Online Dynamic Reordering for Interactive Data Processing

Vijayshankar Raman Bhaskaran Raman Joseph M. Hellerstein
University of California, Berkeley
{rshankar,bhaskar,jmh}@cs.berkeley.edu

Abstract

We present a pipelining, dynamically user-controllable reorder operator, for use in data-intensive applications. Allowing the user to reorder the data delivery on the fly increases the interactivity in several contexts such as online aggregation and large-scale spreadsheets; it allows the user to control the processing of data by dynamically specifying preferences for different data items based on prior feedback, so that data of interest is prioritized for early processing.

In this paper we describe an efficient, non-blocking mechanism for reordering, which can be used over arbitrary data streams from files, indexes, and continuous data feeds. We also investigate several policies for the reordering based on the performance goals of various typical applications. We present results from an implementation used in Online Aggregation in the Informix Dynamic Server with Universal Data Option, and in sorting and scrolling in a large-scale spreadsheet. Our experiments demonstrate that for a variety of data distributions and applications, reordering is responsive to dynamic preference changes, imposes minimal overheads in overall completion time, and provides dramatic improvements in the quality of the feedback over time. Surprisingly, preliminary experiments indicate that online reordering can also be useful in traditional batch query processing, because it can serve as a form of pipelined, approximate sorting.

1 Introduction

It has often been noted that information analysis tools should be interactive [7, 5, 6], since the data exploration tasks they enable are often only loosely specified. Information seekers work in an iterative fashion, starting with broad queries and continually refining them based on feedback and domain knowledge (see [21] for a user study in a business data processing environment). Unfortunately, current data processing applications such as decision-support querying [9] and scientific data visualization [2, 20] typically run in batch mode: the user enters a request, the system runs for a long time without any feedback, and then returns an answer. These queries typically scan large amounts of data, and the resulting long delays disrupt the user's concentration and hamper interactive exploration. Precomputed summaries such as data cubes [14, 28] can speed up the system in some scenarios, but are not a panacea; in particular, they provide little benefit for the ad-hoc analyses that often arise in these environments.

The performance concern of the user during data analysis is not the time to get a complete answer to each query, but instead the time to get a reasonably accurate answer. Therefore, an alternative to batch behavior is to use techniques such as Online Aggregation [18, 4] that provide continuous feedback to the user as data is being processed. A key aspect of such systems is that users perceive data being processed over time. Hence an important goal for these systems is to *process interesting data early on*, so users can get satisfactory results quickly for interesting regions, halt processing early, and move on to their next request.

In this paper, we present a technique for reordering data on the fly based on user preferences — we attempt to ensure that interesting items get processed first. We allow users to dynamically change their definition of "interesting" during the course of an operation. Such online reordering is useful not only in online aggregation systems, but also in any scenario where users have to deal with long-running operations involving lots of data. We demonstrate the benefits of online reordering for online aggregation, and for large-scale interactive applications like spreadsheets. Our experiments on sorting in spreadsheets show decreases in response times by several orders of magnitude with online reordering as compared to traditional sorting. Incidentally, preliminary experiments suggest that such reordering is also useful in traditional, batch-oriented query plans where multiple operators interact in a pipeline.

The Meaning of Reordering

To provide intra-query user control, a data processing system must accept user preferences for different items and use them to guide the processing. These preferences are specified in a value-based, application-specific manner: data items contain values that map to user preferences. Given a statement of preferences, the reorder operator should permute the data items at the source so as to make an application-specific *quality of feedback* function rise as fast as possible. We defer detailed discussion of preference modeling until Section 3 where we present a formal model of reordering, and reordering policies for different applications.

1.1 Motivating Applications

Online Aggregation

Online Aggregation [18, 4] seeks to make decision-support query processing interactive by providing approximate, progressively refining estimates of the final answers to SQL aggregation queries as they are being processed. Reordering can be used to give users control over the rates at which items from different groups in a GROUP BY query are processed, so that estimates for groups of interest can be refined quickly.[1]

Consider a person analyzing a company's sales using the interface in Figure 1. Soon after issuing the query, the user can see from the estimates that the company is doing relatively badly in Vietnam, and surprisingly well in China, although the confidence intervals at that stage of processing are quite wide, suggesting that the Revenue estimates may be inaccurate. With online reordering, the user can indicate an interest in these two groups using the Preference "up" and "down" buttons of the interface, thereby processing these groups faster than others. This provides better estimates for these groups early on, allowing the user to stop this query and drill down further into these groups without waiting for the query to complete.

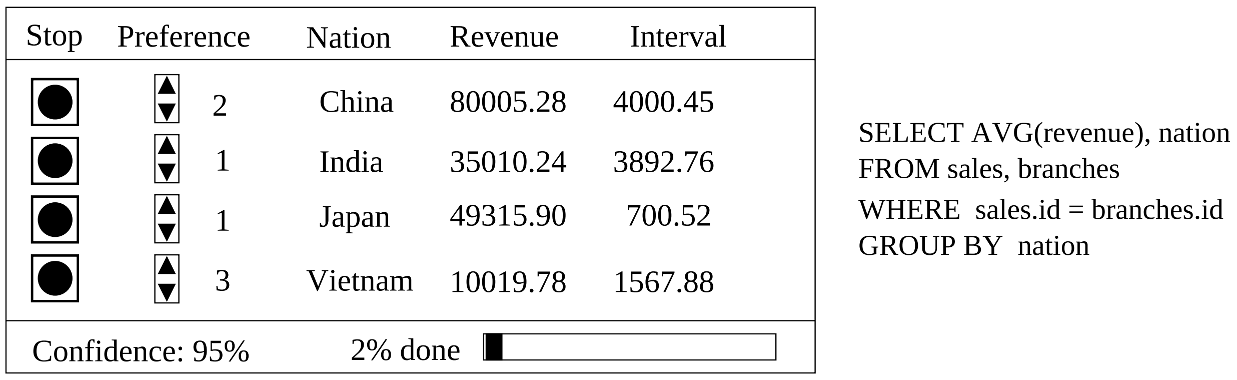
Another useful feature in online aggregation is fairness — it may be desirable that confidence intervals for the different groups should all tighten at same rate, irrespective of their cardinalities. Reordering can provide such fairness even when there is skew in the distribution of tuples across different groups.

Scalable Spreadsheets

DBMSs are often criticized as being hard to use, and many people prefer to work with spreadsheets. However, spreadsheets do not scale well; large data sets lead to inordinate delays with "point-and-click" operations such as sorting by a field, scrolling, pivoting, or jumping to particular cell values or row numbers. MS Excel 97 permits only 65536 rows in a table [11], sidestepping these issues without solving them. Spreadsheet users typically want to get some information by browsing through the data, and often don't use complex queries. Hence usability and interactivity are the main goals, and delays are especially annoying. This is unlike a DBMS scenario, where users expect to wait for a while for a query to return.

We are building [25] a scalable spreadsheet where sorting, scrolling, and jumping are all instantaneous from the user's point of view. We lower the response time as perceived by the user by processing/retrieving items faster in the region around the scrollbar – the range to which an item belongs is inferred via a histogram (this could be stored as a precomputed statistic or be built on the fly [12, 10]). For instance, when the user presses a column heading to re-sort on that column, he almost immediately sees a sorted table with the items read so far, and more items are added as they are scanned. While the rest of the table is sorted at a slow rate, items from the range being displayed are retrieved and displayed as they arrive.

[1] User preferences can be passed from the interface to the DBMS by calling UDFs in auxiliary queries. See [18] for details.



