEAR SORTEDNESS

In this section we develop a tolerant sortedness test, namely a $([k, 6k], [\ell, 6\ell])$-sortedness test, and prove its correctness. The reason for the test to be tolerant (*i.e.,* use $[k, 6k]$ rather than $[0, k]$) is so that more instances for which Algorithm 2 (or Algorithm 1) still works are accepted by the test.

Some inaccuracy in the k parameter cannot be avoided (*i.e.,* there is no test with $[k, k]$ parameters), though currently we do not know how much lower than a factor of 6 one can go. As for the ℓ parameter, there is a test that is fully accurate in ℓ, yet its number of queries depends badly on ℓ. In contrast, the test in this section has no dependency on ℓ at all.

The test presented here is non-adaptive, meaning that it can decide which tuples to read before the first reading of a tuple – only the final decision to accept or reject depends on the actual values read. This can serve to reduce the overhead further: Instead of reading the probes in the order that they are used in the test (which requires fully random-access reads), we can first decide what probes to make and then read them in the order of their positions in the file. Moreover, modern operating systems can first receive the entire list of all reads to be made and then optimize their order of execution further for the file system involved.

THEOREM 6.1. *Algorithm 3 is a $([k, 6k], [\ell, 6\ell])$-sortedness test that makes $O(\frac{n}{k} \log \frac{n}{k} \log n \log \log n)$ probes and errs with probability at most $1/3$.*

Algorithm 3 ($([k, 6k], [\ell, 6\ell])$-sortedness test)

```
a ← 0
for j = 1 to s do
   pick i_j ∈ [n] uniformly at random
   call Algorithm 4 for i_j with confidence parameter δ = 1/(6s)
   if i_j is reported to be active then
      a ← a + 1
   end if
end for
if a ≤ s·(5.5k/n) then
   return ACCEPT
else
   return REJECT
end if
```

Notice that the probe complexity is independent of ℓ. In property testing, k is usually set to ϵn for some small constant ϵ. In these terms, the probe complexity of our test is nearly logarithmic in n, which is known to be optimal even for non-tolerant simple monotonicity testing [2].

In the description of Algorithm 3 we use an undefined parameter s, the value of which will be set later in the proof of Theorem 6.1, and call Algorithm 4 that we define below. It will be clear from the proof that the success probability can be easily amplified by increasing s. First we need the following definition and lemmas.

In the following we will say that a pair (i, j) *violates* ℓ-global sortedness if $i \leq j - \ell$ and $R[i] > R[j]$.

DEFINITION 6.2. *Let (i, j) be a pair of indices with $i < j$ that violate the ℓ-global sortedness of R. We say that i is (δ, ℓ)-active with j (for $\delta > 0$) if at least a δ-fraction of the indices in $[i + 1, j]$ violate ℓ-global sortedness together with i. Similarly, we say that j is (δ, ℓ)-active with i if at least a δ-fraction of the indices in $[i, j - 1]$ violate ℓ-global sortedness together with j. We say that an index i is simply (δ, ℓ)-active if it is (δ, ℓ)-active with some index $j \in [n] \setminus \{i\}$.*

LEMMA 6.3. *Let R be a relation and let k, ℓ be two positive integers. For every $\delta \in (0, 1/2]$ let $d(\delta)$ denote the number of (δ, ℓ)-active indices in R.*

- *If R is (k, ℓ)-nearly sorted then $d(\delta) \leq k + k/\delta$, and in particular $d(1/4) \leq 5k$;*
- *If R is not $(6k, 6\ell)$-nearly sorted then $d(1/3) \geq 6k$.*

PROOF. The proof of second part of the lemma is standard in monotonicity testing: If (i, j) is a pair violating 6ℓ-global sortedness, then every k between $i+\ell$ and $j-\ell$ violates ℓ-global sortedness with either i or j. Hence, at least one of i or j violates ℓ-global sortedness with at least $\frac{j-i-2\ell}{2} \geq \frac{j-i}{3}$ indices in the interval, making it $\frac{1}{3}$-active. As the set of all $\frac{1}{3}$-active indices now intersects all pairs violating 6ℓ-global-sortedness, its size must be at least the distance $6k$.

