Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

INS

Information Systems

INformation Systems

Cooperative scans

M. Zukowski, P.A. Boncz, M.L. Kersten

REPORT INS-E0411 DECEMBER 2004

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-3681

Cooperative scans

ABSTRACT

Data mining, information retrieval and other application areas exhibit a query load with multiple concurrent queries touching a large fraction of a relation. This leads to individual query plans based on a table scan or large index scan. The implementation of this access path in most database systems is straightforward. The "Scan" operator issues next page requests to the buffer manager without concern for the system state. Conversely, the buffer manager is not aware of the work ahead and it focuses on keeping the most-recently-used pages in the buffer pool. This paper introduces "cooperative scans" -- a new algorithm, based on a better sharing of knowledge and responsibility between the "Scan" operator and the buffer manager, which significantly improves performance of concurrent scan queries. In this approach, queries share the buffer content, and progress of the scans is optimized by the buffer manager by minimizing the number of disk transfers in light of the total workload ahead. The experimental results are based on a simulation of the various disk-access scheduling policies, and implementation of the "cooperative scans" within PostgreSQL and MonetDB/X100. These real-life experiments show that with a little effort the performance of existing database systems on concurrent scan queries can be strongly improved.

1998 ACM Computing Classification System: H.2 DATABASE MANAGEMENT; H.2.2 Physical Design
Keywords and Phrases: database management; query processing; table scans; buffer management

Cooperative Scans

Marcin Zukowski
Marcin.Zukowski@cwi.nl

Peter Boncz
Peter.Boncz@cwi.nl

Martin Kersten
Martin.Kersten@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT

Data mining, information retrieval and other application areas exhibit a query load with multiple concurrent queries touching a large fraction of a relation. This leads to individual query plans based on a table scan or large index scan. The implementation of this access path in most database systems is straightforward. The `Scan` operator issues next page requests to the buffer manager without concern for the system state. Conversely, the buffer manager is not aware of the work ahead and it focuses on keeping the most-recently-used pages in the buffer pool.

This paper introduces *cooperative scans* – a new algorithm, based on a better sharing of knowledge and responsibility between the `Scan` operator and the buffer manager, which significantly improves performance of concurrent scan queries. In this approach, queries share the buffer content, and progress of the scans is optimized by the buffer manager by minimizing the number of disk transfers in light of the total workload ahead.

The experimental results are based on a simulation of the various disk-access scheduling policies, and implementation of the *cooperative scans* within PostgreSQL and MonetDB/X100. These real-life experiments show that with a little effort the performance of existing database systems on concurrent scan queries can be strongly improved.

1. Introduction

Database technology applied to data mining, OLAP, multimedia and information retrieval and other areas has to deal with queries that process an entire relation (or a large fraction of it). The relation width and low predicate selectivities often make the query optimizer resort to table scans. Buffering of scan queries has been considered solved, using a simple LRU policy [CR93, SS86]. This strategy is efficient for isolated queries. However, either due to a concurrent environment (multiple users) or domain-specific strategies (e.g. search-space exploration), modern applications often exhibit a query load consisting of *multiple* concurrent scan queries. In this situation, a traditional DBMS scan buffering algorithm causes queries to issue conflicting disk requests, which severely hurts performance.

This paper introduces the *cooperative scans* algorithm, which can significantly improve the response time in these application areas. In this approach, `CScan` – a modified `Scan` operator, announces the pages needed upfront, and an *active buffer manager* (ABM) optimizes the order of disk accesses taking into account all `CScan` requests. Since the semantics of the table scans often do not require a particular order of data processing, this allows multiple running queries to process the same set of tuples at a given time. Additionally, it reduces the number of concurrently issued I/O requests, resulting in improved disk bandwidth. Several policies for scheduling `CScan` requests are discussed to find the algorithm providing the optimal performance.

The *cooperative scans* idea can be readily included in existing DBMSs. This requires an addition of a physical relational algebra operator to the standard repertoire and a buffer manager modification such that it can be informed of the page set needed in advance. The technique is illustrated with an implementation in PostgreSQL [Pos] and a high-performance query processing engine based on the open-source DBMS MonetDB [BZN05].

The rest of this paper is organized as follows. Section 2 describes example scenarios with concurrent scan queries and analyzes the performance of such query loads on modern DBMSs. Section 3 introduces different variants of *cooperative scans* and presents detailed simulation results. The importance of using large-chunk exclusive I/O is also discussed. Section 4 explains in detail how the recommended *"relevance"* scheduling policy can be incorporated into a DBMS buffer manager. In Section 4.2 we expand the scope of this analysis by considering non-scan access paths (index-scans and random requests). The validity of our approach is shown in Section 5 where we present results of real-life experiments with PostgreSQL and MonetDB/X100. Section 6 discusses related work and we conclude in Section 7.

2. Motivation

This section presents typical application areas which generate heavy loads of scan-based queries, and demonstrates the performance problems these pose for database systems.

2.1 Scan-based applications

Database technology continues to pervade into new application areas, which, while posing relatively small (comparing to e.g. OLTP) number of queries, require processing much larger data volumes for each of them. In this situation, the importance of index-supported OLTP usage scenarios diminishes with respect to the scan-based analysis-oriented scenarios. A typical example of such situation are *data-mining* applications, such as presented by the DD Benchmark [BRK98]. This benchmarks tests DBMS performance while executing queries for a data-mining task in which a decision tree is incrementally induced using a *beam search* strategy. Each step k of the beam search generates a batch B_k of *cross-table* requests:

$B_k = \{$ `SELECT` c_i, c_{target}`,COUNT(*) FROM table WHERE` $pred_g$ `GROUP BY` c_i, c_{target} $|\forall g \in G_{k-1}, \forall c_i \in$ `table` $\}$.

The selection predicates $pred_g$ *(1)* are typically conjunctive range-expressions on any of the attributes c_i and *(2)* yield at least some thousands of tuples (for statistical confidence). Property *(1)* implies that a single clustered index cannot be used, and *(2)* implies that unclustered indices will not be beneficial either, such that the hundreds of cross-table requests generated by a single data mining task must be handled with scans.

Another example is *content-based multimedia retrieval*, e.g. a web server that allows people to retrieve images similar to the provided one from a large picture collection [MH04]. KNN-search over multimedia requires searching a high-dimensional space such that indexing becomes hard (the "curse" of dimensionality) and sequential scans are sometimes the best option [BGRS99]. Similar issues arise when databases are used to store *genome data* and free-form subsequence match queries are to be executed, which also results in sequential scans [AMS$^+$97]. Also in *ad-hoc OLAP* environments, users may pose aggregate queries on large fact tables that cannot be answered with predefined materialized views or precomputed data cubes.

All discussed application types, while differing in the performed tasks, put similar requirements on DBMSs. In the next section we analyze how these requirements are met.

2.2 DBMS performance on SCAN queries

While there have been multiple approaches suggested to buffer page replacement (e.g. [CR93, CD85, NFS91, SS86]), most of them concentrated on optimizing random accesses. For scan-based queries, usually a simple LRU replacement policy with limited caching is used.