Stop	Preference	Nation	Revenue	Interval
●	▲▼ 2	China	80005.28	4000.45
●	▲▼ 1	India	35010.24	3892.76
●	▲▼ 1	Japan	49315.90	700.52
●	▲▼ 3	Vietnam	10019.78	1567.88

Confidence: 95%   2% done

```
SELECT AVG(revenue), nation
FROM sales, branches
WHERE  sales.id = branches.id
GROUP BY  nation
```

Figure 1: Relative speed control in online aggregation

Imprecise Querying: With online reordering behind it, the scrollbar becomes a tool for fuzzy, imprecise querying. Suppose that a user trying to analyze student grades asks for the records sorted by GPA. With sorting being done online as described above, the scrollbar position acts as a fuzzy range query on GPA, since the range around it is filled in first. By moving the scrollbar, she can examine several regions without explicitly giving different queries. If there is no index on GPA, this will save several sequential scans. More importantly, she need not pre-specify a range — the range is implicitly specified by panning over a region. This is important because she *does not know in advance what regions may contain valuable information*. Contrast this with the ORDER BY clause of SQL and extensions for "top N" filters, which require a priori specification of a desired range, often followed by extensive batch-mode query execution [8].

Batch Query Processing

Interestingly, we have found that a pipelining reorder operator is useful in batch (non-online) query processing too. Consider a key/foreign-key join of two tables R and S, with the foreign key of R referencing the key of S. If there is a clustered index on the key column of S, a good plan would be to use an index-nested-loops join algorithm. Taking advantage of the clustered index, the DBMS might insert a sort operator on R before the join, so that each leaf of the index is fetched at most once. Unfortunately, since sort is a blocking operator, this plan forfeits the pipelined parallelism that is available in index-nested-loops join. Note that sorting is used as a performance enhancement to batch up index lookups; total ordering is *not* needed for correctness. Hence we can use a pipelining, best-effort `reorder` operator instead to gain most of the benefit of sorting, without introducing a blocking operation into the query pipeline. Not only is the resulting query non-blocking (and hence potentially interactive), but the overall completion time may be faster than if we had sorted, since (a) we need not do a complete sort, and (b) opportunities exist for pipelined parallelism (e.g. if the index for S is on a separate disk from R). We have started experimenting with this idea by inserting reorder operators into traditional query plans, and present preliminary results in Section 5.3.

Organization of the Paper

We present our technique for reordering in Section 2. In Section 3 we describe several policies for reordering, and corresponding quality of feedback functions that are suited for different applications. We then discuss disk management issues for the reordering algorithm in Section 4. We present performance results for different applications in Section 5. We discuss related work in Section 6, and conclude with some avenues for future work in Section 7.

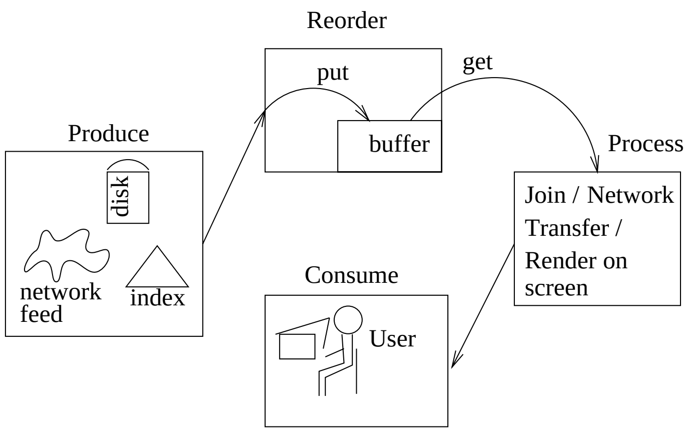


Figure 2: Data flow model for the reordering

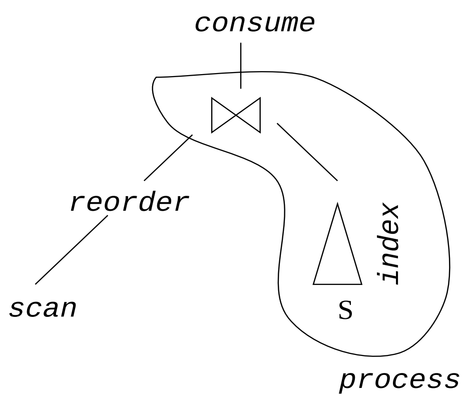


Figure 3: Reorder operators in plan trees for online aggregation

2 Best Effort Online Reordering

Since our goal is interactivity, the reordering must not involve pre-processing or other overheads that will increase runtime. Instead, we want a "best effort" reordering that runs concurrently with the processing, with negligible overhead. Figure 2 depicts our scheme of inserting a reorder operator into a data flow. We divide the data flow into four stages as described below.

`Produce` – this may be a disk scan, an index scan, or a data feed from a network or sensor.
`Reorder` – this reorders the items according to the dynamically changing preferences of the consumer.
`Process` – this is the set of operations done by the application, and it could involve query plan operators in a DBMS, sending data across a slow network, rendering data onto the screen in data visualization, etc.
`Consume` – this captures the user think-time, if any — it is important mainly for interactive applications such as spreadsheets or data visualization.

Since all these operations can go on concurrently, we exploit the difference in throughput between the `produce` stage and the `process` or `consume` stages to permute the items: *while the items taken out so far are being processed/consumed,* `reorder` *can take more items from* `produce` *and permute them.*

Figure 3 shows a sample data flow in a DBMS query plan. `Reorder` is inserted just above a scan operator on the table to be reordered, and the `process`ing cost is the cost of the operators above it in the plan tree.

2.1 The Prefetch and Spool (P&S) technique

`Reorder` tries to `put` as many interesting items as possible onto a main-memory buffer, and `process` issues requests to `get` an item from the buffer. When `Process` issues a `get` operation, `Reorder` decides which item to give it based on the performance goal of the application; this is a function of the preferences, and will be formally derived for some typical applications in Section 3.1.

The user preferences that indicate interest may be at the granularity of either an individual item or a group of items, depending on the application. Even in the former case, we can divide the items into groups based on a histogram, and reorder at the granularity of a group — `process` continually `get`s the best item from the best group on buffer[2]. `Reorder` strives to maintain items from different groups at different ratios on the buffer based on the preferences and the reordering policy. We derive these ratios in Section 3.2.

[2]When `process` issues a `get` operation, it is `reorder` that chooses the item to give out. The decision of which item to give out is made by `reorder` and is transparent to `process`

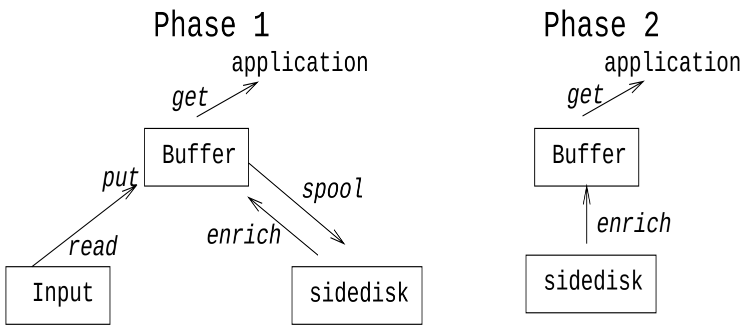


Figure 4: Reordering by Prefetch & Spool

i	index of a tuple
j	index of a group
g	number of groups
n	number of items processed so far
N	total number of items
DP	delivery priority
UP	normalized user preference
F	feedback function

Figure 5: Notation used in the reordering model

The P&S algorithm for reordering uses the time gap between successive `get`s from the buffer (which may arise due to processing or consumption time) to maintain the correct ratios of different groups on the buffer. It has two phases, as shown in Figure 4. In Phase 1 it continually scans the input, trying to maintain the appropriate ratio of items in the buffer by *spooling* uninteresting items to an auxiliary *side-disk*. It spools out items from the group that has the highest difference between the ratio of items actually on the buffer and the ratio desired. In the common case where `produce` is reading data from a disk, `reorder` performs sequential I/Os, and so can often go much faster than the stages down the data flow. If `reorder` finds that it has spooled some interesting items to the side-disk (as will happen if the user changes their definition of "interesting" midway), it may have to read them back from the side-disk to *enrich* the buffer. Again, it reads items from the group that has the highest difference between the ratio of items desired, and what is actually on the buffer. Phase 1 completes when the input is fully consumed. In Phase 2, it directly reads items from the side-disk to fill the buffer with needed items.
When `produce` is a continuous network feed, Phase 1 never finishes. `Reorder` in this situation will still have to spool out uninteresting items to the side-disk, assuming that the feed rate is faster than the `process` rate.