Assuming that R is (k, ℓ)-nearly sorted, we now prove the first part of the lemma. For this we will use some methods from [1], together with additional arguments that are specific to globally sorted relations. Let $E \subseteq [n]$ be a set of at most k indices, whose dismissal makes R ℓ-globally sorted. Such a set must exist since we assumed that R is (k, ℓ)-nearly sorted. By Lemma 2.6 we can fix new values for these indices so that the resulting relation is ℓ-globally sorted. From now on let us fix a set E as above, and a sequence of new values for the corresponding tuples as per Lemma 2.6.

Next we are going to label some of the indices of R. For every $i \in E$, we label i as *high* if its tuple should be decreased (replaced by one with a lower key value under the above correction), and we label it as *low* otherwise. For each (δ, ℓ)-active index $i \in [n] \setminus E$ (we stress that i is *not* in E) we assign some index j_i that witnesses the fact that i is (δ, ℓ)-active. If $j_i > i$, then the label of i is *big*; otherwise its label is *small*. Notice that each index can have at most one label as above (every i is either high, low, big, small or has no label at all). Our aim is to bound the number of indices that are (δ, ℓ)-active, which is upper bounded by the number of labeled indices. By definition, the number of high and low indices is at most k, so it is enough to show that the number of big and small indices is at most k/δ. By letting k_{low} and k_{high} denote the number of low and high indices (respectively), we show how to bound the number of big indices by k_{low}/δ. An analogous argument works for bounding the number of small indices by k_{high}/δ.

We start by assigning weight 1 to every big index. Then, for each big index i, in decreasing order, we divide the weight of i among all the low indices h such that $i \leq h \leq j_i$ and $R(h) < R(i)$. We "spread" the weight of i in a way that maximizes the minimal weight of the receiving indices (the h's). Our goal is to show that after this process, no low index has weight more than $1/\delta$, and hence the total initial weight

of the big indices (which is exactly equal to their amount) was at most k_{low}/δ as required.

Suppose on the contrary that this is not the case, so that some low index g got weight $(1+\epsilon)/\delta$ for some $\epsilon > 0$. Let i be the first (in reverse order) big index that caused g to reach weight $(1+\epsilon)/\delta$. By definition, this event happened while the weight of i was spread among the low indices h such that $i < h \leq j_i$ and $R(h) < R(i)$, denote their number by b. From the way the weight is spread, *all* of these low indices must have weight at least $(1+\epsilon)/\delta$. Hence their total weight is at least $b((1+\epsilon)/\delta)$. By the definition of (δ,ℓ)-active indices, $b \geq \delta(j_i - i + 1)$, so their total weight is at least $(1+\epsilon)(j_i - i + 1)$. Since we iterate on the i's in decreasing order, none of these h's could gain any weight before step j_i, and therefore we should have $(1+\epsilon)(j_i - i + 1) \leq j_i - i + 1$, which is a contradiction. □

Lemma 6.4 allows us to distinguish between indices that are $(1/3,\ell)$-active and indices that are not even $(1/4,\ell)$-active.

LEMMA 6.4. *Given an index $i \in [n]$ and confidence parameter $\delta > 0$, Algorithm 4 satisfies the following:*

- *if i is $(1/3,\ell)$-active, it outputs* ACTIVE *with probability at least $1-\delta$;*
- *if i is not even $(1/4,\ell)$-active, it outputs* INACTIVE *with probability at least $1-\delta$;*
- *its probe complexity is $O(\log \frac{1}{\delta} \log n \log\log n)$.*

The constants $\alpha > 0$ and $t \in \mathbb{N}$ in the definition of Algorithm 4 are set later in the proof.

