Balancing vectorized query execution with bandwidth-optimized storage

Zukowski, M

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 7

Cooperative scans

The previous chapter focused on improving the perceived disk bandwidth from the point of view of a single scan-based query. Another possible direction is to exploit the fact that with multiple queries running at the same time in a system, there are opportunities to share the data between them. This is typically achieved through special buffering policies in the storage layers. While there has been a lot of previous research in this area [CD85, SS86], disk scans were mostly considered trivial, and simple LRU or MRU buffering policies were proposed for them [CR93, SS86]. We show that if scans start at different times, these policies achieve only a low amount of buffer reuse. To improve this situation, some systems support the concept of circular scans [Co98, Coo01, NCR02, HSA05] which allows queries that start later to attach themselves to already active scans. As a result, the disk bandwidth can be shared between the queries, resulting in a reduced number of I/O requests. However, this strategy is not efficient when queries process data at different speeds or a query scans only a range of records instead of a full table.

In this chapter we analyze the performance of existing scan strategies, identifying three basic approaches: normal, attach and elevator. In normal, a traditional LRU buffering policy is employed, while in both attach and elevator incoming queries can join an ongoing scan in case there is overlap in data need, with the main difference that elevator employs a single, strictly sequential scan cursor, while attach allows for multiple (shared) cursors. Benchmarks show that they provide sharing of disk bandwidth and buffer space only in a limited set of scenarios. This is mostly caused by the fact that the disk access order is predefined when a query enters the system, hence it cannot be adjusted to optimize...
To overcome these limitations, we introduce the Cooperative Scans framework, depicted in Figure 7.1. It involves CScan – a modified (index) Scan operator that announces the needed data ranges upfront to an active buffer manager (ABM). The ABM dynamically optimizes the order of disk accesses, taking into account all current CScan requests on a relation (or a set of clustered relations). This framework can run the basic normal, attach and elevator policies, but also a new policy, relevance, that is central to our proposal. Besides optimizing throughput, the relevance policy also minimizes latency. This is done by departing from the strictly sequential access pattern as present in attach and elevator. Instead, relevance makes page load and eviction decisions based on per-page relevance functions, which, for example, try to evict pages with a low number of interested queries as soon as possible, while prioritizing page reads for short queries and pages that have many interested queries.

To further illustrate the need for a more flexible approach to I/O scheduling, consider the following example. Assume that a system has to execute two queries, $Q_1$ and $Q_2$, which enter the system at the same time and process data at the same speed. $Q_1$ needs to read 30 pages and is scheduled first, while $Q_2$ needs 10 different pages. If those queries get serviced in a round-robin fashion, as in the normal policy, $Q_2$ finishes after 20 pages are loaded, and $Q_1$ after 40, giving an average query latency of 30. The elevator policy may perform better, by first fully servicing $Q_2$ and then $Q_1$, reducing the average waiting time from 30 to 25. Still, elevator can choose the opposite order, resulting in waiting times of 30 and 40, hence actually increasing the average time. With relevance, we aim to get close to the optimal average query latency, without relying on the sequential scan order, by making flexible I/O scheduling decisions.
Section 7.1: Traditional scan processing

The outline of this chapter is as follows. First, Section 7.1 analyzes existing approaches to scan processing. In Section 7.2 we introduce the Cooperative Scans framework for row stores and we validate its performance in Section 7.3. In Section 7.4 we extend Cooperative Scans to column stores. The incorporation of ABM into an existing DBMS is discussed in Section 7.5, where we also explore possibilities of adapting order-aware query processing operators to handle out-of-order data delivery. We discuss related work in Section 7.6 before concluding in Section 7.7.

7.1 Traditional scan processing

With multiple scans running concurrently, in a naive implementation sequential requests from different queries can interleave, causing frequent disk-arm movements and resulting in a semi-random access pattern and low overall disk throughput. To avoid this problem, most database systems execute such scans using large isolated I/O requests spanning over multiple pages, together with physical clustering of table pages. As a result, the overhead of shifting the disk-arm is amortized over a large chunk of data, resulting in an overall bandwidth comparable to a standalone scan.

Even when using bandwidth-efficient chunk-based I/O, different scheduling policies are used for concurrent scans. The most naive, called normal in the rest of this chapter, performs scans by simply reading all disk blocks requested by a query in a sequential fashion, using an LRU policy for buffering. The disadvantage of LRU is that if one query starts too long after the other, the loaded pages will already be swapped out before they can be reused. As a result, assuming there is no buffer reuse between the queries, and queries are serviced in a round-robin fashion, the expected number of I/Os performed in the system until a new query $Q_{\text{new}}$ that reads $C_{\text{new}}$ chunks finishes can be estimated by:

$$C_{\text{new}} + \sum_{q \in \text{queries}} \text{MIN}(C_{\text{new}}, C_q).$$

The major drawback of the normal policy is that it does not try to reuse data shared by different running queries. In a dynamic environment, with multiple partial scans running at the same time, it is likely that the buffer pool contains some data that is useful for a given query. With a table consisting of $C_T$ chunks, a query that needs $C_Q$ chunks and a buffer pool of $C_B$ chunks, the probability of finding some useful data in the randomly-filled buffer is:

$$P_{\text{reuse}} = 1 - \prod_{i=0}^{C_B-1} \frac{C_T - C_Q - i}{C_T - i}$$

(7.1)
As Figure 7.2 shows, even for small scanned ranges and buffer sizes, this probability can be high, e.g. over 50% for a 10% scan with a buffer pool holding 10% of the relation. Unfortunately, the normal policy, by enforcing a sequential order of data delivery, at a given time can use only a single page, reducing this probability to $C_B/C_T$.

In many cases it is possible to relax the requirement of sequential data delivery, imposed by normal. Even when using a clustered index for attribute selection, consuming operators often do not need data in a particular order. This allows for scheduling policies with “out-of-order” data delivery.