2.2 Index Stride

We have so far assumed that the order of data provided by `produce` is not under the control of `reorder`. However, if there is an index on the columns that are used to decide the user's preference for a tuple (the group-by columns in the case of online aggregation), we can use the index to retrieve items at different rates based on the ratio we want in the buffer. We open one cursor for each group and keep filling the buffer with items from that group whose ratio is less than what is needed. This approach — Index Stride — was described in [18]. However even if such an index exists, it is may not be clustered. We will see later that despite doing perfect reordering, Index Stride often does worse than regular *P&S* because of random I/Os into the index of `produce`.

As Figure 4 shows, *P&S* is a fairly simple algorithm. The details lie in choosing a reordering policy based on the performance goals, and in managing data in memory and on side-disk so as to optimize the enrich and spool operations. We tackle these issues in the next two sections.

3 Policies for online reordering

What exactly do we want to achieve by reordering a data set? Given an input stream $t_1, t_2, \ldots t_N$, we want to output a "good" permuted stream $t_{\pi_1}, t_{\pi_2}, \ldots, t_{\pi_N}$. Consider the following example. If the user divides data items into groups and is twice as interested in group A as in group B, one good output permutation will be "AABAABAAB...". This sequence corresponds to several possible permutations of the actual tuples, since many tuples fall into each group. Similarly, the permutation "BAABAABAA..." is also just as good. In general, there will be several equivalently good permutations, and our goal is to output some permutation from a good equivalence class.

For each prefix of length n of an output permutation $t_{\pi_1} t_{\pi_2} \dots t_{\pi_N}$, consider an application-specific *quality of feedback* function (henceforth this will called the *feedback function*) $F(UP(t_{\pi_1}), UP(t_{\pi_2}), \dots, UP(t_{\pi_n}))$. This function captures the value of the items in the prefix, and models their "interestingness" to the user. $UP(t_i)$ is the *user preference* for item t_i. Since the goal is to improve feedback in the early stages, the goodness of the output permutation is given by *the rate at which the feedback function rises as the number of items processed n goes from 1 to N*. We try (we can only try since the reordering is best-effort) for an output permutation π, such that for any $n, 1 \leq n \leq N$, the prefix $t_{\pi_1}, t_{\pi_2}, \dots, t_{\pi_n}$ of π maximizes $F(UP(t_{k_1}), UP(t_{k_2}), \dots, UP(t_{k_n}))$ over all n-element subsets $\{k_1, \dots k_n\}$ of $\{1, 2, \dots N\}$

We describe in Section 3.1 how to set F for different applications based on their performance goals. The choice of F dictates the choice of the item that `reorder` gives out when `process` issues a `get`; it gives out the item that will increase F the most.[3] This in turn dictates the ratio of items from various groups that `reorder` should try to maintain in the buffer. We describe how this is derived in Section 3.2.

3.1 Performance Goals and Choice of Items to Remove from the Buffer

Consider the data flow model of Figure 2. When `process` issues a `get`, `reorder` decides which item to give via a *delivery priority* mechanism. This priority for an item is computed dynamically based on how much the feedback F will change if that item is processed. `Reorder` gives out the item in the buffer with the highest delivery priority (which may not be the highest priority item overall). Note that the delivery priority is not the same as the user preference. The user preference depends on the user interest whereas the delivery priority depends on the feedback function. In fact, for the first two metrics given below, in steady state, assuming that the most interesting group will always be available on buffer, the delivery priority for all the groups will be equal. We proceed to outline some intuitive feedback functions for different applications. The notation we use is summarized in Figure 5.

Confidence metric: Average weighted confidence interval

In online aggregation, the goal is to make the confidence intervals shrink as fast as possible. One way to interpret user preferences for different groups is as a weight on the confidence interval. The feedback function is the negative of the average weighted confidence interval (we take the negative since a small confidence-interval width corresponds to high feedback). Almost all the large-sample confidence intervals used in online aggregation (see [15] for formulas for various kinds of queries) are of the general form: (Variance of the results seen so far)/ (number of items seen so far)$^{1/2}$. Hence $1/\sqrt{n_j}$ is a good indicator of the confidence interval for a group j. After n items are processed, the feedback function we want to rise as fast as possible is

$$F = -\sum_{j=1}^{g} \frac{UP_j}{\sqrt{n_j}} \text{ given that } n_1 + \dots + n_g = n$$

The application chooses items for processing such that F rises as fast as possible. If we process an item from group j, $\Delta(n_j) = 1$ and so F increases by the first derivative $UP_j / {n_j}^{1.5}$. Hence, to process the group which

[3] We can do only a local optimization since we know neither the distribution of items across different groups in the input to be scanned, nor the future user preferences. Our aim is to maximize the feedback early on, and not the overall feedback.

will increase F the most, we set a delivery priority of $DP_j = UP_j/{n_j}^{1.5}$. Each time we process an item from a group, the group's delivery priority decreases. Also, we always process an item in the buffer with the highest priority. Hence this acts a negative feedback, and at steady state, assuming that the highest priority item is always present on buffer, all the delivery priorities will be equal.

Rate metric: Preference as rate of processing

A simple alternative is that items from each group be processed at a rate proportional to its preference. This is primarily a functional goal in that it directly tells `reorder` what to do. However, it may be useful in applications such as analysis of feeds from sensors, where we want to analyze packets from different sources at different rates based on preferences; if the user finds the packet stream from one sensor to be anomalous, he may want to analyze those packets in more detail. We want the number of items processed for a group to be proportional to its preference, and the feedback function to maximize is the negative of the net deviation from these proportions:

$$F = -\sum_{j=1}^{g}(n_j - nUP_j)^2 \text{ given that } \quad n_1 + \cdots + n_g = n$$

At any given time we want to process the group that will make this deviation decrease the most. If we process an item from group t, $\Delta n_t = \Delta n = 1$. Hence F increases by the first derivative,

$-\Delta(\sum_{j=1}^{g}(n_j - nUP_j)^2) \; = \; -\sum_{j=1}^{g} 2(n_j - nUP_j)(\Delta n_j - \Delta n UP_j)$

$= 2(nUP_t - n_t)(1 - UP_t) + \sum_{j \neq t} 2(nUP_j - n_j)(0 - UP_j) \; = \; 2(nUP_t - n_t) - 2\sum_{j=1}^{g}(nUP_j - n_j)UP_j$

For F to rise fastest, we must process a group t which will cause the above expression to be maximum. Hence the delivery priority is set as $DP_j = nUP_j - n_j$, since the second term of the previous expression is the same for all groups. As in the previous metric, at steady state, assuming that the highest priority group is always available in the buffer, all the delivery priorities will be equal. For this metric, this also means that all the priorities are 0 (this can be seen by summing the expressions for DP_j). The deviation of the delivery priorities from 0 is a measure of how bad the reordering is.

Strict metric: Enforcing a rigid order

When we use a `reorder` operator in a traditional query plan instead of a sort operator, the goal is a sorted permutation. This can be achieved by assigning monotonically decreasing user preferences for each item from the one that is desired to be the first until the last item. After n items have been processed, the feedback function we want to maximize is

$$F = \sum_{i=1}^{n} UP_i$$

By processing an item n_i, F increases by UP_i. To make this rise fastest, we set the delivery priority to be $DP_i = UP_i$. That is, we always process the item with the highest user preference on buffer. We also use this metric for the spreadsheets application, with the preference for a range of items decreasing with its distance from the range being displayed (this is inferred from the scrollbar position).

3.2 Optimal Ratio on Buffer

Since `reorder` always gives out to `process` the highest delivery priority item in buffer, the delivery priority functions derived above directly dictate the ratio of items from different groups that `reorder` must maintain in the buffer. These ratios in turn determine the buffer replacement policy for `reorder`.

Confidence metric: At steady state, all the DP_j's are equal. Hence for any two groups j_1 and j_2, $UP_{j_1}/(n_{j_1}\sqrt{n_{j_1}}) = UP_{j_2}/(n_{j_2}\sqrt{n_{j_2}})$, and the ratio of items from any group j must be $UP_j^{2/3}/(\sum_{t=1}^{g} UP_t^{2/3})$.