Table 1 presents results of an experiment showing that current DBMS technology does not handle concurrent scans well. Three database systems (MySQL [MyS], PostgreSQL and one of the major commercial players), were filled with the `lineitem` table from the TPC-H benchmark [Tra02] with scale factor 5 (ca. 30 millions tuples). All tests were conducted on a dual AthlonMP system with 1.5GB of main memory running Linux with a 5-disk (100GB 7200RPM SATA) RAID-5 [PGK88] storage. The following scenarios were tested:

- reading the database data with a simple file scan

	MySQL		PgSQL		DB-Y	
File scan	501		285		318	
Standalone	297		209		267	
	Q1	Q2	Q1	Q2	Q1	Q2
Synchronized	263	263	194	194	256	256
De-synchronized						
average	204	225	115	121	171	171
concurrent	157	185	78	84	131	131

Table 1: Simple scan experiment results (in thousands tuples per second)

year	model	size (GB)	seek (ms)	bandw. (MB/s)	cache. (MB)
1994	Seagate Elite	1.25	10.0	2	0
1996	Quantum Empire	4.5	9.5	3.6	0
1998	Quantum Fireball	6.4	10.0	14	0
2000	Quantum Atlas10KII	18.4	7.9	40	0
2002	WD Caviar	200	8.9	60	8
2004	Maxtor MaxLineIII	300	9.3	65	16

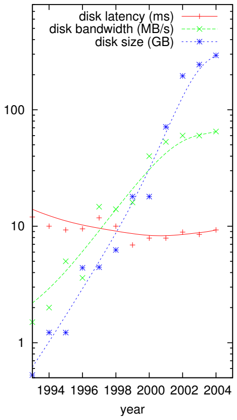

Figure 1: Disk hardware characteristics over the last decade

- a single TPC-H Query-6, standalone.
- two such queries in parallel, starting at the same time.
- two such queries in parallel, the second starting when the first is halfway through [1].

Query 6 was chosen as it is a simple scan query that requires little CPU effort in all tested DBMSs, making it disk-bound. This query could be implemented using an index scan, but no attribute is selective enough to choose an unclustered index plan over scan, and assuming the existence of an index over *all* used attributes is not reasonable.

Table 1 shows that all tested systems, while differing in absolute performance, follow the same tendency. The first observation is that the standalone performance is significantly lower than raw file scan. This shows that DBMSs do not fully exploit the available bandwidth. When two queries start at exactly the same time, performance is in line with single query performance. Since queries are at the same stage of processing, they can share the data that is in the buffer, resulting in only a single series of disk read requests. However, if the queries start at different times, buffer sharing is not possible anymore and the *average* execution speed severely suffers on all platforms. This illustrates the first problem we will address in Section 3, i.e. queries do not exploit the data available in the buffer.

Another problem Table 1 illustrates is that *concurrent* performance is sometimes significantly smaller than half the sequential speed. This is caused by both queries issuing conflicting requests resulting in additional disk arm movements. Figure 1 shows that hardware improvements actually make this problem more severe. Over the last decade, the sequential bandwidth of disks steadily grew,

[1] *concurrent* speed is estimated by taking half of the tuples divided by (total time minus half of time of sequential run)

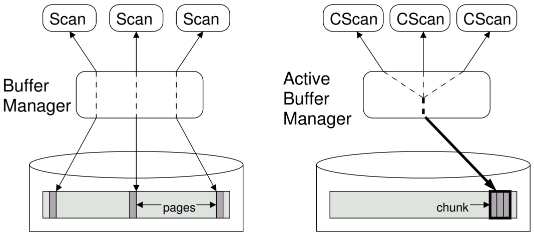

Figure 2: Scan processing using traditional strategy and Cooperative Scans

while seek time, in large part determined by disk arm movements, improved very little. We address this issue in Section 3.2, where we demonstrate the importance of using *exclusive I/O*.

3. Cooperative Scans

The problems identified in the previous section are mainly caused by the fact that queries always keep to the sequential order of processing pages in the table, ignoring the data that is in the buffer but is required further in the execution process. Only in a lucky situation where there was a similar scan performed recently, a query may "follow" the other one reusing already fetched data (cached either by the buffer manager or the operating system).

Figure 2 illustrates the modification to the traditional DBMS architecture that attempts to solve this problem. We introduce two new ideas: *(i)* a new version of the `Scan` operator and *(ii)* a modification of the traditional buffer manager.

The new **CScan** operator registers itself as an *active scan* on a particular table, accepting data as it comes, instead of explicitly asking the buffer manager for particular pages. `CScan` has much the same interface as the normal `Scan` operator, but it is willing to accept that data may come in a random order. Note that some database systems may generate query plans that exploit column ordering when scanning tables stored as clustered B-trees ('index-scans'). Such query plans should stick to using the normal `Scan` and not the new `CScan`. We discuss integration of such (and other non-scan) queries in our framework in Section 4.2.

The **Active Buffer Manager** (`ABM`) extends the traditional buffer manager in that it keeps track of `CScan` operators and which pages have already been processed by each scan, and tries to *schedule* disk reads such that multiple concurrent scans reuse the same pages. The overall goal of the `ABM` is to minimize average table scan cost, keeping the maximum query execution cost reasonable (i.e. ensuring "fair" treatment of all queries). Table 2 lists different `ABM` parameters used further in this article.

The `ABM` reads and caches data in the granularity of *chunks*, which are (much) larger than pages. There are two reasons for introducing chunks. The first is that scans must produce good bandwidth while the order in which the data is fetched might be random, as determined by one of the scheduling policies (in Section 3.2, we investigate how a chunk size influences disk read performance). The second reason is that there are typically two or three orders of magnitude fewer chunks than pages. Thus, it becomes possible to have chunk-level scheduling policies that are considerably more complex than page-level policies.

3.1 Scheduling policies

One of the responsibilities of the `ABM` is to determine the most efficient order of disk fetches to satisfy all the currently running queries. To show the characteristics of the various *scheduling policies* that can be applied to that task, we developed a simulation tool that is described first.

Simulator description Our simulator models a "perfect" disk where each (equi-sized) chunk fetch has a fixed cost, regardless the order of chunk fetches. This is of course not true in real life due to disk access locality and its effect on latency. However, in Section 3.2 we will explain how this problem can

B	the size of a buffer page (e.g. 8KB)
N	the number of pages in the buffer (e.g. 16384, a 128MB buffer pool)
C	the number of pages in a chunk (e.g. 1024, giving 8MB chunks)
Q	the number of all concurrent scans
Q_T	the current number of concurrent scans on a table T
Q_{lim}	the maximum allowed number of scans on a single table
M_T	the number of chunks in a table T
S	the number of currently cached chunks

Table 2: Active Buffer Manager parameters

be minimized with chunk-level exclusive I/O. Also, when multiple queries are running on the CPU at the same time, their speed is assumed to be exactly inversely proportional to their number. This is a simplification, as e.g. CPU cache effects might increase or even decrease the CPU cost of concurrent queries with respect to the single-query cost.

The simulator models a CPU with a 2GHz clock and a hard disk with a 50 MB/s throughput. Queries are defined as processes that need various numbers of cycles per byte (c/b) of data. This means, for the example settings, a perfect query (with CPU and disk requirements balanced) should process data at a rate of ca. 40 c/b. A lower c/b factor means the query is disk-bound, while a higher c/b factor means it is CPU-bound. The simulated dataset is a 100MB table divided into twenty 5MB chunks. The size of the buffer is 6 chunks. These low example values are chosen only to keep the performance graphs readable.

To show differences between various scenarios, two test-cases were used:

HomogeneousRun - 3 copies of the same query that spend 20 c/b, with a 0.5 second delay between them (Figure 3)

HeterogeneousRun - 3 different queries. Query 1 uses 45 c/b and starts at 0.0s. Query 2 processes 5 c/b and starts at 0.5s. Query 3 processes 25 c/b and starts at 1.0s (Figure 4)

Each graph presenting simulation results contains two areas. The upper part displays events related to specific chunks. Dark rectangles represent the time when a given chunk is being fetched from the disk. White rectangles show the time when a chunk is being stored in the cache. Lines inside white rectangles show the time spent by particular queries on processing that chunk.

The bottom part represents the state of global entities, with the following areas: the time that queries spend waiting for I/O, total execution time of the queries, CPU usage (number of queries processing data) and finally disk usage (number of currently running disk requests).

The "Attach" Policy There are situations, where the DBMS needs to execute concurrent queries that have very similar CPU performance and disk bandwidth requirements. This is not an uncommon scenario if the database is a back-end to a web server for e.g. multimedia search or pattern matching on a genome database. Such applications often feed the DBMS with a set of canned scan-based queries that differ only in their bound parameters.