A simple idea of sharing disk access between the overlapping queries is used in the attach strategy. When a query $Q_{new}$ enters the system, it looks at all other running scans, and if one of them ($Q_{old}$) is overlapping, it starts to read data at the current $Q_{old}$’s position. To optimize performance, attach should choose a query that has the largest remaining overlap with $Q_{new}$. Once $Q_{new}$ reaches the end of its desired range, it starts from the beginning until reaching the original position. This policy, also known as “circular scans” or “shared scans”, is used among others in Microsoft SQLServer [Coo01], RedBrick [Col98], and Teradata [NCR02], and allows significant performance improvement in many scenarios. The attach policy, however, may suffer from three problems. First, if one query moves much faster than the other, the gap between them may become so large that pages read by the fastest query are swapped out before the slower reaches them (they “detach”). Second, if queries are range scans, it is possible that one of the queries that process data together finishes, and the other continues by itself, even though it could attach to another running...
query. Also, if a full scan is underway but not yet in its range, *attach* misses this sharing opportunity. Finally, when exploiting per-block meta-data, the scan request can consist of multiple ranges, making it even harder to benefit from sharing a scan with a single query. As a result, the upper bound on the number of I/Os performed by *attach* is the same as in *normal*.

The *elevator* policy is a variant of *attach* that addresses its problems by enforcing *strict* sequential reading order of the chunks for the entire system. This optimizes the disk latency and minimizes the number of I/O requests, and thus leads to good disk bandwidth and query throughput. However, the problem here is that query speed degenerates to the speed of the slowest query, because all queries wait for each other. Also, range queries often need to wait a long time before the reading cursor reaches the data that is interesting for them. In principle, in the worst case the number of I/Os performed by a system before a fresh query $Q_{new}$ finishes can be $MIN(C_T, C_{new} + \sum_{q \in \text{queries}} C_q)$, where $C_T$ is the number of chunks in the entire table.

## 7.2 Cooperative Scans

The analysis in the previous section, further confirmed by results in Section 7.3, demonstrates that existing scan-processing solutions that try to improve over the *normal* policy still suffer from multiple inefficiencies. In this section we propose a new “Cooperative Scans” framework that avoids these problems. As Figure 7.1 presents, it consists of a cooperative variant of the traditional (index) *Scan* operator, named *CScan*, and an Active Buffer Manager (*ABM*).

The new *CScan* operator registers itself as an *active scan* on a range or a set of ranges from a table or a clustered index. *CScan* has the same interface as the normal *Scan* operator, but it is willing to accept that data may come in a different order. Note that some query plans exploit column ordering present on disk. We discuss integration of such queries in our framework in Section 7.5.

The *Active Buffer Manager* (*ABM*) extends the traditional buffer manager in that it keeps track of *CScan* operators and which parts of the table are still needed by each of them, and tries to *schedule* disk reads such that multiple concurrent scans reuse the same pages. The overall goal of *ABM* is to minimize the average query cost, keeping the maximum query execution cost reasonable (i.e. ensuring “fair” treatment of all queries). As discussed in Section 4.3 in MonetDB/X100 scan processing is usually performed with large I/O units we call *chunks*, to achieve good bandwidth with multiple concurrent queries. Note that a chunk in memory does not have to be contiguous, as it can consists of
multiple pages filled in with a single scatter-gather I/O request. In our framework there are two more reasons for using chunks. First, the number of chunks is usually one or two orders of magnitude smaller than the number of pages, thus it becomes possible to have chunk-level scheduling policies that are considerably more complex than disk-block-level policies. Secondly, it is possible to extend chunks to be logical entities whose boundaries may not even correspond exactly to disk block boundaries, a feature that will be exploited in the more complex scenarios with column-based storage.

In our system, the Cooperative Scans framework implements the traditional scan-processing policies: normal, attach and elevator. However, its main benefit comes from a newly introduced relevance policy that takes scheduling decisions by using a set of relevance functions. Both the CScan and ABM processes, as well as the relevance functions used by them, are described in Figures 7.3, 7.4 and 7.5, respectively.

As Figure 7.3 illustrates, the CScan process is called on behalf of a certain query, \( q_{\text{trigger}} \), that contains a CScan operator in its query plan. Each time \( q_{\text{trigger}} \) needs a chunk of data to process, selectChunk is called. This triggers a search over all buffered chunks that still need to be processed by the query, in chooseAvailableChunk, and returns the most relevant one, as governed by

---

**Figure 7.3: Pseudo-code for the Relevance policy: CScan operator**

```
selectChunk(q_{\text{trigger}})
| if finished(q_{\text{trigger}})
| return NULL
else
  | if abmBlocked()
  | signalQueryAvailable()
  | chunk = chooseAvailableChunk(q_{\text{trigger}})
  | if (chunk == NULL)
  |  chunk = waitForChunk(q_{\text{trigger}})
  | return chunk

chooseAvailableChunk(q_{\text{trigger}})
  c_{\text{available}} = NULL, U = 0
  foreach c in interestingChunks(q_{\text{trigger}})
    | if chunkReady(c) and useRelevance(c) > U
    |  U = useRelevance(c)
    |  c_{\text{available}} = c
  return c_{\text{available}}
```
Section 7.2: Cooperative Scans

ABM process

```plaintext
main()
    while (true)
        query = chooseQueryToProcess()
        if query == NULL
            blockForNextQuery()
            continue
        chunk = chooseChunkToLoad(query)
        slot = findFreeSlot(query)
        loadChunk(chunk, slot)
        foreach q in queries
            if (chunkInteresting(q, chunk) and queryBlocked(q))
                signalQuery(q, chunk)
        chooseQueryToProcess()
            relevance = -∞, query = NULL
            foreach q in queries
                qr = queryRelevance(q)
                if (query == NULL or qr > relevance)
                    relevance = qr
                    query = q
        return query

chooseChunkToLoad(q_trigger)
    c_load = NULL, L = 0
    foreach c in interestingChunks(q_trigger)
        if (not chunkReady(c)) and loadRelevance(c) > L
            L = loadRelevance(c)
            c_load = c
    return c_load

findFreeSlot(q_trigger)
    s_evict = NULL, K = ∞
    foreach s in slots
        if empty(s)
            return s
        c = chunkInSlot(s)
        if (not currentlyUsed(s)) and (not interesting(c, q_trigger))
            and (not usefulForStarvedQuery(c))
            and keepRelevance(c) < K
                K = keepRelevance(c)
            s_evict = s
    freeSlot(s_evict)
    return s_evict
```