Algorithm 4 (tests if i is $(1/3,\ell)$-active or not even $(1/4,\ell)$-active)

```
for h = ⌈log_{1+α}(l+1)⌉ to ⌈log_{1+α} n⌉ do
  left ← 0, right ← 0
  for j = 1 to a = t log(1/δ) log log n do
    pick i_j ∈ [ℓ+1, (1+α)^h] uniformly at random
    if R[i] > R[i+i_j] then
      right ← right + 1
    end if
    if R[i] < R[i−i_j] then
      left ← left + 1
    end if
  end for
  if right > (2/7)a or left > (2/7)a then
    return ACTIVE
  end if
end for
return INACTIVE
```

PROOF. Assume first that i is $(1/3,\ell)$-active, and let j_i be an index that witnesses this fact, so there exist at least $\frac{|j_i - i|}{3}$ indices lying between i and j_i that violate ℓ-global sortedness with i. We assume without loss of generality that $j_i > i$. Let $h_0 \in \mathbb{N}$ be such that $(1+\alpha)^{h_0} \leq j_i - i \leq (1+\alpha)^{h_0+1}$. Then in the interval $[i, i+(1+\alpha)^{h_0+1}]$ at least a $1/3-\alpha$ fraction of the indices violate ℓ-global sortedness with i. We fix α to be small enough (say $1/100$), so that $1/3-\alpha$ is much closer to $1/3$ than to $2/7$. For large enough t (matching the parameters in Chernoff bounds), after the h_0+1'th iteration of the internal loop, with probability at least $1-\delta$ the value of *right* will be sufficiently close to $(1/3-\alpha)(1+\alpha)^{h_0+1}$. In particular the value of *right* will exceed $\frac{2}{7}(1+\alpha)^{h_0+1}$, hence the outcome will be ACTIVE as required for the first part of the lemma.

To prove the second part of the lemma, we use a similar argument. Namely, if the index i is not even $(1/4,\ell)$-active, then for all h the fraction of violating (with respect to i) indices between i and $i+(1+\alpha)^h$ (and similarly between i and $i-(1+\alpha)^h$) is at most $1/4$. But now we must make sure that no error occurred, meaning that the values of the counters *right* and *left* did not deviate too much in *any* of the $O(\log n)$ iterations of the outer loop. We can solve this problem by amplifying the success probability to $1-\Omega(1/\log n)$. This is the reason that we have the extra $\log\log n$ factor in the number of iterations of the internal loop. □

PROOF OF THEOREM 6.1. First notice that the confidence parameter δ in the executions of Algorithm 4 is set to $\frac{1}{6s}$, so that with probability at least $1-1/6$ Algorithm 4 did not err during any of the s executions.

If R is (k,ℓ)-nearly sorted, then according to Lemma 6.3 (first item), the number of $(1/4,\ell)$-active indices in R is at most $5k$. Conditioned over the event that none of the executions of Algorithm 4 err (recall that this event occurs with probability at least $5/6$) we have that the expected value of a is at most $s\frac{5k}{n}$. The probability that Algorithm 3 returns REJECT in this case is equal to the probability that the random variable a (being a sum of s independent random variables) deviates from its expectation by a multiplicative factor of 0.1. This probability can be bounded by $1/6$ by taking $s = O(n/k)$, so altogether Algorithm 3 returns ACCEPT with probability at least $2/3$ as required.

If R is $6k$-far from being 6ℓ-globally sorted, then according to Lemma 6.3 (second item), the number of $(1/3,\ell)$-active indices in R is at least $6k$. Conditioned over the event that none of the executions of Algorithm 4 err, the expected value of a is at least $s\frac{6k}{n}$. So as in the previous case, the probability that the random variable a deviates from its expectation by a multiplicative factor of 0.08 can be bounded by $1/6$, and altogether Algorithm 3 returns REJECT with probability at least $2/3$ as required. The probe complexity of Algorithm 3 is $s \cdot O(\log \frac{1}{1/s} \log n \log\log n) = O(\frac{n}{k} \log \frac{n}{k} \log n \log\log n)$. □

7. DISCUSSION

In the following we touch upon some possible extensions of our methods, and some implementation issues.

7.1 Testing for several values of k and ℓ at once

In all of the above we used algorithms that take the values of k and ℓ in advance. However, we may be interested in learning actual approximate values of k for many values of ℓ at once. First, our computer memory puts constrains only on $k+\ell$ (for Algorithm 1) or $2k+\ell$ (For Algorithm 2) and we may want to search for an optimal ℓ for which this fits our memory (every input R has for every ℓ a minimum k

for which it is (k,ℓ)-nearly sorted, the worst case being $k = n - \ell$). Second, sometimes we would like to use our sorting algorithm even if we know that it may fail (this scenario fits Algorithm 1), because it could still lead to faster sorting (see Section 7.3 and Section 7.4 below).