If all such queries are issued at the same time, it is probable that they will go over the table with the same speed. In this situation, they will share the disk access cost, resulting in good performance. However, if queries come in at different times, they will ask for different parts of the table concurrently, and will start to fight for disk bandwidth, as shown in the upper picture of Figure 3.

For this class of problems, the *"attach"* policy starts processing each new query not at the beginning, but at the most recently fetched chunk, until reaching the end of the table, and then continue from

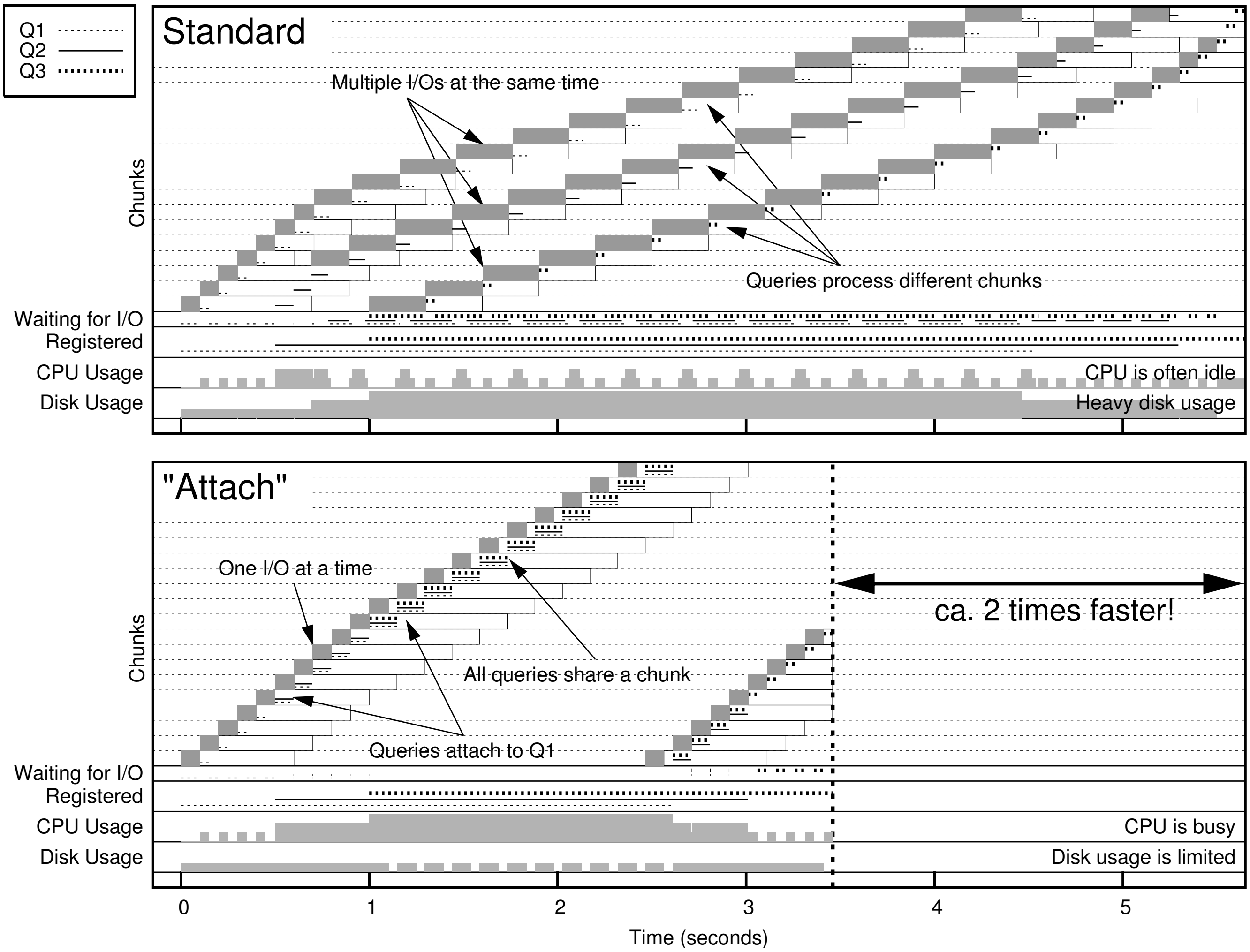

Figure 3: HomogeneousRun Simulation

the beginning until the initial point. The lower picture of Figure 3 shows the execution timeline of our `HomogeneousRun`. All queries obtain good performance thanks to sharing the chunks.

However, if the queries do not proceed with the same speed, they may get "desynchronized". The top part of Figure 4 presents what happens for `HeterogeneousRun`. At the beginning, queries share data that is in the buffer, but once they are not synchronized anymore, they start to issue conflicting requests to the disk, such that chunk sharing is lost, and the number of chunk loads needed to satisfy the query batch goes up.

The "Elevator" Policy The *attach* policy can be refined to work well with unbalanced queries, by forcing faster queries to wait for slower ones. This can be done by keeping a "sliding window" of currently buffered chunks, and allowing a request for the next chunk only if the last chunk in this window does not have any queries currently processing it. The middle part of Figure 4 shows that it maximizes chunk reuse and improves system throughput.

This *"elevator"* approach has one important drawback: quick queries spend significant amounts of time just waiting for other queries. This means that while the average response time for the slowest