Figure 7.4: Pseudo-code for the Relevance policy: Active Buffer Manager
### NSM Relevance Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>queryRelevance(q)</code></td>
<td><code>if not queryStarved(q) return -∞ return - chunksNeeded(q) + waitingTime(q) / runningQueries()</code></td>
</tr>
<tr>
<td><code>useRelevance(c, q_trigger)</code></td>
<td><code>return Q_{max} - numberInterestedQueries(c)</code></td>
</tr>
<tr>
<td><code>loadRelevance(c)</code></td>
<td><code>return numberInterestedStarvedQueries(c) * Q_{max} + numberInterestedQueries(c)</code></td>
</tr>
<tr>
<td><code>keepRelevance(c, q_trigger)</code></td>
<td><code>return numberInterestedAlmostStarvedQueries(c) * Q_{max} + numberInterestedQueries(c)</code></td>
</tr>
<tr>
<td><code>queryStarved(q)</code></td>
<td><code>return numberOfAvailableChunks(q_{trigger}) &lt; 2</code></td>
</tr>
</tbody>
</table>

Figure 7.5: Pseudo-code for the Relevance policy: Relevance functions

**useRelevance.** If no such chunk is available, the operator blocks until the ABM process loads a chunk that is still needed by `q_{trigger}`. Our `useRelevance` function promotes chunks with the smallest number of interested queries. By doing so, the less interesting chunks will be consumed early, making it safe to evict them. This also minimizes the likelihood that less interesting chunks will get evicted before they are consumed.

The ABM thread continuously monitors all currently running queries and their data needs. It schedules I/O requests on behalf of the query with the highest priority, considering the current system state. For this query, it chooses the most relevant chunk to load, possibly evicting the least relevant chunk present in the buffer manager. Once a new chunk is loaded into the ABM, all blocked queries interested in that chunk are notified. This is the core functionality of ABM’s main loop, as found in the middle part of Figure 7.4.

The `chooseQueryToProcess` call is responsible for finding the highest priority query, according to `queryRelevance`, to load a chunk for. This `queryRelevance` function considers non-starved queries (i.e. a queries that have 2 or more available chunks, including the one they are currently processing) equal, assigning them the lowest priority possible. Starved queries are prioritized according to the
amount of data they still need, with shorter queries receiving higher priorities. However, to prevent the longer queries from being starved forever, the priorities are adjusted to also promote queries that are already waiting for a long time. By prioritizing short queries, ABM tries to avoid situations where chunks are assigned to queries in a round-robin fashion, as this can have a negative impact on query latency. Besides, a chunk loaded for a short query has a higher chance of being useful to some large query than the other way around. In case ABM does not find a query to schedule a chunk for, it blocks in blockForNextQuery, until the CScan operator wakes it up again using signalQueryAvailable.

Once ABM has found a query to schedule a chunk for, it calls the chooseChunkToLoad routine to select a not yet loaded chunk that still needs to be processed by the selected query. The loadRelevance function determines which chunk will actually be loaded, not only by looking at what is relevant to the current query, but also taking other queries needs into consideration. To maximize sharing, it promotes chunks that are needed by the highest number of starved queries, while at the same time slightly adjusting priorities to prefer chunks needed by many non-starved queries.

If there are no free slots in the buffer pool, ABM’s findFreeSlot routine needs to swap out the chunk with the lowest keepRelevance. This function is similar to loadRelevance, except that when looking at queries, we treat queries on the border of starvation as being starved, to avoid evicting their chunks, which would make them starved, hence schedulable, immediately.

The relevance policy tries to maximize buffer pool reuse without slowing down fast queries. Thus, a slow query will re-use some of the chunks loaded by a fast query, skipping over chunks that it was too slow to process. These are read again later in the process, when the fast query might already be gone. The access pattern generated by this approach may be (quasi-) random, but since chunks consist of multiple sequential pages, disk (arm) latency is still well amortized.

7.3 Row-wise experiments

Benchmark system: We carried out row storage experiments using the PAX storage model, which is equivalent to NSM in terms of I/O demand, but better suited for the MonetDB/X100 execution layer. The chunk size used was 16MB, and the ABM buffer-pool size was set to 64 chunks (1GB), unless stated otherwise. Direct I/O was used, to avoid operating system buffering. Our test machine was a dual-CPU AMD Opteron 2GHz system with 4GB of RAM. The
storage facility was a 4-way RAID system delivering slightly over 200 MB/s. **Benchmark dataset:** We used the standard TPC-H \cite{Tra06} benchmark data with scale factor 10. In this setting the *lineitem* table consumes over 4GB of disk space. The other tables are fully cached by the system. **Queries:** To allow flexible testing of our algorithms we have chosen two queries based on the TPC-H benchmark. Query *FAST* (F) is TPC-H Q6, which is a simple aggregation. Query *SLOW* (S) is TPC-H Q1 with extra arithmetic computations to make it more CPU intensive. For all queries we allow arbitrary scaling of the scanned table range. In this section we use the notation QUERY-PERCENTAGE, with QUERY representing the type of query, and PERCENTAGE the size of the range being scanned. For example, with F-10 we denote query *FAST*, reading 10% of the full relation from a random location. We use multiple query streams, each sequentially executing a random set of queries. There is a 3 second delay between starting the streams, to better simulate queries entering an already-working system.

### 7.3.1 Comparing scheduling policies

Table 7.1 shows the results for all scheduling policies when running 16 streams of 4 queries. We used a mix of slow and fast queries with selectivity of 1%, 10%, 50% and 100%. The two major system-wide results are the average stream running time, representing the system throughput, and the average normalized latency of a query (running time in this benchmark divided by the base time, when the query runs by itself with an empty buffer), representing the system latency. Additionally we provide the total execution time, CPU-utilization, and the number of issued I/Os. The difference between the total time and the average stream time comes from the random distribution of queries in the streams, resulting in a significant variance of stream running times. For each query type and policy we provide the average latency, normalized latency and number of I/Os issued when scheduling this query type. Additionally, Figure 7.6 presents a detailed analysis of the I/O requests issued by each policy.

As expected, the *normal* policy achieves the worst performance. As Figure 7.6 shows, it maintains multiple concurrent sequential scans, which leads to the largest number of I/O requests and a minimal buffer reuse. Since the query load is relatively CPU-intensive, it still manages to use a significant fraction of the CPU time.