There is an easy extension of Algorithm 3 that allows to efficiently test for many values of k and ℓ at once. First, we construct Algorithm 5, a version of Algorithm 4 which tests whether an element is active for every possible $\ell = c^r$, where $c > 1$ is any fixed constant (that determines the number of different ℓ's we inspect) and $r = 1, 2, \ldots, \log_c n$. We restrict ourselves to powers of c so that this output would be of manageable size, and this would still give a good approximation of the optimal k and ℓ. The parameters t and α used below are the same as in Algorithm 4.

Algorithm 5 (tests for every $\ell = c^r$ if i is $(1/3,\ell)$-active or not even $(1/4,\ell)$-active)

$A[0], A[1], \ldots, A[\lceil \log_c n \rceil] \leftarrow$ INACTIVE
for $h = 1$ to $\lceil \log_{1+\alpha} n \rceil$ **do**
  $LF[0], LF[1], \ldots, LF[\lceil \log_c n \rceil] \leftarrow 0$
  $RT[0], RT[1], \ldots, RT[\lceil \log_c n \rceil] \leftarrow 0$
  for $j = 1$ to $a = 2t \log \frac{1}{\delta} \log \log n$ **do**
    pick $i_j \in [1, (1+\alpha)^h]$ uniformly at random
    if $R[i] > R[i + i_j]$ **then**
      for $r = 0$ to $\lfloor \log i_j \rfloor$ **do**
        $RT[r] \leftarrow RT[r] + 1$
      end for
    end if
    if $R[i] < R[i - i_j]$ **then**
      for $r = 0$ to $\lfloor \log i_j \rfloor$ **do**
        $LF[r] \leftarrow LF[r] + 1$
      end for
    end if
  end for
  for $r = 0$ to $\lceil \log_c n \rceil$ **do**
    if $RT[r] > \frac{2}{7}a$ or $LF[t] > \frac{2}{7}a$ **then**
      $A[r] \leftarrow$ ACTIVE
    end if
  end for
end for
return A

Now we can use Algorithm 6, a variant of Algorithm 3. It uses the same s as Algorithm 3, but here it is calculated not as a function of k (which is not provided in advance) but as a function of a desired approximation parameter $\hat{k}$, which should be set equal to a small constant fraction of the available computer memory. Here we also keep count of every possible $\ell = c^r$. In addition, instead of deciding on ACCEPT or REJECT, we just output all counters after the appropriate normalization.

Algorithm 6 (approximates $k = k(\ell)$ for every $\ell = c^r$)

$B[0], B[1], \ldots, B[\lceil \log_c n \rceil] \leftarrow 0$
for $j = 1$ to $b = s\lceil \log n \rceil = O(n \log n / \hat{k})$ **do**
  pick $i_j \in [n]$ uniformly at random
  call Algorithm 4 with index i_j and $\delta = \frac{1}{6b}$ to obtain A
  $B \leftarrow B + A$ (coordinate-wise)
end for
return $\frac{n}{b} B$ (coordinate-wise)

The probe complexity and running time of Algorithm 6 is $\widetilde{O}(n/\hat{k})$, and with probability at least $\frac{2}{3}$ it provides for every $\ell = c^r$ an approximation k such that R is $(6k + \hat{k}, 6\ell)$-nearly sorted while not being $(k - \hat{k}, \ell)$-nearly sorted. The proof is a straightforward extension of the argument given in Section 6, since having the confidence parameter $(1 - \delta)$ amplified to $(1 - 1/\text{polylog}(n))$ allows us to simply apply a union bound over all values $\ell = c^r$.

7.2 Testing files with some sparseness

Real-life database files may be somewhat sparse, in that not every block will contain the maximum number of tuples. This can happen when tuples are deleted, or when the file was constructed beforehand using a primary index, for example in a B*-tree structure. Let n denote the actual number of tuples in a file (it is reasonable to assume that n is known), and let n' denote the number of "potential" tuples, i.e. the maximum number of tuples per block times the number of blocks in the file. It could happen that the number of tuples in each block is not known until a read is made to the block, which would make the probe model of the test hard to implement (the sort algorithm would continue to operate exactly as before).