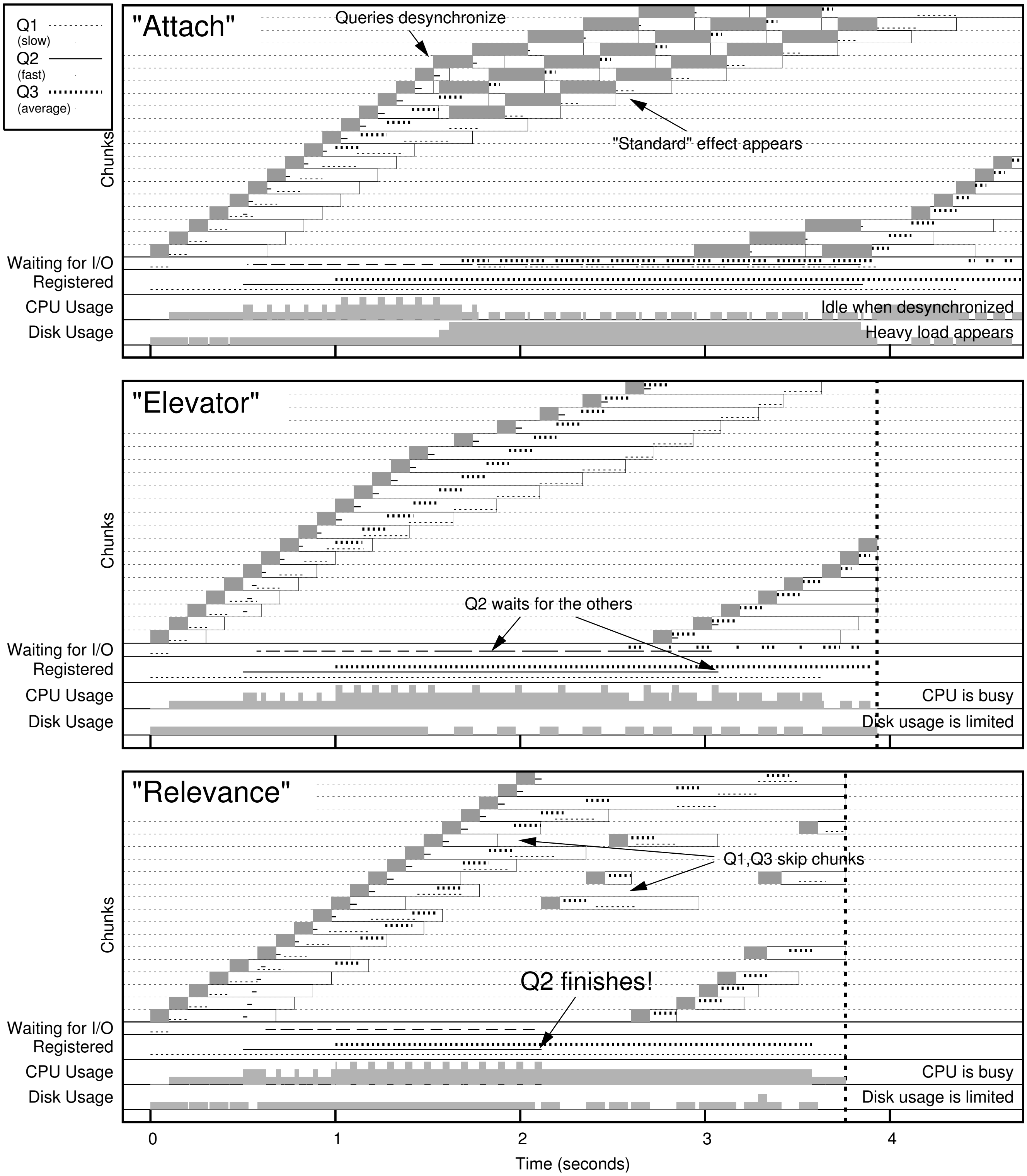

Figure 4: HeterogeneousRun Simulation

type of queries is minimized, all queries get this response time. Thus, there is an opportunity to further improve average response times by better servicing quick queries.

The "Relevance" Policy The *"relevance"* policy generalizes chunk scheduling by computing for each chunk two *relevance functions*. Chunk-selection is triggered whenever a query $q_{trigger}$ finishes processing a chunk. Then, for all chunks in the table $fetchRelevance(c, q_{trigger})$ is computed, and if this function is positive on any chunk, the ABM loads the chunk with the maximum score. Also, if the chunk cache is already full, a cached chunk with the lowest $keepRelevance(c, q_{trigger})$ score is chosen for eviction.

Query and Chunk Features There are various features of each active query (q) and table chunk (c) that can be taken into account for the relevance functions, including:

$availableChunks(q) : \{c\}$ - The set of chunks a query has not processed yet and that are in the cache. For example, queries that still have many chunks available may be ignored.

$desiredChunks(q) : \{c\}$ - the set of chunks that a query still needs to process. Analogous to what was found in [AGM89], queries that are almost finished may be promoted.

$priority(q) : int$ - chunks that benefit queries with a high priority [CJL89] might get loaded earlier.

$startTime(q) : int$ - algorithm may explicitly try to minimize maximum query response time (i.e. select chunks that benefit the longest running query).

$cached(c) : bool$ - true if the chunk is currently cached.

$processingQueries(c) : \{q\}$ - the set of queries that is currently processing this chunk. If this set is non-empty, the chunk cannot be evicted in any policy.

$interestedQueries(c) : \{q\}$ - the set of queries that still want to read this chunk. The larger this set, the more relevant the chunk might be.

$seekDistance(c) : int$ - the absolute difference in chunk sequence number with respect to the last chunk fetched by the ABM. Chunks with a low value might be preferred, as this might reduce chunk fetch cost.

This approach is highly extensible with many possible relevance functions exploiting other application-specific chunk and query features. In the following, however, we recommend functions that we think fit for application in a generic DBMS, as we expect them to perform well in most circumstances.

We should note that computing relevance functions on all chunks leads to a complexity:

$$O(M_T * cost(fetchRelevance()) + S * cost(keepRelevance())$$

This is quite expensive. Recall, though, that with $C = 1024$ a linear complexity in the number of chunks might be acceptable. In any case, for our "recommended" functions in Section 4 we propose an implementation with complexity independent of the number of chunks.

Recommended "Relevance" Functions The goal of the functions we recommend here is to (strictly) minimize the average query response time. That is, we do not take into account query priorities or deadlines.

Our *fetchRelevance()* determines which queries are "starved" (i.e. do not have any interesting chunk cached) and selects a chunk that benefits the highest number of "starved" queries, including the query that triggered chunk-selection, if that one is "starved". Forcing a starved triggering query

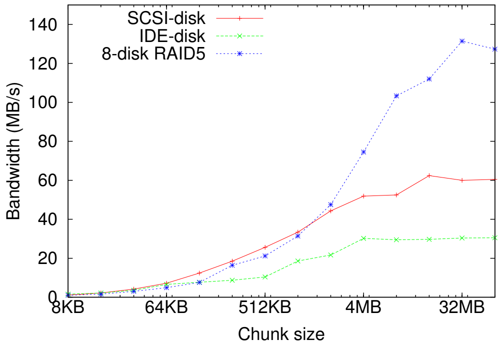

Figure 5: Bandwidth vs Chunk Size

to be selected ensures that each query receives a chunk once in a while. If there are no starved queries, we select the chunk that interests most queries.

Before a new chunk can be loaded, a slot needs to be prepared in the chunk cache. The *keepRelevance* function needs to ensure that a chunk chosen to be evicted is not being processed by any query. Additionally, chunks that would increase the number of "starved" queries should stay in the cache. If the chunk cache size S is at least $2Q_T$ it is certain that there exists such a chunk.

The proposed relevance functions are defined as follows:

$fetchRelevance(c_{cand}, q_{trigger})$ ::=

- 0, if $availableChunks(q_{trigger}) = \emptyset \wedge$ $\neg q_{trigger} \in interestedQueries(c_{cand}))$.
- $|\{q|\forall q : availableChunks(q) = \emptyset \wedge$ $q \in interestedQueries(c_{cand})\}|$, otherwise.

$keepRelevance(c_{cand}, q_{trigger})$ ::=

- ∞, if $processingQueries(c_{cand}) \neq \emptyset$.
- Q_T, if $\exists q_{req} : availableChunks(q_{req}) = \{c_{cand}\}$.
- $|interestedQueries(c_{cand})|$, otherwise.

The lower part of Figure 4 shows execution of `HeterogeneousRun` with this approach. Comparing the results with the *elevator* results, the slow queries take more or less the same amount of time to complete, but query 2 (the quickest one) finishes much faster.

3.2 Chunk-based I/O

`ABM` performs I/O requests using chunks instead of pages. In order to determine a good chunk size, we performed disk microbenchmarks on comparable Linux machines with the `ext3` filesystem. It consists of a series of random accesses to the chunks from a very large (non-cached) file with the following hardware:

IDE disk IBM DeskStar 7200RPM 40GB (a typical desktop PC drive).

SCSI disk Seagate Cheetah 10000RPM 36GB (a typical server drive).

	random page		chunk			
	num	num	read		write	
	reads	writes	num	BW	num	BW
single	1	1	1	1	1	1
RAID-0	X	X	1	X	1	X
RAID-1	X	$X/2$	2	$X/2$	1	$X/2$
RAID-5	X	$X/2$	1	$X-1$	1	$X-1$

Table 3: Chunk and Page I/O Concurrency and Bandwidth for RAID systems consisting of X disks

RAID-5 disk 3WARE 7810 hardware RAID-5 [PGK88] with SCSI interface, internally consisting of 8 SATA drives (7200RPM), configured with 64KB RAID stripes.

Figure 5 shows a trend of increasing performance with larger chunk sizes over all drives. After a certain point, optimal bandwidth is reached beyond which increasing the chunk size does not improve the performance anymore.

This is explained as follows. The cost of fetching a low-level disk block is the sum of access time and the sequential read time. Figure 1 shows that while the latter improves constantly, the former did not change significantly over the last decade. The reasons for improvements in sequential disk bandwidth were the increases of disk rotation speed and (especially) the density of disks. On the other hand, the seek time depends more on mechanical phenomena, like acceleration power and precision of the motor that steers a disk head, hence it is harder to improve.