The *attach* policy allows merging requests from some queries. As a result, it consistently improves the performance of all query types and the system throughput. Still, in Figure 7.6 we see that there are multiple (albeit fewer than
Table 7.1: Row-storage experiments (PAX) with a set of FAST and SLOW queries scanning 1%, 10%, 50% and 100% of a table (16 streams of 4 random queries, all times in seconds)
Figure 7.6: Behavior of different scheduling algorithms: disk accesses over time
in normal concurrent scans, since not all queries can share the same chunk sequence. Additionally, we can see that a faster query can detach from a slower one (circled), resulting in a split of a reading sequence, further reducing the performance.

The elevator policy shows a further reduction of the I/O requests and improvement of system throughput. This is a result of its simple I/O pattern seen in Figure 7.6. However, we see that the average normalized latency is very bad for this policy. This is caused by the short queries that suffer from a long waiting time, and achieve results even worse than in normal. This blocking of queries also degrades the overall system time, since it delays the start moment of the next query in a given stream. We also see that fast and slow queries differ little in performance - this is caused by the fast queries waiting for the slow ones.

Our new relevance policy achieves the best performance, both in terms of global system parameters, as well as in most query times. As Figure 7.6 shows, its I/O request pattern is much more dynamic than in all the other policies. Interestingly, although relevance issues more I/Os than elevator, it still results in a better throughput. This is because the system is mostly CPU-bound in this case, and extra available I/O time is efficiently used to satisfy further query requirements. Also, relevance differs from other policies by significantly improving the performance of I/O bound queries. Average query latency is three times better than normal and two times better than attach (I/O bound queries like F-100 can even be three times faster than attach).

7.3.2 Exploring many different query mixes

To provide more than accidental evidence of the superior performance of relevance, we conducted experiments with the same basic settings as in the previous section: 16 streams of 4 queries, TPC-H table with scale factor 10 and buffer size of 1GB. However, we changed the set of queries to explore two dimensions: range size and data-processing speed. Figure 7.7 shows the results, where we compare throughput as the average stream running time (y axis) and average normalized query latency (x axis) of normal, attach and elevator, with respect to our relevance policy. Each point represents a single run for one policy. The point labels describe runs in a format “SPEED-SIZE”, where SPEED defines what query speeds were used (FS - mix of fast and slow queries, F - only fast ones, FFS - 2 times more fast queries than slow ones etc.), and SIZE represents the size of the range being scanned: S - short (mix of queries reading 1, 2, 5, 10 and 20% of a table), M - mixed (1,2,10,50,100) and L - long (10,30,50,100).

From this experiment we conclude that indeed relevance, representing the
Figure 7.7: Performance of various scheduling policies for query sets varying in processing speed and scanned range

(1,1) point in this scatter plot, is consistently better than all other policies. Recall that our objective was to find a scheduling policy that works well both on the query throughput and on the query latency dimension, and this is exactly what is shown in Figure 7.7. This scatter plot also allows us to better understand the other policies. We see that normal is inferior on both dimensions, whereas elevator gets close to relevance on throughput, but its performance is significantly hindered by poor query latencies. As for the attach, it does find a balance between throughput and latency, but it is consistently beaten by relevance in both dimensions.
Section 7.3: Row-wise experiments

Figure 7.8: Behavior of all scheduling policies under varying buffer pool capacities
7.3.3 Scaling the data volume

With growing dataset sizes, the percentage of a relation that can be stored inside the buffer pool decreases. To simulate this, we tested the performance of different policies under varying buffer size capacities, ranging from 12.5% to 100% of the full table size. This allows us to observe how different scheduling policies would behave under growing relation sizes, when a smaller fraction of a table can be buffered. In this experiment we used a version of our relation trimmed-down to 2 GB, that can be fully cached in the memory of our benchmark machine. Figure 7.8 shows the results of a benchmark with two sets of queries, one disk-intensive, consisting only of fast queries, and a CPU-intensive one, consisting of a mix of fast and slow queries. We used 8 streams of 4 queries.

As expected, the number of I/Os is decreasing with increasing buffer size. In the disk-intensive case, the absolute system performance is directly influenced by this number, because the system is never CPU-bound. Still, thanks to better request scheduling, the relevance policy manages to improve the performance, issuing significantly fewer I/O requests even when using a 87.5% buffer capacity.

In the CPU-intensive scenarios, the number of I/Os influence the absolute time only partially. This is because most algorithms manage to make a system CPU-bound with some buffer capacity. For relevance even a small buffer size of 12.5% of the full table is enough to achieve this, as we can see by its mostly constant performance.

Interestingly, Figure 7.8 shows that the performance advantages of relevance over the other policies as observed in Figure 7.7 are maximized when tables get bigger (i.e. at low buffered percentages). When looking at attach, the most viable competitor, we see that throughput in I/O bound situations, as well as latency in CPU bound queries deteriorate strongly, and we expect the advantage of relevance to grow even more if table sizes become huge.

7.3.4 Many concurrent queries

With more concurrent queries the opportunities for data-reuse increase. In Figure 7.9 we present how the average query time changes when an increasing number of concurrent queries reads 5, 20 and 50% of our relation, using a buffer pool of 1GB. As expected, the benefit of relevance over normal grows with larger scans and more concurrency. We see that relevance also enhances its advantage over attach when more queries run concurrently, even exceeding the factor two observed in Figures 7.8 and 7.7 when scan ranges are very or moderately selec-
Section 7.3: Row-wise experiments

Figure 7.9: Performance comparison with varying number of concurrent queries and scanned ranges

tive. As this query set is uniform in terms of range sizes, elevator can score close to relevance, but we know from previous experiments that on combinations of short and long ranges it is not a viable competitor.

7.3.5 Scheduling-cost scalability

The cost of scheduling in relevance is significantly higher than for other policies. For example, the loadRelevance function needs to check every query for every table chunk, and do this for each chunk a query requests. Figure 7.10 presents the average times spent on scheduling when running 16 concurrent streams of 4 I/O-bound queries, each with the same relation stored in a different number of chunks of varying sizes. As expected, the overhead grows super-linearly - with smaller chunks, every query needs to scan more of them, and the decision process for each data request needs to consider more chunks. Still, even with the largest tested number of chunks, the scheduling overhead in the worst case does not exceed 1% of the entire execution time. In situations when such overhead is not acceptable, e.g. with relations consisting of hundreds of thousands of chunks, slightly less complex policies can be considered. Also, our relevance implementation is rather naive, leaving opportunities for optimizations that can significantly reduce the scheduling cost.
7.4 Improving DSM scans

After our successful experiments with relevance in row-storage, we now turn our attention to column-stores. The decomposed storage model (DSM) has recently gained popularity for its reduced disk-bandwidth needs, faster query processing thanks to improved data locality [ADHS01, HP03], possibility of vectorized processing (Section 5.2.2) and additional compression opportunities (Chapter 6 [AMP06]). While we will show that relevance can also be successful here, we first discuss why DSM is much more complex than NSM when it comes to scans in general and I/O scheduling in particular, and how this influenced our Cooperative Scans framework.