If the density of each block is bounded from below by some constant α, then we can modify Algorithm 6 as follows: Instead of implementing it for n, implement it for n' (assuming for now that all blocks are full), and whenever the test attempts to probe a "missing tuple" (i.e. a tuple index inside a block that is larger than the actual number of tuples in that block) ignore it. In the main loop of Algorithm 6 this would mean that an i_j that is a missing index is not counted at all, and the loop goes until b valid indexes are tested (or until $O(b/\alpha)$ iterations have gone by and not enough indexes were found, in which case the tester fails). Algorithm 5 needs to be changed a little more. Again we search neighborhoods of increasing sizes, but for every neighborhood we also keep a statistic of the number of non-missing indexes. We then use it to estimate the neighborhood's "actual size", which will be used to decide which coordinate of A will it affect. The above would give estimates similar to those of Algorithm 6, with the complexity parameters now multiplied by $O(\alpha^{-2})$.

7.3 Distances larger than memory

Suppose that there is enough memory for handling a (k,ℓ)-nearly sorted relation, but we attempt to use Algorithm 1 when the input is in fact not (k,ℓ)-nearly sorted. If we implement Remark 4.2 then we will fall-back to the Replacement Selection algorithm. If the input is (k',ℓ)-nearly sorted for $k' > k$, then we can still bound the number of runs that will be produced – it is not hard to see, by partitioning R into consecutive subsequences so that each of them is (k,ℓ)-nearly sorted, that the number of runs is at most $k'/k + 1$ (the "+1" refers to the tuples remaining in memory in the end). Similarly, using the fall-back mechanism of Algorithm 2 described below would cause no more than k'/k overflows (however here more care is needed in handling the overflows).

A more careful analysis of Algorithm 3 would reveal that its probability for a false positive decays exponentially in k'/k, and so whenever it accepts, the expected risk of running Algorithm 1 or Algorithm 2 (instead of an algorithm more optimized for the completely unsorted case) is still small, because we would most likely still have a small number of runs allowing for an efficient merge. A similar decay in the error

probability holds for Algorithm 6. Finally, note that in some instances it may even be beneficial to run into the fall-back mode of our sorting algorithm on purpose, if the resulting number of runs r would be small enough to make an r-way merge more efficient than a multi-pass sorting algorithm.

7.4 Recovering from a fail mode

When Algorithm 2 reaches a failure mode in the first pass (we recall that it will not reach a failure mode in the second pass unless one was reached in the first pass) we cannot just dump the contents of G to a temporary file and continue from where we were.

An appealing direction would be then to try to partition to input file into consecutive segments where on each of them we can run Algorithm 2, and pipe their second phase input to a merge sort algorithm. This will not work with a multi-way merge sort algorithm as per [5], but it will work with the iterated 2-way merge sort algorithm of [6].

To follow on this, when S becomes empty and we reach the fail state of the first pass, we write G to a temporary file and start over from where we were. We also write down the location in R where the fail state was reached, and reset *last_handled* to MIN_VAL. Then, for the second pass, we have a partition of the original relation into subsequences, and a collection of runs from G, so that each pair of an original subsequence and a corresponding run written from G can be merged into a sorted run as per the second stage of Algorithm 2 (without a failure mode being reached). In terms of Section 7.3 we would have $k'/k+1$ such segments, which we pipe in turn to the iterated 2-way merge sort algorithm.

The reason that this will not work with a multi-way merge algorithm is the memory requirement of holding $O(k+\ell)$ tuples for "decoding" each sorted run. The only way to enable multi-way merge is in effect to revert to Algorithm 1 after the first time a fail state was reached. Starting from the second run, we just write it in its entirety to a temporary file. Then, under reasonable memory assumptions we can merge all the runs at once, with the first run requiring us to hold $O(k+l)$ tuples in memory to decode, and each subsequent run requiring enough memory to hold the $O(1)$ tuples that are buffered from the respective file.

7.5 Solid State Drives

Another issue to consider is how our algorithm can perform when the database is not stored on a traditional harddisk drive, but rather on a Solid State Drive (SSD). While current SSDs are mostly restricted to small appliance devices and high-end laptops, one can imagine a time where the technology would advance enough to supplant traditional drives, so it is not too soon to consider database algorithms designed for this new hardware profile.