Bandwidth close to the optimal can only be achieved, when read time strongly dominates over access time. This happens when the disk controller gets a number of consecutive low-level block requests for which it does not have to change the position of the disk head. Such requests are submitted to the disk controller only when a program issues a large chunk read to the OS.

This fully explains the behavior of Figure 5. The optimal bandwidth of the IDE drive is reached at a chunk size of 4MB, and for the SCSI drive this is 8MB. The 8-drive RAID-5, however, needs large 32MB blocks. This implies that each individual drive in the RAID needs a 4MB chunk read to perform optimally (just like we measured).

Exclusive I/O A related issue is how page and chunk requests are submitted. In OLTP scenarios, it may often be beneficial to let each query issue its page (single-block) request to the disk subsystem concurrently. This means that the disk device will get a batch of block requests. SCSI disks then reorder those requests, so a single disk head movement serves multiple such requests, minimizing the average seek time. A second benefit involves RAID subsystems, which are often used in database configurations to increase fault tolerance and performance. The RAID controller partitions the set of outstanding block requests into subsets of block requests for each different disk in the RAID, combining the first benefit with the additional benefit of improved throughput by processing requests to different disks in parallel.

Table 3 shows the capacities of the most used disk configurations: single disks and RAID, for which we look at the most relevant levels RAID-0 (block striping), RAID-1 (block striping, with one mirror for each block) and RAID-5 (one parity block per $X-1$ disks, with the parity blocks distributed between all disks).

The first two columns of Table 3 show that between $X/2$ and X *page* (single-block) requests can be handled in parallel, depending on the disk subsystem configuration and whether it is a read or write request. The other columns show the situation for *chunk* I/O, which is geared towards achieving high bandwidth. While the results show that a high bandwidth is possible, the level of concurrency is very limited. Therefore, we propose that the `ABM` serializes each chunk read to the same disk subsystem with respect to any other buffer I/O to that subsystem.

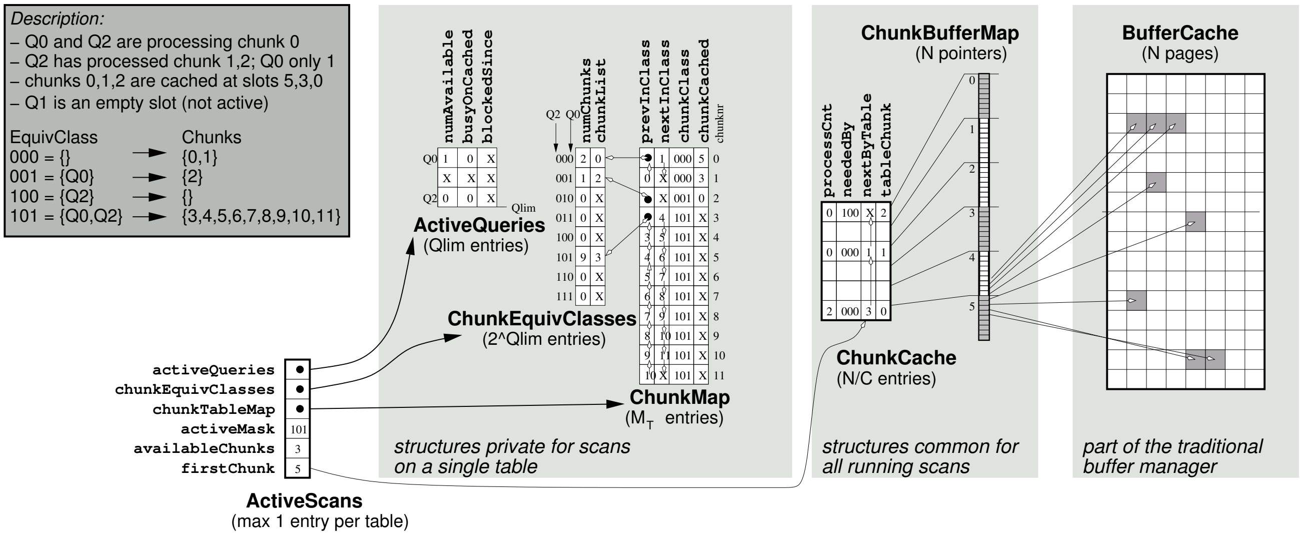

Figure 6: Implementation of Active Buffer Manager

4. ABM Implementation

This section demonstrates how *cooperative scans* can be integrated into the buffer manager of a DBMS. The goal is to improve the performance of queries that depend on scans, without hurting the performance of queries that do not. Also, we would like the modification to be as unintrusive as possible, in order to facilitate the introduction of *cooperative scans* into an existing DBMS. Therefore, we take care to leverage the buffer manager infrastructure found in a typical DBMS. This allowed us to add *cooperative scans* to PostgreSQL in less than two weeks (see Section 5).

This section concentrates on implementing the *relevance* policy introduced in Section 3.1, as the simulations in Section 3 clearly show it to be the most versatile and high performance scheduling strategy. We also discuss how non-scan queries can be taken into account with that approach.

4.1 A Fast "Relevance" Algorithm

All obvious algorithms, including the one suggested in Section 3.1 have complexity linear with respect to the number of chunks. While this number is surely much smaller than the number of page-chunks, it can still be substantial (thousands or more) for very large databases. To reduce the number of computations, chunks may be grouped into *equivalence classes* - groups of chunk that are interesting for exactly the same set of queries. The algorithm presented in this section has a complexity linear with respect to the number of equivalence classes, hence $O(2^{Q_T})$, regardless of the number of chunks. As most DBMSs will not be able to sustain a great many concurrent queries anyway, it seems reasonable to assume the amount of concurrent scans to not more than a few Q_{lim} (≤ 16) queries on the same table (or enforce this using queuing).

ActiveScans data structures Figure 4 presents the basic data structures that the `ABM` adds to an existing relational buffer manager. For each table that is being scanned, the buffer manager creates a record in `ActiveScans`. Each active scan maintains the following information:

- `availableChunks` - a count of how many chunks for this scan are currently cached.
- `firstChunk` - a link to the list of these chunks.

- `ActiveQueries` - whenever a `CScan` starts on a table, it is assigned a free slot in this array. Its `numAvailable` field contains the number of `availableChunks` that are actually interesting for this query (it may already have seen some of the cached chunks). Its `busyOnCached` field tells which cached chunk is currently being processed (if any) by this query. The `blockedSince` tells whether and how long a query is waiting for an available chunk.
- `activeMask` - a bit-mask defining which queries are currently running.
- `ChunkMap` - we store information for each chunk of the table in the `ChunkMap` array that is indexed by chunk number. Given an interest in a certain chunk c, we can look up information about it in $O(1)$. If the `chunkCached` field contains a valid `ChunkCache` index number, the chunk is cached there.

  Chunks are said to belong to the same *equivalence class* if they have the same query bit-mask, that is, the same set of queries is interested in them. For each chunk, we keep this set of interested queries in a query bit-mask `chunkClass`. The fields `nextInClass` and `prevInClass` maintain a doubly-linked list that connect chunks of the same equivalence class.
- `ChunkEquivClasses` - an array of $2^{Q_{lim}}$ pointers. For each non-empty equivalence class, this holds a chunk pointer into `ChunkMap`, that is the first element of the above mentioned doubly-linked list of all chunks in that equivalence class. By indexing this array with a bit-mask, we can thus determine in $O(1)$ whether there is a chunk in a certain equivalence class (i.e. a particular set of interested queries).

ChunkCache datastructures The `ChunkCache` represents the currently cached chunks for all scans (possibly on various tables). It has the following columns:

- `processCnt` - defines how many queries are currently *processing* the pages of this chunk.
- `neededBy` - a bit-mask that tells which queries have not yet started processing this chunk but still want to
- `nextByTable` - a linked list between `ChunkCache` entries that contains all chunks of the same table.
- `tableChunk` - tells which chunk (sequence number in a table) is cached here.