Rate metric: As explained before, at steady state all DP_j's are 0. Since $DP_j = nUP_j - n_j$, the ratio of items from group j is UP_j. Indeed, the goal is to have the processing rate be proportional to preference.
Strict metric: If DP_i is UP_i, there is no specific ratio — the reorderer tries to have the highest preference item, then the next highest, and so on.

3.3 Handling Preference Changes

In the discussion so far we have not considered dynamic changes in the preferences. When this happens, we can do the subsequent reordering in one of two ways. We can either express the feedback F as a goal over the items to be delivered subsequently, or as a goal that "remembers history", and is expressed over all the items previously delivered as well. Correspondingly, we can compute delivery priorities either based only on items processed since the last preference change, or on all the items that have been processed since the initiation of the data flow. For the Confidence metric, we re-calculate the delivery priorities based on the new user preferences taking into account all the items that have been processed. This will mean, in the confidence metric case, that if the user preference for a group is increased in the middle of processing, there will be a spurt in the processing for that group — not only is the new preference high, but also we must compensate for not having processed enough items from that group (commensurate with the new preferences) earlier. However, in the Rate metric we calculate the delivery priorities based on the number of items processed since the last preference change, and not on the total number of items processed ($DP_j = n'UP_j - n'_j$, where n' is the number of items processed since the last preference change). Hence the user preferences determine the rate at which tuples from different groups are processed between consecutive preference changes. We chose not to do this for the Confidence metric because, statistically, the large-sample confidence interval for a group is defined in terms of the total number of items processed for that group. This is not an issue with the Strict metric; the priorities are independent of the number of items processed.

4 Disk Management During Reordering

The goal of `reorder` is to ensure that items from different groups are maintained in the buffer in the ratios desired by the application, as derived in Section 3.2. There are four operations which alter the set of items in the buffer: scanning from the input, spooling to the side-disk, enrichment from the side-disk, and `get`'s by the application (Figure 4). The ratios in the buffer are maintained by (a) evicting (spooling) items from groups that have more items than needed, and (b) enrichment with items from the group that is most lacking in the buffer. In essence the buffer serves as a preference-based cache over `produce` and the side-disk. We always strive to maintain some items in the buffer, even if they are not the most interesting ones; the presence of uninteresting items in the buffer may arise, for instance, if the user preferences during Phase 1 are very different from the data distribution across different groups. By guaranteeing the presence of items in the buffer, the `process` stage never has to wait for the `reorder` stage, and the overhead for introducing `reorder` into a data flow is minimized. In addition to the buffer, some memory is required for buffering I/O to and from the side-disk. Choosing the amount of memory for I/O buffers depends on the management of the side-disk, so we defer this discussion until after discussing our treatment of the side-disk.

4.1 Management of Data on Side-Disk

Since we want to process interesting items and give good feedback to the user early on, we must make Phase 1 as fast as possible and postpone time-consuming operations as long as possible. Another reason to finish

Phase 1 quickly is that during Phase 1 we cannot control the order of values appearing from `produce` if preferences change; whereas in Phase 2 we know the layout of data on the side-disk and can enrich the buffer with items that best satisfy preferences at a given time. To speed up Phase 1, we want a data layout on side-disk that makes spooling go fast even at the expense of enrichment, because we mainly do spooling in Phase 1 and enrichment in Phase 2.

Graefe [13] notes a duality between sorting and hashing. Hashing initially does random I/Os to write partitions, and later does sequential I/Os to read partitions. Sorting first writes out runs with sequential I/Os, and later uses random I/Os to merge the runs. Unfortunately, neither scheme is appropriate for reordering. Hashing into partitions is undesirable because the random I/Os in Phase 1 slow down spooling. Writing out sorted runs to disk is infeasible for two reasons. First, enrichment of the buffer with items from a particular group would involve a small, random I/O from each run, especially when the cardinality of the group is low. Second, dynamically-changing user preferences drive the decision of what we spool to or enrich from side-disk, meaning that the distribution of values to different spooled runs would be non-uniform[4].

To achieve the best features of both sorting and hashing, we decided to lay out tuples on the side-disk as fixed size *chunks* of items, where all the items in a chunk are from the same group. Spooling is done with only a sequential write, by appending a chunk to a sequential file of data on the side-disk. Enrichment is done via a random read of a chunk of that group which is most lacking in the buffer. Intuitively, this approach can be viewed as building an approximately clustered index (with only sequential I/Os) on the side-disk, concurrent with the other processing.

Returning to the main-memory layout described earlier, in Phase 1 we require an I/O buffer for each group to collect items into chunks. We also need a in-core index of pointers to chunks for each group to quickly find chunks corresponding to a group.

4.2 Total Ordering

We have so far assumed that the reordering is done at the granularity of a group. However in some applications, such as the reorder operators in batch query processing, our goal is a total ordering on the individual items. We tackle this by dividing the data into groups based on an approximate histogram. This histogram need not be accurate since we only want a best effort reordering. We want the number of groups to be as high as possible (this number is limited by the amount of memory we can allocate for the I/O buffers storing the chunks), so that the size of a group is small, and we do a "fine granularity" reordering that can distinguish between the priorities of small batches of tuples. The reorderer ensures a good ratio of items from different groups in the buffer, and the application removes for processing the best item in the buffer. Our experiments show that this technique is surprisingly successful, since these applications need only an approximate ordering.

5 Experimental Results

We present results that show the usefulness of reordering in online aggregation and in scalable spreadsheets. The aim is to study a) the responsiveness of the rate of processing to dynamic changes in preference, b) the

[4] We also tried out writing out items as packed runs on the side-disk, where the ratio of items in these runs is determined by current user preferences. With this method, ideally, with no preference changes, we never have to do any random I/Os — we keep appending runs to the side-disk in Phase 1, and keep reading runs in Phase 2. However we found that this method leads to severe fragmentation of items from sparse groups, and that the packing of items into runs is useless if the preference changes. This results in several small, random I/Os.

```
select o_orderpriority, count(*) from order
where o_orderdate >= '10/10/96' and
      o_orderdate < '10/10/96' + 90
and exists (select * from lineitem
            where  l_orderkey = o_orderkey and
                   l_commitdate < l_receiptdate)
group by o_orderpriority
```

Figure 6: TPC-D Query 4

Group	A	B	C	D	E
Preference at the start	1	1	1	1	1
Preference after 1000 tuples processed (T0)	1	1	1	5	3
Preference after 50000 tuples processed (T1)	1	1	3.5	0.5	1

Figure 7: Changes in User Preferences

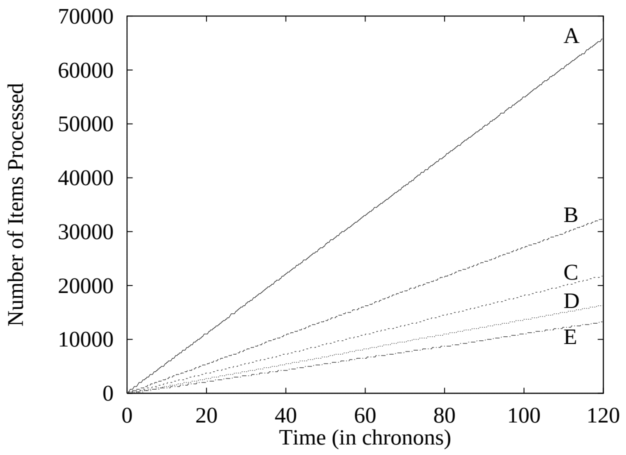


Figure 8: Performance of sequential scan, with *Rate* metric

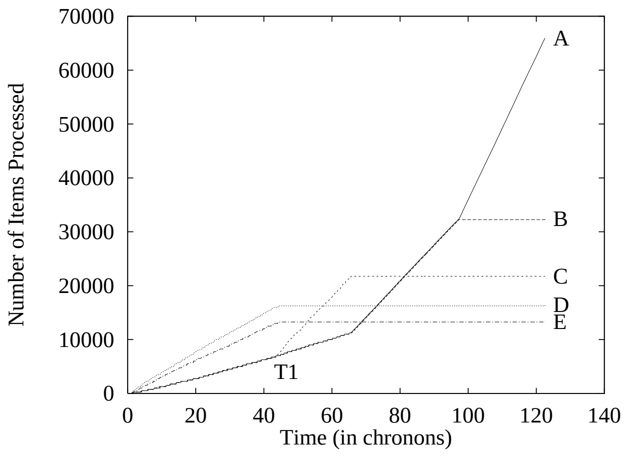


Figure 9: Performance of *P&S*, with *Rate* metric.

robustness of reordering to different data distributions and processing costs, and c) the overhead in overall completion time due to reordering. We also present promising initial results that show the advantage of using reorder operators in traditional batch query plans. We scale up all results involving Informix systems by an undisclosed factor to honor privacy commitments while still allowing comparative analysis of algorithms (hence time is expressed in abstract "chronons").