7.4.1 DSM challenges

Table columns stored using DSM may differ among each other in physical data representation width, either because of the data types used, or because of com-
Section 7.4: Improving DSM scans

For example, Figure 7.11 depicts column storage of a part of the TPC-H lineitem table, with some columns compressed with techniques presented in Chapter 6. This shows that we cannot assume a fixed number of tuples on a disk page, even within a single column. As a result, a chunk cannot consist of a fixed number of disk pages as in NSM. Instead, chunks are logical concepts, i.e. a horizontal partitioning of the table on the tuple granularity. For example, one may divide a table in conceptual chunks of a 100,000 tuples, but it is also possible to use variable-size chunks, e.g. to make the chunk boundary always match some key boundary. This implies that chunk boundaries do not align with page boundaries. The underlying storage manager should provide an efficient means to tell which pages store data from a chunk. Depending on the physical data representation, a single logical chunk can consist of multiple physical pages, and a single physical page can contain data for multiple logical chunks.

This logical-physical mismatch present in DSM becomes even more problematic when using large physical blocks of a fixed size for I/O, a technique introduced in NSM for good concurrent bandwidth. The first problem here is that when loading a block for one chunk, a potentially large amount of data from a neighboring chunk can be loaded at the same time. In NSM this does
not occur, as chunks and blocks are equivalent. In DSM, however, ABM needs to take special care to minimize situations in which this extra data is evicted before it could be used by a different chunk. Also, keeping a full physical block in memory to provide it to another query in the near future may result in a sub-optimal buffer usage. The second issue, buffer-space demand, is a general problem for DSM I/O scheduling. In NSM, for $Q$ concurrent queries the system requires memory for $2 \times Q$ (factor 2 because of prefetching) blocks, which is usually acceptable, e.g. 512MB for 16 concurrent scans using 16MB chunks/blocks. In DSM, however, to process a set of rows, data from multiple columns needs to be delivered. While some optimizations are possible, e.g. performing selections on some columns early [HLAM06], in general all the columns used by a query need to be loaded before the processing starts. As a result, a separate block is needed for every column used in a query, increasing the buffer demand significantly for multi-column scans, e.g. to 4GB for 16 scans reading 8 columns each. This can be improved (in both models) by analyzing which pages from blocks are already processed, and re-using them as-soon-as-possible for I/Os for different queries (with scatter-gather I/O). Both problems demonstrate that in DSM the performance and resource demand can be significantly improved by making algorithms page-aware. However, such solutions significantly complicate the implementation, and our current system currently handles only chunk-level policies.

The final DSM problem for Cooperative Scans is the reduced data reuse opportunity between queries. Figure 7.12 shows two queries reading a subset
of a table and their I/O requirements in both NSM and DSM. Comparing the logical data need to the physical data demand in both models, we see that the vertical expansion present in DSM is usually significantly smaller than the horizontal expansion present in NSM. As a result, fewer disk blocks are shared between the queries, reducing the chance of reusing the same block for different scans. In NSM, for a block fetched by some query $Q_1$, the probability that another query $Q_2$, reading $T_2$ tuples, will use it is proportional to $\frac{T_2}{T_T}$, where $T_T$ is the number of tuples in the entire table. In DSM, we need to take into account both vertical (as in NSM) and horizontal overlap, reducing this probability to $\frac{T_2}{T_T} \times P_{\text{overlap}}(Q_1, Q_2)$, where $P_{\text{overlap}}(Q_1, Q_2)$ is the probability of a column from $Q_1$ also being used in $Q_2$.

### 7.4.2 Cooperative Scans in DSM

The DSM implementation of the traditional policies is straightforward. In normal, the order of I/Os is strictly determined by the query and LRU buffering is performed on a (chunk,column) level. DSM attach joins a query with most overlap, where a crude measure of overlap is the number of columns two queries have in common. A more fine-grained measure would be to get average page-per-chunk statistics for the columns of a table, and use these as weights when counting overlapping columns. Just like in NSM, the DSM elevator policy still enforces a global cursor that sequentially moves through the table. Obviously, it only loads the union of all columns needed for this position by the active queries.

The framework for relevance in DSM is similar to that in NSM, with a few crucial differences, caused by the challenges discussed in the previous section:

**avoiding data waste** – as discussed, with I/O based on large physical blocks, it is possible that a block loaded for one logical chunk contains data useful for neighboring chunks. When the first chunk is freed, this data would be evicted. To avoid that, the choice of the next chunk for a given query is performed before the query blocks for a fully available chunk. The already-loaded part of the chunk is marked as used, which prohibits its eviction.

**finding space for a chunk** – in DSM it is possible that a subset of columns in a buffered chunk is not useful for any query. ABM first evicts blocks belonging to such columns. Then, it starts evicting useful chunks, using the $\text{keepRelevance}$ function to victimize the least relevant chunk. Note that, unlike in NSM, this eviction process is iterative, since due to different physical chunk sizes, possibly
multiple chunks need to be freed.\footnote{If multiple chunks need to be freed, the dependency between them should be taken into account, something missed by the greedy iterative approach used here. Choosing the optimal set of chunks to free is a good example of a \textit{knapsack problem} surfacing in DSM I/O scheduling.}

column-aware relevance functions – Figure 7.13 shows that the DSM relevance functions need to take into account the two-dimensional nature of column storage and the varying physical chunk sizes. Like in NSM, \textit{useRelevance} attempts to use chunks needed by few queries, to make them available for eviction. However, it also analyzes the size of a chunk, to additionally promote chunks occupying more buffer space. The \textit{loadRelevance} function looks at the number of starved queries that overlap with a triggering query and are interested in a given chunk. It also estimates the cost of loading a given chunk by computing the number of cold pages required for all needed columns. The returned score promotes chunks that benefit multiple starved queries, and require a small amount of I/O. The DSM \textit{keepRelevance} function promotes keeping chunks that occupy little space in the buffer pool and are useful for many queries.