The `ChunkBufferMap` array is used to map each chunk onto C buffer pages, located anywhere in the buffer pool. Note that pages do not need to be contiguous in order to perform chunk reads, as we can use *scatter/gather* I/O calls (`readv()` on UNIX systems). A scatter/gather read provides a list of buffers (pages) and file offsets to the OS, which reads one big chunk, whose data is then put into these non-contiguous buffers. This allows us to play some tricks, e.g. if some pages in a chunk are already cached and dirty, we can redirect the data read from those (stale) disk pages to one "dummy" page to be ignored, and afterward put the references to the already cached pages in the `ChunkMap`.

Only a subset of the `ChunkCache` needs to be filled at one any time. Free slots mean that there are buffer pages allocated under the normal buffer manager policies.

Algorithm details There are various situations the scheduling algorithm needs to handle:

ABM initialized - only `ChunkCache` and an array storing multiple `ActiveScans` need to be allocated.

new table scanned - `ActiveScans` and all table-related structures (`ActiveQueries`, `ChunkEquivClasses`, `ChunkMap`) need to be initialized

query enters - a new query is added to `ActiveQueries` for a given table and `activeMask` is updated. All entries in `ChunkEquivClasses` need to be updated, as equivalence class changes for all the chunks.

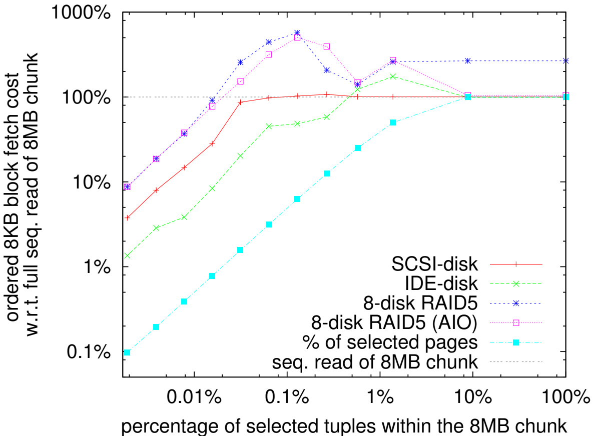

Figure 7: Sequential Scan vs. Ordered Index Scan, depending on selectivity

chunk released - when a query Q_i finishes processing a chunk, the proper bit in `neededBy` is flipped and `processCnt` is decreased. If `processCnt` becomes 0, it is possible to free the pages from `BufferCache`. It is also necessary to update the lists of both old and new classes in `ChunkMap` and adjust the respective values in `ChunkEquivClasses`.

new chunk required - when a query asks for a new chunk, first `ChunkCache` is checked if it contains a chunk with matching `neededBy` value. If so, the query starts processing that chunk and `processCnt` is increased. If not, the query sets its `blockedSince` timestamp and blocks until chunk becomes available.

fetching a chunk - when no I/O is in process, it is possible to fetch the next relevant chunk. The fetching process chooses a query that is waiting the longest and traverses `ChunkMap` to find the chunk class that is not empty, satisfies this query and maximizes the number of other "starved" queries (our *relevance* policy). The algorithms first examines a class that satisfies all running queries and successively tries masks with 1, 2 and so on queries removed from the mask (except for the "triggering" query). It continues until a non-empty equivalence class is found. If no queries are waiting, a chunk class that satisfies most queries is chosen. A first chunk from the chosen class is scheduled for I/O. If necessary, we need to free one of the slots in `ChunkCache`

freeing a chunk slot - when no slots in `ChunkCache` are available there are two choices. First, `BufferCache` can be asked for a new collection of pages to create a new cache slot. Second, one of the chunks can be evicted. To do that the entries of the current table in the `ChunkCache` are processed and a chunk with the lowest *keepRelevance()* value is chosen. It is also possible to remove entries of other tables.

In all situations, our algorithm performs at most $O(2^{Q_{lim}})$ steps. The most often executed step (determining a chunk to be fetched) executes in $O(2^{Q_T})$ steps in the worst case.

4.2 Using the ABM for non-scan queries

Until now, the discussion of our framework was limited to pure scans. We now first extend its use to index scans, and then discuss how other relational operators can benefit from the ABM infrastructure.

Index Selections A query that uses an unclustered index (B-tree) to evaluate some selection predicate typically yields a list of `rids` (record ID-s). The B-tree provides these `rids` in the order of the attributes included in the predicate. If the number of `rids` is substantial, many DBMSs use the policy of sorting the `rids` in order to improve the access pattern in the next phase, which is record-fetching by `rid`.

In an unclustered index, the selected records tend to be uniformly distributed over the relation. In Figure 7, we examine *per chunk* of 8MB what is the I/O cost of fetching individual pages in *sorted* order of `rids` with respect to the percentage of selected tuples.[2] All costs are normalized with respect to the cost of sequentially reading a full 8MB chunk.

Thus, our experiment consists of repeated synchronous seek-forward, read-page actions. Obviously, if multiple (subsequent) `rids` need the same page, it is fetched only once. In the case of our database server platform with the RAID-5 system, we had Linux kernel 2.6 installed, which supports *asynchronous I/O* and the `lio_listio()` system call[3]. This call takes a list of file positions and page buffers, such that in one system call the disk receives our list of desired pages.

The results clearly show that page fetch costs quickly surpass sequential chunk access cost, in the case of RAID and SCSI disks well before 0.1% of selected tuples. This corresponds to about a half percent of selected pages (i.e. about 5 8KB pages from the 8MB chunk). Since this break-even point is determined by the balance between disk bandwidth and latency, it is likely that it will shift towards even lower percentages over time. We predict that in a few years, individual page fetches will only be useful for (almost) single-record access, such as equi-selections on a (foreign) key.

The results suggest that the benchmarked SCSI disk switches automatically from seek/read/seek mode of operation to a sequential scan, such that its performance does not (significantly) degrade from sequential scan performance. The RAID subsystem shows that even with the `lio_listio()` the penalty of random access is relatively high from the start. Its random access performance at low selectivities is actually not worse than a single disk, but because a RAID disk provides high sequential bandwidth, its performance penalty is relatively higher.

We conclude from this experiment that if an DBMS detects an index selection with a selection percentage of 0.1%[4] or more, the resulting I/O will be as costly as a sequential scan. Thus, it can just as well use the *cooperative scan* infrastructure, register itself as a scan, and as chunks come in "cherry pick" only the pages it needs. If *cooperative scan* is aware that *all* queries interested in a chunk are not normal scans but only index selections with (sorted) page lists attached to them, it can reduce its memory consumption by making a union of all wanted pages in a chunk, and have the array passed to the scatter-`readv()`, containing a pointer to the same "dummy" page for the unwanted pages.

Other Queries We should note than scan and index scan are two main operators for accessing relations stored on disk. That is, other query processing operators, such as *Aggregation*, *Projection*, *Hash Join* and *Scan Select* all access their input data through a sequential scan, and hence can benefit from `CScan` as well. In the case of *nested-loop join*, the inner relation is scanned in a *looping sequential* [CD85] fashion. If there are multiple such operators running on the same inner table, each repeated scan can be treated as an independent `CScan`, since the order of the data from the inner relation is not relevant. Obviously, the outer relation can be serviced by a `CScan` as well. Similarly, for *Nested-loop Index Join* and *Index Select*, which use index scans, we can also retrieve their data by-the-chunk via the ABM (as discussed in the previous section), if the page selection percentage of the index scan is estimated to be sufficiently high.

Even if the page selection percentage is low, as in single-page queries found in OLTP loads, using the ABM may sometimes be beneficial. The simplest idea to benefit from the ABM is to also take the number of single-page queries that fall in a chunk into account in the chunk relevance criteria, such that chunks are selected in a way that helps the current OLTP query load. One complication of mixing OLTP queries with large scans is that OLTP throughput on a RAID system is optimized by issuing multiple page requests concurrently, while sequential throughput is optimized using exclusive bulk I/O (Section 3.2). These conflicting interests may cause the OLTP throughput to drop below the

[2] We use 50 bytes tuples, and 8KB pages here, such that 150 tuples fit on a page.