5.1 Online Aggregation

We have implemented our algorithms for reordering in the context of Online Aggregation in Informix Dynamic Server with Universal Data Option (UDO)[5]. The goal of reordering is to shrink the confidence intervals for the interesting groups as quickly as possible. We test the responsiveness and robustness of the reordering by varying a number of parameters:

Data distribution: To study the effect of skew in the data distribution across different groups, we test with Zipf and uniform distributions.

Processing Rate: We study a TPC-D query (Q4, see Figure 6), modifying it by adding and removing filters and tables to generate index-only joins (where we only need to look at the index of the inner table), index-nested-loop joins, and single table queries. We do not present results for queries with multiple joins because

[5]We ran all experiments on a 200 MHz UltraSPARC machine running SunOS 5.5.1 with 256MB RAM. We used the Informix Dynamic Server with Universal Data Option version 9.14 which we enhanced with online aggregation and reordering features. We chose a chunk size of 200KB for all our experiments. This is reasonable because the number of groups is typically small, and so the I/O buffer (whose size is chunk size $\times$ number of groups) is not very big. We used a separate disk for the side-disk, and the size of our buffer was 2MB (including the I/O buffer)

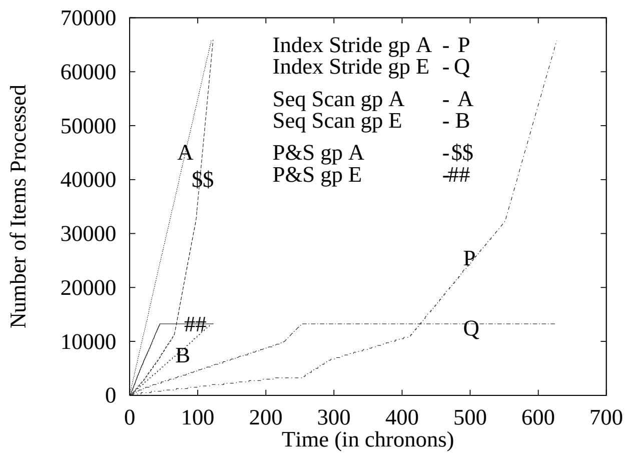


Figure 10: Comparison of different algorithms for processing groups A and E

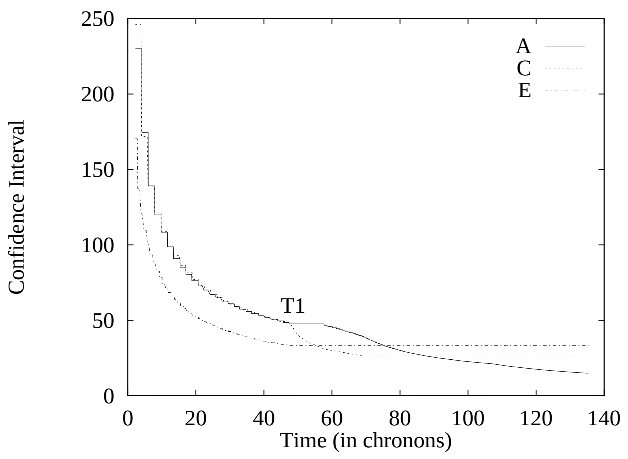


Figure 11: Confidence intervals of different groups for P&S

we see experimentally that even one join suffices for good reordering; more joins only make reordering easier by reducing the processing rate. In [17] we present experiments involving reordering with other `process` operators such as non-flattenable sub-queries and ripple joins[6].

Preference Change Model: We look at preference changes in Phase 1 and Phase 2, under the Confidence and Rate performance metrics from Section 3.

Algorithm used: We compare *P&S*, Index Stride, and a simple sequential scan (no reordering).

Space constraints prevent us from presenting exhaustive results along all dimensions. We show only results that illustrate salient features of the algorithms and indicate the trade-offs involved. We use the TPC-D *dbgen* program to generate data of different distributions, with a scale factor of 0.1. For our experiments we clustered the data in random order, so as to give statistically correct estimates for the aggregates [15].[7].

Rate Metric, Index-Only Join, Zipf distribution

Our first experiment uses a low processing cost query: *select avg(o_totalprice), o_orderpriority from order where exists (select * from lineitem where l_orderkey = o_orderkey) group by o_orderpriority.* We have removed filters from TPC-D Q4 to make it an index-only join with Order as the outer relation. Order has 150000 tuples. There are five groups which we will call A, B, C, D, and E for brevity, and we use a Zipf distribution (parameter 1) of tuples, in the ratio $1{:}\frac{1}{2}{:}\frac{1}{3}{:}\frac{1}{4}{:}\frac{1}{5}$ respectively. Initially all groups have equal preference of 1. After 1000 tuples are processed (time T0), the preference for D is increased to 5 and that for E to 3 (the change at T0 is not seen in most graphs because it occurs too early). After 50000 tuples are processed (time T1), the preference for C is increased to 3.5, and that for D is reduced to 0.5. Thus we separately test the effect of a preference change in Phase 1 and Phase 2 (Phase 1 ends at about 40000 tuples in most cases), and the ability to reorder when the interesting groups have low cardinalities. Figure 7 summarizes these details. Tables 1 and 2 summarize the main results from the experiments, which we proceed to explain below.

Figures 8 and 9 show the number of tuples processed for each group for sequential scan and *P&S* for the Rate metric. Despite the low processing cost and the high-preference groups being rare, the reordering of *P&S* is quite responsive, even in Phase 1. *P&S* has finished almost all tuples from the interesting groups D

[6]Ripple joins are specialized join algorithms for online aggregation that are efficient yet non-blocking [16]

[7]Note that the P&S reordering algorithm preserves the randomness properties of the data within a given group.

Distribution	Metric	Proc. Rate	Algorithm	Completion time	Completion time of interesting gp	Overhead
Any	Any	Low	Seq. Scan	119.8	119.8	0%
Zipf	Rate	Low	*P&S*	122.5	44.2	2.2%
Zipf	Rate	Low	Index Stride	627.1	253	423%
Zipf	Confidence	Low	*P&S*	135.3	47.0	12.9%
Zipf	Confidence	Low	Index Stride	615.5	425	413.7%
Uniform	Rate	Low	*P&S*	122.7	106.8	2.4%
Any	Any	High	Seq. Scan	1083.1	1083.1	0%
Zipf	Confidence	High	*P&S*	1094.8	684.4	1.1%
Any	Any	Very Low	Seq. Scan	16.1	16.1	0%
Zipf	Rate	Very Low	*P&S*	16.5	14.0	2.6%
Zipf	Rate	Very Low	Index Stride	100.9	45.3	626.7%

Table 1: Completion times in different cases

Dist.bn	Proc. Rate	Algorithm	Deviation at 30000 tuples processed					Deviation at 60000 tuples processed				
			A	B	C	D	E	A	B	C	D	E
Zipf	Low	*P&S*	−1098	−1097	−1097	27	3266	−0.18	−0.18	0.36	Fin	Fin
Zipf	Low	Index Stride	0.09	0.09	0.09	0.27	−0.54	−0.27	−0.27	0.54	Fin	Fin
Uniform	Low	*P&S*	−0.64	0.36	0.36	−0.18	0.09	−0.88	0.55	0.33	Fin	Fin
Zipf	High	*P&S*	−0.64	0.36	0.36	−0.18	0.09	−0.65	0.35	0.23	0.05	Fin
Zipf	Very Low	*P&S*	−5075	−4785	−2772	8507	4124	−2016	−1571	2170	−860	2278
Zipf	Very Low	Index Stride	0.64	−0.36	−0.36	0.18	−0.09	0	0	0	Fin	Fin

Table 2: Deviation of number of tuples processed of groups A,B,C,D,E, from desired values, for *P&S*. A value of Fin for a group means that it has been exhausted

and E by 44 chronons while sequential scan takes 120 chronons, *almost 3 times longer*. *P&S* imposes almost no overhead in overall completion time (2%).