There are two crucial differences between traditional drives and SSDs. The first is that with an SSD the seek time is negligible, and so a sequential read operation takes the same time as a random access read operation (however when many operations are involved, block sizes may become an issue). In fact seek times are recently becoming less and less relevant also for traditional disk drives under a modern operating system [13]. The second difference is that with some SSD devices writing is very time-costly relative to reading, and may also wear down the device itself.

In this context our algorithm should be compared against the algorithm from [5], which performs an $O(n/\ell)$-way merge sort, where n is the size of R and ℓ is the size of the runs produced by using quicksort on consecutive subsequences. This is a 2-pass algorithm for the practical purposes of current hardware memory size, where the second pass uses random access reads. In the following we assume that the relations are sufficiently nearly-ordered for our algorithms to work.

When comparing sort operations, Algorithm 2 outperforms the multi-way merge sort algorithm. Both our algorithm and the algorithm of [5] perform two passes of reading the full relation. However, while our algorithm writes nothing on disk apart from the final output (which can also be piped for further processing), the multiway merge sort algorithm has to first rewrite the entire relation as a sequence of sorted runs, which are then read for the merging stage.

The comparative analysis of Algorithm 2 also holds for the evaluation of other operators as in Section 5. Both the output of Algorithm 2 and the output of (the second stage of) the multiway merge algorithm can be directly piped to the operation at hand, but our algorithm saves the writing of the entire relation as sorted runs. In fact the only instance in which the algorithm of [5] is significantly better is when (say, due to an unlikely error of the near-sortedness test) we tried to run our algorithm on an input that is $\Theta(n)$-far from being ℓ-globally sorted.

There is additionally the question of testing whether our algorithm applies to a given relation, *i.e.*, testing whether the relation is nearly sorted. In SSDs the situation is even better than that of traditional drives, because the testing algorithm in fact fully depends on random access read operations.

7.6 Distributed scalability

Our sorting algorithms for nearly sorted inputs can be scaled to a distributed implementation as per the following sketch (we omit the fall-back procedure in case the algorithm fails).

To further understanding, we present the distributed implementation for Algorithm 1. A distributed version for Algorithm 2 can be derived from the following in much the same fashion as the original derivation of Algorithm 2 from Algorithm 1, by deferring all output to the final stage.

Assume that we have d processors, each with $O(k+\ell)$ memory (where we need to sort a (k,ℓ)-nearly sorted file), and with access to the whole file.[2] We also assume that d is small with respect to k and ℓ, and that all processors are trustworthy. The input file is partitioned into d consecutive equal size segments, each processor receiving charge of one segment. Each processor performs an independent run of the sorting algorithm for (k,ℓ)-nearly sorted input on its segment. However, also the first $\ell+k$ and the last $\ell+k$

[2]In fact it is easy to adapt the following to the case where each processor has only access to a consecutive sequence, where the entire input is the concatenation of these sequences. In this case the scalability would also depend on the way the input is "spread" between the processors.

records that were supposed to be output (to the respective intermediate file) are not output, but instead are collected along with the up to k out of order records. Exceptions are the beginning of the very first segment and the end of the very last segment, that are still written to their segment's output. A copy of the first and last record that was output for each segment is also kept for further checks, and then all segments are considered to be concatenated.

If none of the processors failed, we check whether the concatenation of the output segments is sorted (for this we kept the first and last record in each segment's output). If this check has passed then we may continue. We sort all the records that were not output yet. This requires $\widetilde{O}(d(k+\ell))$ computation and communication to pass records between processors so that each processor gets a consecutive segment from the sorted in-memory records, by proceeding along the following steps:

1. A designated processor chooses uniformly at random $t = O(d \log d)$ numbers in $\{1, \ldots, m\}$, where m is the total number of in memory records.
2. The processors holding the corresponding records report them. Let $r_1, ..., r_t$ denote the values of their key attributes in sorted order, let $r_0 = \text{MIN\_VAL}$ and let $r_{t+1} = \text{MAX\_VAL}$.
3. Each processor reports how many records it holds with key values between r_i and r_{i+1} for $0 \leq i \leq t$.
4. A designated processor calculates $0 = i_0 < i_1 < \cdots < i_d = t+1$ such that there are $O(m/d)$ records between r_{i_j} and $r_{i_{j+1}}$ for all $0 \leq j < d$; with high probability such i_j exist, and otherwise Step 1 above is restarted.
5. For every $j = 1, \ldots, d$ all processors communicate their records between $r_{i_{j-1}}$ and r_{i_j}, which processor j stores in order as its assigned records.