[3] In Linux `listio` seems to be implemented efficiently only for up to 16 combined requests.

[4] depending on the tuple size

required OLTP query rate when the ABM is performing bulk exclusive I/O. In those cases, it seems best to alternate between exclusive bulk I/O and concurrent single page I/O periodically.

5. Experiments

To evaluate our proposal in a real-life environment, we implemented *cooperative scans* in PostgreSQL. Since this DBMS is known to be not very CPU-efficient we also added this functionality to the high-performance X100 processing engine of the open-source DBMS MonetDB [BZN05], to be able to experiment with large numbers of concurrent queries.

5.1 PostgreSQL extension

In PostgreSQL, each query performing a table scan issues sequential page requests, possibly initializing an I/O event for each of them. Data is processed once a page is available. Since the default page size is 8 kilobytes, such behavior results in many disk accesses, which in a concurrent query environment may conflict with each other.

We have modified PostgreSQL to support *cooperative scans*. The first modification was to change the default behavior of the `heap` structure. Instead of reading pages one after another, the `heap` asks the `ABM` for the next chunk number and processes data within this chunk. Additionally, our extension introduces a new process that is responsible for fetching data chunks from the disk - we call it the *fetcher*. When the `ABM` chooses a chunk that is not cached, it sends a request to the *fetcher* to perform exclusive I/O. The requesting query polls the *fetcher* about data availability and starts processing once it is ready.

As discussed in Section 3.2, large I/O requests should be serialized to avoid disk interference. The *fetcher* process consumes an entire chunk before starting the next one. Note that it does not read an entire chunk (e.g. 8MB) in one I/O requests, instead, it issues multiple smaller (e.g. 64KB) requests marking already read parts of a chunk as available for processing. Thanks to that, the requesting query does not need to wait for the entire chunk loading time, resulting in better interleaving of I/O and processing.

Experimental setting For our experiments we used variants of TPC-H [Tra02] Query 1. This query consists of two processing steps. The first step is a scan of all the tuples and a selection on them using a simple predicate based on a 'DELTA' parameter (relatively cheap). The second step runs multiple aggregations the remaining on tuples (relatively expensive). Depending on the 'DELTA' value, this query can be either CPU-bound (when many tuples pass the first step) or disk-bound (when only few tuples are aggregated).

The test machine is a dual AthlonMP 1400+ with 1GB RAM running Linux kernel 2.6. The disk has an average bandwidth of ca. 20MB/sec. We used PostgreSQL server version 7.4.1 with the extensions described previously. It was filled with a TPC-H database of scale factor 3 - the `lineitem` table in this setting consumes ca. 3GB in PostgreSQL storage, which is enough to ensure it is not cached by the operating system. With the chunk size set to 2MB (256 pages) it resulted in dividing this table into ca. 1500 chunks. In all our experiments the ChunkCache size was set to 8 chunks (16 MB).

Experimental results The experiments presented in Table 4 consist of two runs. The **HomogeneousRun** runs of two identical disk-bound instances of Query 1 (selectivity of the first step is ca. 1.5%). In the **HeterogeneousRun** the second query was replaced with a much slower, CPU-bound one. Like in Section 2.2, three different scenarios were tested: *standalone*, *synchronized* and *de-synchronized*. We compare 3 different execution strategies: an unmodified PostgreSQL approach, and *attach* and *relevance* strategies presented in Section 3.

As the upper part of Table 4 shows, for the homogeneous queries all approaches behave equally well in the *synchronized* scenario. However, with queries *de-synchronized*, the traditional PostgreSQL method clearly looses to the new *attach* and *relevance* strategies. Note that since both queries have

	Standalone		Synchronized		De-synchronized	
	Q1	Q2	Q1	Q2	Q1	Q2
	HomogeneousRun					
PostgreSQL	127	133	132	132	478	477
Attach	126	128	127	127	131	131
Relevance	126	126	128	128	130	134
	HeterogeneousRun					
PostgreSQL	127	249	339	367	480	538
Attach	126	239	265	313	171	272
Relevance	126	244	129	258	130	258

Table 4: Performance of two homogeneous and two heterogeneous queries on modified PostgreSQL (seconds)

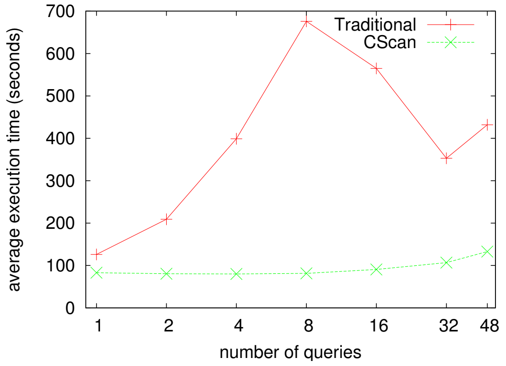

Figure 8: Heavy-load performance with MonetDB/X100

the same CPU requirements, the *relevance* approach does not introduce any benefit over the *attach* strategy.

For the heterogeneous run, the PostgreSQL performance degrades in both synchronized and de-synchronized scenarios. The *attach* approach improves the performance, mainly due to requesting large chunks, hence limiting disk interference. Still, when queries run de-synchronized, the performance drops significantly. This experiment shows the full potential of the *relevance* approach. In all test cases its performance stays more or less the same.

5.2 Heavy query-load with MonetDB/X100

Since PostgreSQL can only handle a few concurrent queries due to high CPU cost, we have decided to investigate the `CScan` performance with MonetDB/X100 [BZN05], a high-performance query processing engine geared both at high CPU efficiency and processing large (disk-resident) datasets. Its use of a vertically decomposed storage model limits per-query disk bandwidth requirements compared to other DBMSs. However, its highly CPU-efficient execution results in a very low cycle-per-byte ratio, increasing the hunger for disk bandwidth dramatically. We extended MonetDB/X100 with a slightly simplified *relevance* policy implementation. For our tests, we used a `lineitem` table of scale factor 30 (ca. 180M tuples). The columns used by Query 6 occupy ca. 3GB in this setting.

In the experiment, batches with varying numbers of queries (from 1 to 48) have been issued with a delay inversely proportional to the size of the batch. Figure 8 shows the results for both the traditional approach and the `CScan` implementation. The traditional strategy results in a sharp increase of execution time up to 8 queries. After that point, an interesting phenomenon can be noticed. Since the delay between the queries decreases, some of them may reuse already cached results of the previous ones, resulting in the forming of "convoys" (like in the *attach* approach), and a decrease of average query cost. In the `CScan` experiments the queries cooperate in disk I/O, resulting in a constant execution time for up to 32 queries! With more than 32 queries, the CPU becomes overloaded and the performance of both approaches starts to degrade slowly.

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. As we present in Section 3, this idea can be re-used for servicing query requests in the DBMS buffer manager. Other related operating system work is in the area of virtual memory and file system paging policies, for which generally LRU schemes are used. Some more recent work in this area focuses on policies that handle large multimedia streams efficiently [SV98].

A seminal paper in the area of buffer management for database systems is by Chou and DeWitt [CD85], that identifies the crucial advantage that DBMS buffer managers have over operating systems in that the queries that approach the buffer manager for pages convey *access patterns* upfront through their query plans that can be taken into account in the buffer scheduling algorithm. The proposed DBMIN algorithm applies a different buffering strategy for each of a predefined set of access patterns. This work was refined to take into account the actual availability of buffer pages in [FNS91, NFS91]. Another line of extension improved the estimated access patterns with feedback from actual queries [SS86, CR93].