Figure 10 compares the number of tuples processed for the largest group (A) and the smallest group (E) for different algorithms. The tuples of interesting group E are processed much faster for *P&S* than for other methods, while for the highest cardinality (and least interesting) group A, sequential scan produces items faster than *P&S*. Index stride does very poorly because of many random I/Os (it has a 427% completion time overhead).

To test the effect of reordering on groups with extremely small cardinalities, we added a new group F with 70 tuples, and gave it a constant preference of 2. While Index Stride finished all items from group F in 0.9 chronons, *P&S* took 41.6 chronons and the sequential scan took 111.2 chronons. Index Stride easily outperforms both *P&S* and sequential scan for this outlier group because *P&S* can only provide as many tuples of F as it has scanned. This advantage of Index Stride is also reported in [18].

Confidence Metric, Index-Only Join, Zipf distribution

We then removed group F and repeated the previous experiment with the Confidence metric. Figure 11 shows the shrinking of the confidence intervals for different groups for *P&S*.[8] We see that with *P&S* the confidence intervals shrink rapidly for the interesting groups (D,E, and later C), even when they have low cardinalities. In contrast, with a sequential scan, the intervals for D and E shrink more slowly than those for the other groups because D and E have low cardinality (we omit the graph to save space).

Interestingly, the overhead for *P&S* with the Confidence metric is about 12%, whereas it is only 2% for the Rate metric (see Table 1). To explain this, we plot the number of tuples processed from each group

[8] We do not plot the curve for the first few chronons since the large sample confidence intervals have not stabilized by then and plotting them will cause the rest of the graph to be scaled down too much. Also, we plot only groups A, C, and E to avoid cluttering the graph. The curve for group B overlaps that of A, and the curve of D is very similar to that of E.

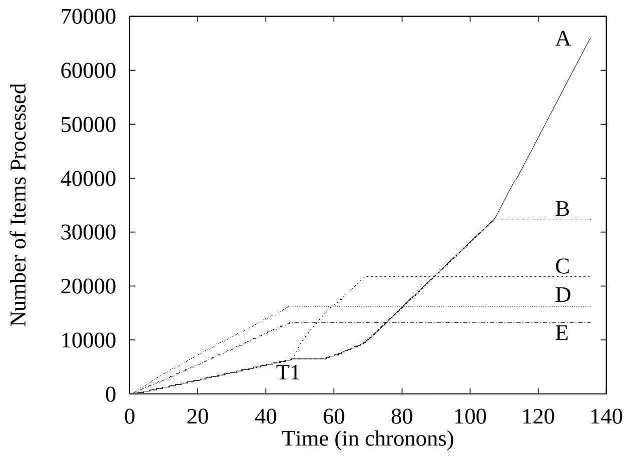


Figure 12: Rate of processing with the *Confidence* metric for P&S

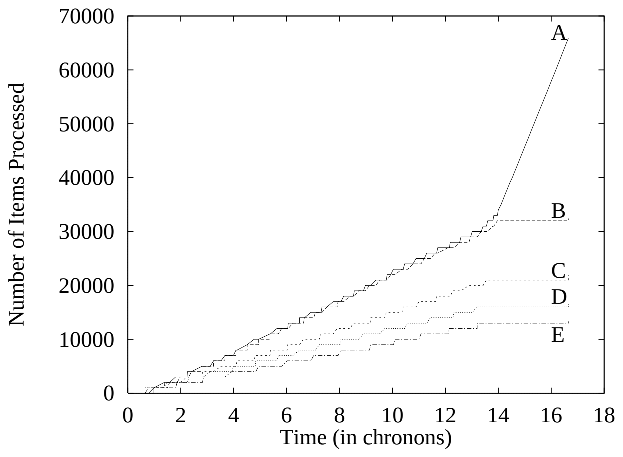


Figure 13: Rate of processing for *Rate* metric for P&S, single table query

for the Confidence metric for *P&S* in Figure 12. Comparing it with the performance under the Rate metric in Figure 9 we notice that just after T1 when the preferences are changed, the overall rate of processing in Figure 12 drops for a while. This arises because the Confidence metric remembers, and compensates for history as discussed in Section 3.3; after T1 there is a spurt in the processing of C because not only is its preference high, but also until then very few Cs have been processed. In fact, for a while after T1 (up to about 58 chronons), we are processing only group C. This results in random I/Os to get chunks of Cs from the side-disk, and hence a lower rate of processing. This spurt does not happen with the Rate metric because the priority depends only on the number of items processed since the last preference change.

Rate Metric, Index-Only Join, Uniform distribution

We now look at the effect of changing the distribution and the processing rates. We first repeat the previous experiment with the Rate metric and a uniform data distribution across different groups. To save space, we do not plot graphs but instead give in Table 2 the delivery priorities of different groups after 30000 tuples are processed and after 60000 tuples are processed. Recall that for the rate metric these priorities capture the deviation in the number of tuples processed from the number that should have been processed. After 30000 tuples are processed, the deviations are $A = -0.64$, $B = 0.36$, $C = 0.36$, $D = -0.18$, and $E = 0.09$ – this is almost an exact reordering. In contrast the deviations after 30000 tuples are processed for the Zipf distribution of the earlier experiment are much higher: $A = -1098.64$, $B = -1097.64$, $C = -1097.64$, $D = 27.09$, and $E = 3266.82$. The uniform distribution is easier to reorder since the interesting groups are available plentifully. The deviations after 60000 tuples have been processed are very small in both cases; reordering is easy in Phase 2 since we can directly read the needed chunks off the side-disk.

Rate Metric, Index Join, Zipf distribution

We next change the distribution back to Zipf and increase the processing cost: we reintroduce a filter to force an explicit join of Order and Lineitem. The new query is *select o_orderpriority, count(*) from order where exists (select * from lineitem where l_orderkey = o_orderkey and l_commitdate < l_receiptdate) group by o_orderpriority.* The reordering is much better even with the Zipf distribution: the deviations after 30000 tuples are processed are only -0.64,0.36,0.36, -0.18, and 0.09. The higher processing cost allows much better reordering because there is more time to reorder between consecutive `gets` of the data by `process`. The completion time overhead is also much lower (1% for the Confidence metric as against 12% in the

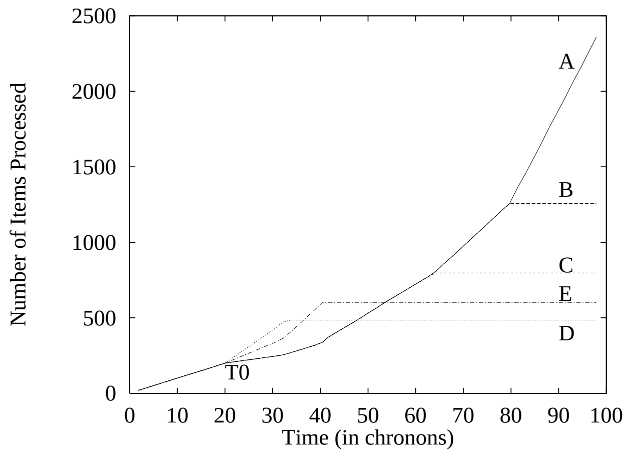


Figure 14: Rate of processing for *Rate* metric for *P&S*, original TPC-D query:

Operation	Number of tuples accumulated in range being panned over	Time Taken
Sort started	500	2.1 secs
User Thinking	5 seconds	
Short Jump	1000 1500	25.5 msecs 2.4 secs
User Thinking	10 seconds	
Random Jump	2500 5000	263 msecs 21.9 secs
Phase 2 has begun		
Short Jump	2500 5000	6.9 msecs 13.5 msecs
Random Jump	2500 5000	139 msecs 201 msecs

Figure 15: Scalable Spreadsheet: Latencies for various user actions

index-only join) since it is overshadowed by the cost of the processing.