column loading order – a final issue in DSM is the order of columns when loading a chosen chunk. If some queries depend only on a subset of the columns, it may be beneficial to load that subset first. Our current crude approach is to

\begin{verbatim}
useRelevance(c, qtrigger)
  cols = queryColumns(qtrigger)
  U = |interestedOverlappingQueries(c, cols)|
  Pu = numberCachedPages(c, cols)
  return Pu/U

loadRelevance(c, qtrigger)
  query_cols = queryColumns(qtrigger)
  queries = overlappingStarvedQueries(c, query_cols)
  cols = columnsUsedInQueries(queries)
  L = |queries|
  Pt = |columnPagesToLoad(c, cols)|
  return L/Pt

keepRelevance(c, qtrigger)
  starved = almostStarvedQueries(c)
  cols = columnsUsedInQueries(starved)
  E = |starved|
  Pe = |cachedColumnPages(c, cols)|
  return E/Pe
\end{verbatim}

Figure 7.13: DSM Relevance Functions
Section 7.4: Improving DSM scans

### Normal | Attach | Elevator | Relevance
---|---|---|---
System statistics
avg stream time | 536.18 | 338.24 | 352.35 | 264.82
avg norm. lat. | 7.05 | 4.77 | 15.11 | 2.96
total time | 805 | 621 | 562 | 515
CPU use | 61 % | 77 % | 82 % | 92 %
I/O requests | 6490 | 4413 | 2297 | 3639

<table>
<thead>
<tr>
<th>query</th>
<th>cold latency</th>
<th>avg latency</th>
<th>avg latency</th>
<th>avg latency</th>
<th>avg latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>F-01</td>
<td>0.92</td>
<td>6.12</td>
<td>4.68</td>
<td>26.95</td>
<td>3.17</td>
</tr>
<tr>
<td>F-10</td>
<td>2.99</td>
<td>21.01</td>
<td>16.39</td>
<td>45.64</td>
<td>10.19</td>
</tr>
<tr>
<td>F-50</td>
<td>15.88</td>
<td>191.12</td>
<td>108.53</td>
<td>141.84</td>
<td>64.97</td>
</tr>
<tr>
<td>F-100</td>
<td>26.53</td>
<td>364.33</td>
<td>198.86</td>
<td>145.81</td>
<td>90.16</td>
</tr>
<tr>
<td>S-01</td>
<td>1.90</td>
<td>6.92</td>
<td>5.07</td>
<td>54.75</td>
<td>3.33</td>
</tr>
<tr>
<td>S-10</td>
<td>8.15</td>
<td>47.93</td>
<td>37.96</td>
<td>103.12</td>
<td>21.93</td>
</tr>
<tr>
<td>S-50</td>
<td>36.28</td>
<td>148.19</td>
<td>126.20</td>
<td>134.19</td>
<td>88.19</td>
</tr>
<tr>
<td>S-100</td>
<td>71.25</td>
<td>346.65</td>
<td>259.14</td>
<td>184.60</td>
<td>231.38</td>
</tr>
</tbody>
</table>

Table 7.2: Column-storage experiments with a set of FAST and SLOW queries scanning 1%, 10%, 50% and 100% of a table (16 streams of 4 random queries, all times in seconds)

Just load column chunks in increasing size (in terms of pages), which maximizes the number of “early” available columns, allowing queries to be awoken earlier. Another approach could prioritize columns that faster satisfy some query needs. Finally, if data is compressed on disk but kept decompressed in the buffer manager (like in SybaseIQ), it might be valuable to first load compressed columns, so their decompression is interleaved with loading the remaining ones.

#### 7.4.3 DSM results

Table 7.2 presents DSM results for an experiment similar to the one presented in Table 7.1 for NSM/PAX. One difference is that we used a faster “slow” query, since, due to the faster scanning achieved by DSM, with the original query the system was completely CPU bound, making it impossible to demonstrate performance differences of different policies with these queries. Also, we increased the lineitem size from factor 10 (60Mtuples) to 40 (240Mtuples), to compensate for the lower data-volume demand of DSM. Finally, since our current implementation requires reserved memory for each active chunk, and there are
Chapter 7: Cooperative scans

Table 7.3: Performance of DSM queries when scanning different sets of columns of a synthetic table

<table>
<thead>
<tr>
<th>Queries (used columns)</th>
<th>Normal</th>
<th>Relevance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>number of I/Os</td>
<td>query latency</td>
</tr>
<tr>
<td></td>
<td>avg.</td>
<td>stddev</td>
</tr>
<tr>
<td>Non-overlapping queries</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABC</td>
<td>5094</td>
<td>100.58</td>
</tr>
<tr>
<td>ABC,DEF</td>
<td>6215</td>
<td>121.83</td>
</tr>
<tr>
<td>Partially-overlapping queries</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABC</td>
<td>5094</td>
<td>100.58</td>
</tr>
<tr>
<td>ABC,BCD</td>
<td>5447</td>
<td>107.86</td>
</tr>
<tr>
<td>ABC,BCD,CDE</td>
<td>5791</td>
<td>113.26</td>
</tr>
<tr>
<td>ABC,BCD,CDE</td>
<td>6313</td>
<td>125.14</td>
</tr>
</tbody>
</table>

The first part of Table 7.3 shows the performance changes when query types

more chunks in DSM (one for each column), we had to increase the buffer size from 1GB to 1.5GB to allow concurrent execution of 16 queries.

The results confirm that also in DSM relevance is clearly the best scheduling approach. All policies behave as observed earlier in NSM: normal performs bad in both dimensions, while attach and elevator both improve the system throughput, with the former additionally improving query latencies. The relevance policy beats the competitors in both dimensions, only losing slightly to elevator on the slow full-table scan.