In contrast with this work, we focus on only one of the identified access patterns, namely sequential scan (although in Section 4.2 we extend this to index-scan and repetitive scan as well). The previous work on buffer replacement strategies considers this an "easy" case, where the suggested policy is to *not* buffer any pages. Such a policy diminishes the chance of buffer page reuse by concurrent scans. The previous work does not consider concurrent queries apart from the fact that queries may compete for buffer manager pages [FNS91, NFS91].

The interaction and optimization of concurrent queries are studied in the research area of multi-query optimization [KdB94, MPK00, SSB00]. The general idea is to identify common work that is encountered in a query batch into a generalized query. The materialized results of this generalized query are then re-used multiple times to (partially) answer the queries from the batch. 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. However, it is still possible that two queries need the same disk pages using two different query processing algorithms (i.e. a sequential scan and an index-scan). Also, like the "elevator" approach, multi-query optimization introduces delays to gather a sufficient batch of queries, which may not always be acceptable.

A related approach is multi-query *execution* (rather than optimization). The NonStop SQL/MX server [ea99] introduced a special SQL construct, named 'TRANSPOSE', that allows to explicitly specify multiple selection conditions and multiple aggregate computations in a single SQL query, and which is executed internally as a single scan. This extension was aimed at helping data mining performance.

Outside the DBMS area, some of the multi-query optimization ideas have also been proposed. Müller and Henrich [MH04] discuss how a multi-media retrieval server can batch queries over VA-files [WB97] at multiple resolutions, applying the technique of query generalization to use the lowest resolution still sufficient to answer all queries with high precision. Similarly, Jónsson et al. [JFS98] presented a method for buffering an *inverted index* [Fal85]. It modifies the query execution order to first examine pages that are already buffered. Also, it introduces a new page replacement algorithm that assigns a special "importance" value to every page. Apart from not integrating these techniques in a DBMS buffer management context, this work differs in that it considers giving partial and non-precise query answers progressively, by basing the first answers on the cached page subset only.

Ideas close to our algorithm have been proposed in research related to using tertiary storage.

Sarawagi and Stonebraker [SS96] propose a solution that reorders query execution to maximize data sharing among the queries. Yu and DeWitt [YD97] present a solution that uses pre-execution 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. Our ideas, although strongly related, go higher in DBMS storage hierarchy, and additionally efficiently manage queries with varying CPU requirements.

Finally, Ramamurty and DeWitt recently proposed a query optimization framework that takes actual buffer content into account [RD05]. Note that choosing a sequential scan over an index scan if the (part of) relation happens to be cached, achieves a similar effect as our proposal to divert index scans into the active buffer manager. Still, buffer-aware query optimization will not help concurrent scan queries as we seek not a modification of the query plan but rather of the buffer management policy.

7. Conclusion

In this paper we considered application scenarios where multiple concurrent queries involving a table scan should be answered efficiently. Such environments are not supported well by current DBMS technology for two reasons: *(i)* pages needed by multiple concurrent queries are not shared, and *(ii)* I/O costs may strongly increase as multiple concurrent queries fight for disk bandwidth.

Our simple extension to the DBMS buffer manager infrastructure called *cooperative scans* addresses these problems effectively. This approach is extensible with different scheduling policies. We use simulation to describe a number of such policies and identify their main characteristics. Our recommended *"relevance"* policy can be efficiently implemented, as demonstrated in PostgreSQL, for which we present experimental results that show that DBMS performance under concurrent scans can be strongly improved. We also extend our approach from scan-only to index-scans, such that a wide class of join- and aggregation- and selection-queries can benefit from it.

A final advantage of adopting our approach is that the buffer manager can flexibly integrate I/O on the granularity of single pages (which benefits latency-bound OLTP queries) with large chunk-based I/O that extracts much higher bandwidth from modern disk hardware on queries that involve large data volumes.

References

[AGM89] R. Abbott and H. Garcia-Molina. Scheduling Real-Time Transactions with Disk Resident Data. In *Proc. VLDB*, Amsterdam, Netherlands, 1989.

[AMS$^+$97] S. Altschul, T. Madden, A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. Lipman. Gapped BLAST and PSI–BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, 25, 1997.

[BGRS99] K. Beyer, J. Goldstein, R. Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *ICDT*, ICDT99, 1999.

[BRK98] P. Boncz, T. Rühl, and F. Kwakkel. The Drill Down Benchmark. In *Proc. VLDB*, New York, USA, 1998.

[BZN05] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*, Asilomar, CA, USA, 2005.

[CD85] H.-T. Chou and D. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *Proc. VLDB*, Stockholm, Sweden, 1985.

[CJL89] M. Carey, R. Jauhari, and M. Livny. Priority in DBMS Resource Scheduling. In *Proc. VLDB*, Amsterdam, Netherlands, 1989.

[CR93] C.-M. Chen and N. Roussopoulos. Adaptive database buffer allocation using query feedback. In *Proc. VLDB*, Dublin, Ireland, 1993.

[ea99] J. Clear et al. NonStop SQL/MX primitives for knowledge discovery. In *Proc. KDD*, San Diego, CA, USA, 1999.

[Fal85] Christos Faloutsos. Access methods for text. *ACM Comput. Surv.*, 17(1), 1985.

[FNS91] C. Faloutsos, R. Ng, and T. Sellis. Predictive load control for flexible buffer allocation. In *Proc. VLDB*, Barcelona, Spain, 1991.

[JFS98] B. Jónsson, M. Franklin, and D. Srivastava. Interaction of query evaluation and buffer management for information retrieval. In *Proc. SIGMOD*, Seattle, USA, 1998.

[KdB94] M. Kersten and M. de Boer. Query Optimisation Strategies for Browsing Sessions. In *Proc. ICDE*, 1994.

[MH04] W. Müller and A. Henrich. Reducing I/O Cost of Similarity Queries by Processing Several at a Time. In *Proc. MDDE*, Washington, DC, USA, 2004.

[MPK00] S. Manegold, A. Pellenkoft, and M. Kersten. A Multi-Query Optimizer for Monet. In *Proc. BNCOD*, Exeter, United Kingdom, 2000.

[MyS] MySQL. `http://www.mysql.com`.

[NFS91] R. Ng, C. Faloutsos, and T. Sellis. Flexible buffer allocation based on marginal gains. In *Proc. SIGMOD*, Denver, USA, 1991.

[PGK88] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proc. SIGMOD*, Chicago, USA, 1988.

[Pos] PostgreSQL. `http://www.postgresql.org`.

[RD05] R. Ramamurthy and D. DeWitt. Buffer pool aware query optimization. In *Proc. CIDR*, Asilomar, CA, USA, 2005.

[SS86] G. Sacco and M. Schkolnick. Buffer management in relational database systems. *ACM Trans. Database Syst.*, 11(4), 1986.

[SS96] S. Sarawagi and M. Stonebraker. Reordering query execution in tertiary memory databases. In *Proc. VLDB*, Mumbai, 1996.

[SSB00] Pand S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proc. SIGMOD*, Dallas, USA, 2000.

[SV98] Prashant J. Shenoy and Harrick M. Vin. Cello: a disk scheduling framework for next generation operating systems. In *Proc. SIGMETRICS*, Madison, Wisconsin, United States, 1998.

[TP72] T. Teorey and T. Pinkerton. A comparative analysis of disk scheduling policies. *Commun. ACM*, 15(3), 1972.

[Tra02] Transaction Processing Performance Council. *TPC Benchmark H version 2.1.0*, 2002. `http://www.tpc.org/tpch/spec/tpch2.1.0.pdf`.

[WB97] R. Weber and S. Blott. An approximation based data structure for similarity search. Technical Report 24, ESPRIT project HERMES (no.9141), October 1997.

[YD97] J.-B. Yu and D. DeWitt. Query pre-execution and batching in paradise: A two-pronged approach to the efficient processing of queries on tape-resident raster images. In *Proc. SSDBM*, Olympia, WA, USA, 1997.