Rate Metric, Single Table Query, Zipf distribution

Next, to stress our reorderer, we form a minimal-`process` query by removing Lineitem: *select o_orderpriority, count(*) from order group by o_orderpriority.* Figure 13 shows that the reordering is relatively ineffective in this case. Groups D and E are processed infrequently, since we can never spool to the side-disk and can only reorder within the buffer. This affirms that we can reorder effectively only when the `process`ing rate is less than the `produce` rate (*i.e.* the `process`ing cost is more than the `produce` cost). Here the only cost is that of the input scan — there is no processing save the addition to the Count aggregate. As Table 2 shows, Index Stride works very well, but it has a huge overhead of 626% — random I/Os have a high penalty since the processing cost is low.

Rate Metric, Original TPC-D Query, Zipf distribution

Finally we add back all the filters and tables, and run the complete TPC-D query given in Figure 6. Since this query specifies a low-selectivity filter on the Order table, very few tuples (5669) are handled by the reorderer. Figure 14 shows that *P&S* performs very well. The change at 1000 tuples processed is seen on this graph (T0) since the total number of tuples processed is small. Interestingly, with the predicate applied on Order, E becomes a more frequent group than D.

Discussion

Our experiments show that reordering by *P&S* is quite responsive to dramatic preference changes even with skewed distributions and low processing-cost queries such as index-only joins; interesting groups are processed much earlier than others even if they have low cardinalities. If the distribution is not too skewed, or the processing cost is higher (even one join suffices), or preferences are changed in Phase 2, we see that the reordering is virtually perfect, with small overheads for completion time. Index Stride has a very high overhead because of random I/Os, but works well for extremely low cardinality "outlier" groups. The reordering in the case of single-table queries is not good because of the low processing cost. However, note that joins are common in decision support queries — for example, 15 out of 17 TPC-D queries involve joins.

As the outlier group case shows, reordering is intrinsically difficult when the preferred groups are extremely rare. We could use a non-clustered index to fetch tuples from the most preferred groups alone (this is a hybrid of *P&S* and Index Stride), but this will involve several random I/Os. Alternatively, we can store tuples from these rare groups in a separate table, and have a clustered index on the group-by column on this table — this is similar to a partial index [24, 26], except that these tuples are now clustered separately and therefore one avoids multiple random I/Os. The challenge lies in automatically deciding which values of which column(s) to treat in this manner, taking into account the frequency of queries that will use this table, as well as the cardinality of tuples in the different groups. We intend to address this in future work.

5.2 Scalable Spreadsheets

In related work [25], we are building a GUI widget toolkit, and are using it to construct a spreadsheet that will be as interactive as a regular spreadsheet, but will scale up to large data sizes. An integral component of this spreadsheet is the online reordering library we have implemented, which provides immediate responses for operations such as sorting and pivoting. We have completed the reordering facility for sorting and present results which show that the system can respond almost instantaneously to operations such as scrolling and "jumping" while concurrently sorting the contents of the spreadsheet.

For our experiment, we used a table of 2,500,000 records of 100 bytes each (250MB total). The sort was issued on a 4 byte column, and we modeled a uniform distribution of values for this column in the table. We assume that we have a pre-computed equidepth histogram on that column, though it could be computed on the fly if we sample the data [10]. For `reorder` we chose a chunk size of 50KB in order to amortize the costs of random I/Os to enrich in Phase 2, and have an I/O buffer of 25MB. As explained in Section 4.2, we divided the data into the maximum number of groups possible, which is 25MB/50KB = 500 in this case; correspondingly, we divide the range of the key values into 500 groups based on the histogram, and use this partitioning to decide which group any given tuple belongs to. Therefore, each range has 2500000/500 = 5000 tuples. Since our goal is to sort, we reorder using the Strict metric of Section 3: the preference for different ranges decreases monotonically with their distances from the range the user is currently looking at (the exact values of the preferences assigned do not matter for the Strict metric).

Note that this is an application where the data can be `consumed` multiple times since the user may view the same portion of the spreadsheet many times. Hence the buffer is really only a cache of what is stored on the side-disk. Currently we have an additional in-memory display cache (1.5MB) to store the items that were recently panned over by the user. If we were to do this in a client-server setting where the user is seeing a spreadsheet on a client node and the data is on a separate server, then we believe that we could use multi-level caching schemes to avoid repeatedly sending the same data to the client [1].

To place the timings in our experiment in perspective, we sorted a 250MB file with 100 byte records using the UNIX `sort` utility. This took 890.4 seconds (we used a separate disk for the temporary files in order to avoid giving `reorder` any unfair advantage). We studied the following scenario (the timings are summarized in Figure 15): The user starts off by issuing a command to sort by a field, and we immediately start displaying the tuples of the topmost range – within 2.1 seconds, we have output 500 tuples, which is already enough to fill the screen. The user then analyzes the data in this range for 5 seconds, after which he makes a short jump (we model scrolls as short jumps to adjacent ranges) to the next range. As we see in the table, we are able to give 1000 tuples of this range almost at once (25 milliseconds), by enrichment from the side-disk – we have exploited the time the user spent analyzing the previous range to sort more items and so we can respond quickly. After this initial spurt of items in the desired range, we have exhausted all that is available on side-disk, and settle down to fetch more items at the sequential read bandwidth — the next 500 tuples in this range take around 2 seconds.

The user looks at this data for 10 seconds and then makes a random jump to a new location. Again we see that `reorder` (in 263 milliseconds) immediately provides 2500 items by enrichment before settling down to sequential read bandwidth (giving 5000 tuples, which is the total size of that range, takes 21.9 seconds). By this time, `reorder` has scanned the entire input and moved into Phase 2. All subsequent latencies are in milliseconds — a short jump (scroll) to a nearby range is about 20 times faster than jumping to a random location because the nearby range has a higher preference.

The above scenario clearly illustrates the advantage of online reordering in a scalable spreadsheet — *most operations have millisecond latencies with* `reorder`*, whereas a blocking sort take 15 minutes, which is several orders of magnitude higher.*

5.3 Query Processing

With the *Strict* metric, one could view `reorder` as an approximate, pipelining sort operator. As sketched in Section 1.1, best-effort reordering can be exploited in query plans involving "interesting orders" used for performance enhancements.

Although our focus is on user-controlled reordering, we have performed initial experiments to validate our intuitions, comparing the insertion of sort versus reorder operators into a few query plans in UDO. We consider a key-foreign key join of two tables R and S, where the foreign key of S references the key of R. R has 10^5 rows and S has 10^6 rows, each of size 200 bytes. S has a clustered index on the join column, but rows of R are clustered randomly (the join column values are uniformly distributed). A direct index nested loops join of R and S took 3209.6 chronons because it performed several random I/Os. Adding a sort operator before the join reduced the total time taken for the query to 958.7 chronons, since the sort batches index lookups into runs of similar values, resulting in at most one I/O per leaf page of the index.