Finally, the records in memory are merged with the previous output. This can also be done in a distributed manner, after a preliminary binary search is performed over the previous intermediate output to assign to each processor a segment into which its records will be merged (one would expect the assigned segment boundaries to typically resemble the boundaries of the original output segments, but this cannot be analytically guaranteed). Note that although this procedure is only guaranteed for (k, ℓ)-nearly sorted relations, it may also work for sufficiently "evenly spread" $(O(kd), \ell)$-nearly sorted relations.

7.7 Preliminary experiments about the occurrence and usefulness of near-sortedness

We built and ran some TPC-C benchmarks according to the guidelines in the TPC-C standard [8] using a database generated according to Section 3 of the "Installation and User Guide" [8]. During the benchmark runs, we monitored the sort operations, and for each execution of a sort operation we checked the near-sortedness condition of the relation being sorted for various parameters k and ℓ. Our tests showed that for $k = \ell = \lceil \sqrt{n} \rceil$, where n is the number of tuples, more than 90% of the relations were (k, ℓ)-nearly sorted before the sorting started.[3]

The above are very preliminary experimental results. Currently we are working on building an implementation of our second sorting algorithm and plan to empirically measure its performance on nearly sorted relations (an in-memory version of the first sort algorithm has already been implementedand measured, but our main interest here is in external sort). The longer term goal is to build a fuller implementation of our algorithms inside a working database.

Acknowledgement. We thank Yehoshua Sagiv for his comments about the algorithms and their implementation.

8. REFERENCES

[1] N. Ailon, B. Chazelle, S. Comandur, and D. Liu. Estimating the distance to a monotone function. *Random Struct. Algorithms*, 31(3):371–383, 2007.

[2] F. Ergün, S. Kannan, R. Kumar, R. Rubinfeld, and M. Viswanathan. Spot-checkers. *J. Comput. Syst. Sci.*, 60(3):717–751, 2000.

[3] E. Fischer. The art of uninformed decisions: A primer to property testing. In *Current Trends in Theoretical Computer Science: The Challenge of the New Century*, volume I, pages 229–264. 2004.

[4] E. Fischer, E. Lehman, I. Newman, S. Raskhodnikova, R. Rubinfeld, and A. Samorodnitsky. Monotonicity testing over general poset domains. In *STOC*, pages 474–483, 2002.

[5] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems*. Prentice Hall, New Jersey, 2009.

[6] D. E. Knuth. *The art of Computer Programming*. Addison Wesley, third edition, 1998.

[7] P.-Å. Larson and G. Graefe. Memory management during run generation in external sorting. In *SIGMOD Conference*, pages 472–483, 1998.

[8] D. R. Llanos. Tpcc-uva: An open-source tpc-c implementation for global performance measurement of computer systems. 2006. ISSN 0163-5808.

[9] M. Parnas, D. Ron, and R. Rubinfeld. Tolerant property testing and distance approximation. *J. Comput. Syst. Sci.*, 72(6):1012–1042, 2006.

[10] D. Ron. Property testing (a tutorial). In *Handbook of Randomized Computing*. Kluwer Press, 2001.

[11] R. Rubinfeld. Sublinear time algorithms. In *International Congress of Mathematicians*, volume III, pages 1095–1110. EMS, 2006.

[12] E. K. S. Halevy. Distribution-free property testing. In *Proc. RANDOM*, pages 302–317, 2003.

[13] C. Staelin. Disk I/O in Linux. Technical Report HPL-2002-352, Hewlett-Packard Laboratories, 2002.

[3] These values for k and ℓ were selected because they fit the case where for a fixed n they would minimize the required working memory for the testing algorithm paired with the sorting algorithm; in fact for the current systems the amount of working memory would accommodate higher values of k.