7.4.3.1 Overlap-ratio experiments

While so-far we have seen another success story of relevance, in DSM there is the caveat of column overlap. If queries have a significant percentage of overlapping columns, DSM provides good I/O reuse opportunities, which are then best exploited by relevance. In the following experiment, however, we investigate to what extent decreasing column overlap affects performance. We have performed a synthetic benchmark, where we run various queries against a 200M-tuple relation, consisting of 10 attributes (called A to J), each 8 bytes wide. The buffer size is 1GB. We use 16 streams of 4 queries that scan 3 adjacent columns from the table. In different runs, corresponding queries read the same 40% subset of the relation, but may use different columns. The total number of chunks for the entire run is 7680 chunks.

The first part of Table 7.3 shows the performance changes when query types
do not have any overlapping columns. With 16 parallel queries and one resp.
two query types, we thus have 16 resp. 8 queries of the same type (but with randomly chosen 40% scan ranges). Due to the rather large size of the scans, *normal* can still re-use quite a few blocks in case of a single query type (around 33% of the 7680 chunks), but about half of that is lost when two column-disjunct queries are used. As for *relevance*, very good re-use is achieved using a single query type, with *relevance* beating *normal* by a factor 4. With two query types, the average query latency doubles, which corresponds to the 0.5 reduction of sharing opportunities, but *relevance* still beats *normal* by a factor two there.

With non-overlapping query families, numbers are somewhat harder to un-
derstand, but the general trend is that I/O reuse drops with decreasing column overlap. As *relevance* normally benefits more from bandwidth sharing, it is hit more, relative to *normal*, but we still observe *relevance* beating *normal* by a factor two in these situations. These results confirm that the benefit of the *relevance* policy does depend on the columns used in the queries. This knowledge can be exploited by applications. For example, when looking for correlations in data mining, assuming thousands of queries are issued in the process and but only few are executing at the same time, it may be beneficial to schedule the queries such that the column overlap is maximized.

## 7.5 Cooperative Scans in a RDBMS

In this section we outline how existing DBMSs can be extended with Cooperative Scans, focusing on the *ABM* implementation and adapting order-aware operators to out-of-order data delivery.

### 7.5.1 ABM implementation

The most practical and least intrusive way to integrate Cooperative Scans into
an existing RDBMS is to put *ABM* on top of the standard buffer manager. We successfully created an early *ABM* prototype in PostgreSQL [ZBK04]. Here, to load a chunk, *ABM* requests a range of data from the underlying buffer manager. This request is fulfilled by reading multiple pages occupying random positions in the standard buffer pool. These pages, locked by *ABM* after reading, are provided to all interested CScan operators and finally are freed when *ABM* decides to evict them. An additional benefit is that *ABM* can dynamically adjust its buffer size in situations when the system-wide load changes, e.g. when the number of active CScan operators decreases. Also, if the buffer manager provides an appropriate
interface, it is possible to detect which pages of a chunk are already buffered and promote partially-loaded chunks in ABM.

Though the original focus of CScan was improving the scans of a single table, a production-quality implementation of CScan should be able to keep track of multiple tables, keeping separate statistics and meta-data for each (large) table in use. As our approach targets I/O bound situations, for small tables CScan should simply fall back on Scan.

Finally, ABM only improves performance on clustered scans. For unclustered data access, CScan should not be used. Still, ABM can exploit the queue of outstanding page requests generated by the normal buffer manager to prioritize chunks more as they intersect more with this queue. When the chooseChunk-ToLoad() decides to load a chunk, any intersecting individual page requests should be removed from the normal page queue.

7.5.2 Order-aware operators

In this section, we discuss the impact of the out-of-order delivery of tuples by CScan on query processing. In its purest form, the relational algebra is order-unaware, and this holds true for many physical operators (e.g. nested-loop join, scan-select, hash-based join and aggregation, etc.). However, query optimizers make a clear distinction between order-aware and unaware physical operators (e.g. by enumerating sub-plans that preserve certain “interesting orders”). The two major order-aware physical operators are ordered aggregation and merge-join.

Ordered aggregation exploits the key-ordering of the input for efficient computation of per-key aggregate results that can be immediately passed to the parent once a key change is detected. With Cooperative Scans, ordered aggregation can still exploit the fact that the per-chunk data is internally sorted. We pass the chunk number of a tuple as a virtual column via the Volcano-like operator interface of MonetDB/X100. When processing a given chunk, the operator performs inside-chunk ordered aggregation, passing on all the results except for the first and the last one in the chunk, as these aggregates might depend on the data from other chunks. These border values are stored on a side, waiting for the remaining tuples that need to be included in that computation. A key observation is that chunks are large, so not huge in number, and the number of boundary values to keep track of is limited by the number of chunks. Looking at the chunk sequence, it is also possible to detect the “ready” boundary values early and pass them to the parent immediately, which is especially useful with
multiple consecutive chunks delivered.

**Merge Join** can be handled in the *attach* and *elevator* policies as follows \[HSA05\]: at a moment when a scan starts on one table, a matching position in the other table is found, and join processes until the end of table. Then, the scan on both tables starts from the beginning, processing until the original position.

Since *relevance*’s data delivery pattern is much more dynamic, a more complex approach is necessary. In case the inner table fits main memory, it is enough to switch to a proper position in this table (using search or index lookup) whenever a chunk in the outer table changes. Similarly, with a multi-level storage hierarchy, including both fast but small solid state memory and large but slow disks, similar strategy can be applied if the inner table fits on a flash drive and the cost of finding a matching position in it is small.

Situation complicates when both tables need to be scanned from disk. After loading data from the outer table, it is necessary to enforce loading a matching range in the inner table, and synchronize their delivery to the reading operators. Since a chunk in the inner table can be useful for multiple chunks in the outer relation, the *loadRelevance* function for the outer table should be extended to take into account the availability of matching data in the inner table, optimizing its reuse. Clearly, this approach significantly complicates implementation, especially in multi-join scenarios, suggesting using traditional *Scan* operators in such cases.

There is one special, yet valuable, scenario where *CScan* can be applied on both sides of a merge join. MonetDB/X100 uses *join indices* for foreign-key relationships. For example, the join index over *orderkey* between *lineitem* and *order* in TPC-H adds the physical row-id *#order* as an invisible column to *lineitem*. By storing the *lineitem* table sorted on *#order* (and *order* itself sorted on *orderdate*), we get *multi-table clustering*, where tuples are stored in an order corresponding to the foreign key join relationship.