We then replaced the sort operator with a `reorder` operator, using the *Strict* metric, with a chunk size of 25KB (we chose this so as to amortize the cost of a random I/O over a reasonably large chunk). We used a I/O buffer size of 2.5MB. This means that the number of groups we could support was 2.5MB/25KB = 100 (recall that we want the number of groups to be as high as possible so that we can do a fine granularity reordering). Hence, we divided the range of join column values into 100 groups for the reordering, and used a pre-computed equidepth histogram on the join column values of table R to identify the group to which each value belongs. The time required for the join with `reorder` is 899.5 chronons, which is even better than the time required for the traditional join by sorting. The 6% improvement in completion time occurs because we spool out fewer tuples to disk — the process stage directly `get`s 12.1% of the tuples from the buffer. We are able to do this because we only do a fuzzy, approximate reordering. This suffices because the only purpose of the sorting is to batch up tuples of R that match similar rows of S together. However, the biggest gain by using `reorder` instead of sort is that the plan has become non-blocking. This can allow us to exploit pipelined parallelism (if we have a parallel DBMS) and also allow interactive estimation techniques such as online aggregation. If we consider the rate at which output tuples are delivered, a plain index-nested-loops join delivers 311.6 tuples/chronon, and adding `reorder` increases the rate to 1111.7 tuples per chronon.

As future work, we want to study the effect of different distributions, different kinds of queries, and that of dynamically estimating the histogram [10, 12] instead of assuming a pre-computed one.

6 Related Work

Algorithmically, our reorder operator is most similar to the unary sorting and hashing operators in traditional query processing [13]. However our performance and usability goals are quite different, which leads to a different implementation. Logically our operator does not correspond to any previous work in the relational context, since it is logically superfluous – ordering is not "supposed" to matter in a strict relational model.

Our focus on ordering was anticipated by the large body of work on ranking in Information Retrieval [27]. In more closely related work, there have been a few recent papers on optimizing "top N" and "bottom N" queries [8], and on "fast-first" query processing [3]. These propose enhancing SQL with a stopping condition clause which the optimizer can use to produce optimal plans that process only those parts of the data that are needed for output. However in these papers, the user is required to specify *a priori* what portions of the data he is interested in, and does not have any dynamic control over the processing. Our work on spreadsheets can be viewed as an extension of this, where the user can dynamically specify what portions of the data interest him by moving a scrollbar, *after* seeing some partial results.

7 Conclusions & Future Work

Interactive data analysis is an important computing task; providing interactive performance over large data sets requires algorithms that are different than those developed for relational query processing. We have described the benefits of dynamically reordering data at delivery in diverse applications such as online aggregation, traditional query processing, and spreadsheets. A main advantage of reordering is that the user can dynamically indicate what areas of the data are interesting and speed up the processing in these areas at the expense of others. This, when combined with continual feedback on the result of the processing, allows the user to interactively control the processing so that he can extract the desired information faster.

This paper presents a framework for deriving the nature of the desired reordering based on the performance goals of an application, and have used this to come up with reordering policies in some typical scenarios. We have designed and implemented a reordering algorithm called Prefetch and Spool (P&S), which implements these ideas in a responsive and low-overhead manner. P&S is relatively simple, leveraging the difference between processing rate and data production. We have integrated P&S with a commercial database management system, and are using it as a core component in the development of a scalable spreadsheet. In the case of online aggregation, a single join above the reorder operator is sufficient for good reordering. Our simulation experiments with spreadsheet scrolling and sorting scenarios show that we can provide almost immediate — on the order of milliseconds — responses to sort operators that would otherwise take several minutes, by preferentially fetching items in the range under the scrollbar. Inserting reorder operators into standard query plans in place of sort operators is promising; initial results show that we are able to convert blocking plans into pipelines while actually reducing the completion time overhead.

This paper opens up a number of interesting avenues for further work that we intend to explore.

Other metrics for evaluating the feedback: Other feedback functions appear to be appropriate for applications that process real-time data such as stock quotes, since recent items are more important than earlier ones, and this must considered when calculating delivery priorities. We intend to study this and other policies in the future. We also believe that the idea of reordering policies is related to the idea of scheduling policies in the scheduling literature, and intend to investigate this further.

Supporting preferences from multiple users: We have only considered the problem of reordering data delivery to meet the dynamic preferences of a single user. When online reordering is used in applications such as broadcast disks, we need to consider the aggregate preferences from several users, and the reordering policy needs to be chosen suitably.

Data Visualization: In graphical data visualization ([2, 20]), large volumes of information are presented to the user as a picture or map over which he can pan and zoom. Fetching this data from the disk and rendering it onto the screen typically takes a long time. It makes sense to fetch more data points from the region the user is currently panning over and a small region around it, so that these portions can be rendered in greater detail / higher resolution. Here the user interest is inferred based on mouse position, and this is a two-dimensional version of the spreadsheet problem.

Other uses of reordering in query processing: A pipelining best-effort reorder operator appears to be substitutable for regular sort operators at other places in query plans. For instance, it can replace a sort operator that is designed to reuse memoized values of a correlated subquery or expensive user-defined function [22, 19, 23]. Here, online reordering amounts to computing the set of variable bindings on the fly, possibly with some duplication.

Acknowledgments

We would like to thank all the members of the CONTROL project at Berkeley for many useful discussions. Mehul Shah helped clarifying the performance goals for the Rate metric. Paul Aoki gave several detailed comments on a draft of the paper. The idea of using reordering in general query plans was suggested by Surajit Chaudhuri. Discussions with Miron Livny were useful in understanding the limits of best-effort reordering in meeting performance goals. We want to thank Informix Corporation for allowing us access to their source code. Satheesh Bandaram and Kathey Marsden helped us with the programming on UDO. Computing and network resources were provided through NSF RI grant CDA-9401156. This work was supported by a grant from Informix Corporation, a California MICRO grant, NSF grant IIS-9802051, and a Sloan Foundation Fellowship.

References

[1] S. Acharya, M. Franklin, and S. Zdonik. Balancing push & pull for data broadcast. In *SIGMOD*, 1997.

[2] A. Aiken et al. Tioga-2: A direct-manipulation database visualization environment. In *ICDE*, 1996.

[3] G. Antoshnekov and M. Ziauddin. Query processing and optimization in Rdb. *VLDB Journal*, 1996.

[4] R. Avnur, J. Hellerstein, et al. CONTROL: Continuous output and navigation technology with refinement on-line. In *SIGMOD*, 1998. Demonstration Description.

[5] M. Bates. Information search tactics. *Journal of the American Society for Information Science*, 30(4):205–214, 1979.

[6] M. Bates. *User Interface Design*, chapter The Berry-Picking Search. Addison-Wesley, 1990.

[7] D. Blair and M. Maron. An evaluation of retrieval effectiveness for a full-text document retrieval system. *CACM*, 28(3), 1985.

[8] M. Carey and D. Kossman. On saying "enough already" in SQL. In *SIGMOD*, 1997.

[9] S. Chaudhuri and U. Dayal. Data warehousing and olap for decision support. In *SIGMOD*, 1997.

[10] S. Chaudhuri et al. Random sampling for histogram construction. In *SIGMOD*, 1998.

[11] *Microsoft Excel 1997 – Online Help.*

[12] P. Gibbons et al. Incremental maintenance of approx. histograms. In *VLDB*, 1997.

[13] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 1993.

[14] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *ICDE*, 1996.

[15] P. Haas. Large-sample and deterministic confidence intervals for online aggregation. In *SSDBM*, 1997.

[16] P. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, 1999.

[17] J. M. Hellerstein et al. Informix under Control: Online query processing. *submitted for publication.*

[18] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, 1997.

[19] J. M. Hellerstein and J. Naughton. Query Execution Techniques for Caching Expensive Methods. In *SIGMOD*, 1996.

[20] M. Livny et al. Devise: Intergrated querying and visualization of large data sets. In *SIGMOD*, 1997.

[21] V. O'day and R. Jeffries. Orienteering in an information landscape: how information seekers get from here to there. In *INTERCHI*, 1993.

[22] P. Selinger et al. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[23] P. Seshadri et al. Cost based optimization for magic. In *SIGMOD*, 1996.

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

[25] Scalable Spreadsheets for Interactive Data Analysis. http://control.cs.berkeley.edu/ssheet.

[26] M. Stonebraker. The case for partial indexes. In *SIGMOD Record*, volume 18, 1989.

[27] C. van Rijsbergen. *Information Retrieval.* Butterworths, 1975.

[28] Y. Zhao, P. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimentional aggregates. In *SIGMOD*, 1997.