Within MonetDB/X100, there is ongoing work on *Cooperative Merge Join* (CMJ) that works on top of such clustered tables, fully accepting out-of order data as it is delivered by *CScan*. The key observation is that multi-table clustered DSM tables can be regarded as a single, joined, DSM table on the level of *ABM*, as it already has to deal with the fact that in DSM columns have widely varying data densities and chunk boundaries never coincide with page boundaries. Thus, *ABM* views the physical representation of the clustered *order* and *lineitem* table as the physical representation of the already joined result, even though the data density in the *order* columns is on average six times lower than in *lineitem*. Using the freedom to choose the boundaries of logical chunks at will, it makes
Chapter 7: Cooperative scans

Sure that matching tuples from order and lineitem always belong to the same chunk. Thus, a single CScan operator can deliver matching column data from both order and lineitem tables and the special CMJ merge-join reconstructs joined tuples from these.

7.6 Related work

Disk scheduling policies are a topic that originated from operating systems research [TP72]. Various such policies have been proposed, including First Come First Served, Shortest Seek Time First, SCAN, LOOK and many others. Most relevant for our work is SCAN, also known as the “Elevator” algorithm. In this approach, a disk head performs a continuous movement across all the relevant cylinders, servicing requests it finds on its way. Other related operating system work is in the area of virtual memory and file system paging policies, for which generally LRU schemes are used. Note that these solutions are mostly targeted at optimizing the disk seek time with multiple random disk accesses. In case of large sequential scans, these policies will offer very little improvement.

Previous research in DBMS buffer management [CD85, SS86, CR93, FNS91] usually considered large table scans trivial and suggested a simple LRU or MRU policy, which minimized the possibility of inter-query data reuse. To overcome this problem, the concept of circular-scans has been introduced in some commercial DBMSs, e.g. Teradata, RedBrick and Microsoft SQLServer [NCR02, Col98, Coo01]. A variation of this idea was suggested in [KSR01], where authors issue a massive number of concurrent request to the buffer manager and serve them in a circular fashion. It was also discussed as a part of the Q-Pipe architecture [HSA05]. All these approaches follow either the attach or elevator policies, which in Section 7.3 have been shown as inferior to the new proposed relevance policy. Recently, a modified version of the attach policy has been suggested for table scans [LBM+07] and index scans [LBMW07] in the IBM DB2 system. This solution introduces slight improvements to attach, by adding explicit group control and allowing a limited throttling of faster queries, but still suffers from the main attach problems.

Most previous work regarding table scans has focused on row storage only, ignoring scans over column-oriented data. A recent paper by Harizopoulos et al. [HLAM06] provides a detailed analysis of the I/O behavior differences between DSM and NSM. However, this paper concentrates on single-query scenarios and does not analyze the problem of the DSM buffer demand, which we found important, especially in a concurrent environment.
Scheduling is also important in real-time database systems [KGM95], where transactions need to be scheduled to meet certain time critical constraints. This involves making scheduling decisions based on the availability of certain resources, such as CPU, I/O, and buffer-manager space. Our work differs from such scheduling, in that we do not treat buffer-manager space as a resource being competed for, but rather schedule in a way to maximize sharing opportunities.

In multi-query optimization [SSB00], the optimizer identifies common work units in concurrent queries and either materializes them for later use [MPK00] or creates pipelined plans where operators in multiple queries directly interact with each other [DSRS01]. The concept of shared scans is often a base for such query plans [DSRS01]. When compared with our work, multi-query optimization is performed on a higher level, namely on the level of query processing operators that may be shared. Such operator sharing is even the cornerstone of the Q-Pipe architecture [HSA05].

A related approach is multi-query execution (rather than optimization). The NonStop SQL/MX server [Cle99] introduced a special SQL construct, named ‘TRANSPOSE’, that allows explicitly specifying multiple selection conditions and multiple aggregate computations in a single SQL query, which is executed internally as a single scan.

Ideas close to our algorithms have been explored in research related to using tertiary storage. Sarawagi and Stonebraker [SS96] present a solution that reorders query execution to maximize data sharing among the queries. Yu and DeWitt [YD97] propose pre-executing a query to first determine the exact access pattern of a query and then exploit this knowledge to optimize the order of reads from a storage facility. Moreover, they use query batching to even further improve performance in a multi-query environment. Shoshani et al. [Sho99] explore the multi-dimensional index structure to determine files interesting for queries and apply a simple file weighting based on the number of queries interested in it. This is a special-purpose system, while we attempt to integrate Cooperative Scans in (compressed column) database storage and query processing architectures.

Ramamurthy and DeWitt recently proposed to use the actual buffer-pool content in the query optimizer for access path selection [RD05]. This idea can be extended for Cooperative Scans, where the optimizer could adjust the estimated scan cost looking at the currently running queries.

All research discussed so far focused on improving disk data delivery performance. Modern multi-core CPUs with complex memory hierarchies result in RAM bandwidth becoming a bottleneck, as discussed e.g. in Section 6.2.9. As a
result, data-sharing algorithms are also applied on the CPU level. For example, in [QRR+08] queries running on different cores that share some level of cache reuse the data stored there, minimizing main-memory traffic.

7.7 Conclusions and future work

This chapter motivated and described the Cooperative Scans framework that significantly enhances existing I/O scheduling policies for query loads that perform concurrent (clustered index) scans. One area where this is highly relevant is data warehousing, but (index) scan-intensive loads are found in many more application areas, such as scientific databases, search, and data mining.

The Active Buffer Manager (ABM) coordinates the activities of multiple Cooperative Scan (CScan) queries in order to maximize I/O bandwidth reuse, while ensuring good query latency. We compared a number of existing scheduling policies (LRU, attach, elevator), and have shown that our new relevance policy outperforms them consistently.

We have shown the benefit of our approach in experiments using both row-wise storage (NSM or PAX) and column-wise storage (DSM). While column-stores have gained a lot of interest in recent years, we are not aware of significant previous work on I/O scheduling for column stores. One of our findings here is that DSM scheduling is much more complex, and efficient DSM I/O requires considerably more buffer space than NSM. Our new policy performs progressively better when buffer space is scarce, which plays to its advantage in DSM.

We described how ABM can be implemented on top of a classical buffer manager and also discussed order-aware query processing despite out-of-order data delivery, which is a topic of ongoing research.