Increasing Buffer-Locality for Multiple Index Based Scans through Intelligent Placement and Index Scan Speed Control

Christian A. Lang
Bishwaranjan Bhattacharjee
Tim Malkemus
IBM T.J. Watson Research Center
19 Skyline Dr
Hawthorne, NY 10532
+1 (914) 784-7387
{langc,bhatta,malkemus}@us.ibm.com

Kwai Wong
IBM Toronto Lab
8200 Warden Ave
Markham, ON L6G 1C7
+1 (905) 413-2818
kwaiwong@ca.ibm.com

ABSTRACT

Decision support systems are characterized by large concurrent scan operations. A significant percentage of these scans are executed as index based scans of the data. This is especially true when the data is physically clustered on the index columns using the various clustering schemes employed by database engines. Common database management systems have only limited ability to reuse buffer content across multiple running queries due to their treatment of queries in isolation. Previous attempts to coordinate scans for better buffer reuse were limited to table scans only. Attempts for index based scan sharing were non existent or were less than satisfactory due to drifting between scans.

In this paper, we describe a mechanism to keep scans using the same index closer together on scan position during scanning. This is achieved via intelligent placement of index scans at scan start time based on their scan ranges and speeds. This is then augmented by adaptive throttling of scan speeds based on the index scans' runtime behavior during scan execution. We discuss the challenges in doing it for index scans in comparison to the more common table scan sharing. We show that this can be done with minimal changes to an existing database management system as demonstrated in our DB2 UDB prototype. Our experiments show significant gains in end-to-end response times and disk I/O for TPC-H workloads.

1. INTRODUCTION

Decision support systems (DSS) are characterized by the presence of large queries which perform scans over a substantial part of the data (e.g., to compute aggregate values). A significant percentage of these scans are executed as index based scans of data. This is especially true when the data is physically clustered on the index columns. The importance of clustering in reducing physical I/O during query processing is well recognized and is reflected in the fact that most database engines support one or more types of indexed physical clustering schemes nowadays. Examples include Multi Dimensional Clustering (MDC) [1][2] in DB2 UDB [9], Partitioned Primary Index in Teradata [3] and Index Clustered Tables in Oracle [4]. Recent trends indicate that typical DSS users are moving towards more and more concurrent queries [5]. Common database management systems have only limited ability to reuse memory buffer content across multiple running queries due to their treatment of queries in isolation.

In addition to this shift in workload characteristics, technological changes in the storage subsystem demand better memory buffer reuse as well. Disk drives are increasing in capacity but seek and access time is not keeping up [6]. Thus systems are going to be more and more prone to becoming I/O bound [7]. Therefore, a mechanism which can reduce random seeks on disk and reduce the stress on the storage subsystems will go a long way in improving overall system throughput.

DBMS engines which try to optimize scan-heavy query workloads have generally limited themselves to table scans only. Attempts for index scan based scan sharing were non existent or were less than satisfactory due to drifting between scans. This is despite the fact that in real customer situations, one encounters a lot of overlapping index based scans. These are used to access the hotspots of a Data Warehouse. For example, a Data Warehouse might have 7 years of data and multiple analysts might be interested in the last year or month of data. Their queries would likely use an index based scan of some sort over that part of the data. An analysis of a DB2 customer scenario indicated that their database had 150 users who were submitting 215 different types of queries. These were using 553 index scans with two tables having more than 100 index scans and 15 tables having more than 10 index scans each. This scenario has a high index scan sharing potential and is representative of many other scenarios.

In this paper, we describe a mechanism to increase buffer reuse and thereby reduce I/O for concurrent index based scans. This mechanism has been prototyped in DB2 UDB and provides significant performance improvement for concurrent queries. The mechanism increases buffer reuse by keeping scans using the same index closer together on scan position during scanning. This is achieved via intelligent placement of index scans at scan start time based on their scan ranges and speeds. This is then augmented by adaptive throttling of scan speeds based on the index scans' runtime behavior during scan execution.

The index scan sharing technology was built on the infrastructure developed for facilitating the grouping and throttling of table scans in DB2 UDB described in [8]. In this paper we discuss the challenges in doing index scan sharing in comparison to the more common table scan sharing. The effort required to extend existing database management systems with our new algorithm is minimal, as shown in our DB2 UDB prototype. The prototype currently supports MDC Block Index Scans but can be modified for other index scans very easily. Our experiments gave end-to-end gains of 21% for 5-stream TPC-H benchmark runs.

The rest of the paper is organized as follows. In Section 2, we present related work. In Section 3, we discuss how index scans work in a typical RDBMS. Section 4 presents our new "sharing index scan operator", SISCAN. Section 5 describes the module that controls sharing between SISCANs. In Section 6 and 7, we show how SISCANs are started and adaptively throttled. Section 8 contains our experimental results and we conclude in Section 9.

2. RELATED WORK

This work touches various areas: caching, database query optimization, and database query execution. In the following, we therefore summarize related work in these areas.

Different techniques have been proposed for increasing buffer locality for various workloads. One of the oldest and most basic algorithms is LRU which evicts the page from the buffer that was not accessed the longest. LRU is currently the policy of choice in many database systems due to its small overhead and tuning-free operation. Many variants of LRU have been proposed since. Examples are LRU-K [10], 2Q [11], LFU [12], and hybrids such as LRFU [13] and ARC [14]. All these techniques are for general access patterns, while this paper focuses on ordered access patterns only and can therefore achieve much improved buffer utilization for this specific type of access.

Commercial database vendors such as Red Brick [15], Teradata [16,17], and Microsoft SQL Server [18] employ proprietary algorithms to let the database synchronize multiple table scan operations in order to maximize buffer locality. This idea was taken even further by Harizopoulos et al. [19]. They propose ideas for a new database architecture that tries to maximize reuse of partial query results from the query down to the page access level. This is achieved by detecting overlaps in active query plan operators at query execution time and then exploiting it by pipelining one operator's results to all dependent operators where possible. Two of the operators discussed in that paper are the table and index scan operators. For these, the authors propose to use one scan thread that keeps scanning all pages while table scan operators can attach to and detach from this thread in order to share the scanned pages.

While this approach works well for scans with similar speeds, in practice scan speeds can vary by large margins and even single scans' speeds are usually far from constant due to changes in predicate evaluation overhead. Therefore, the benefit can be lower as scans may start drifting apart. Techniques to prevent drift by automatically throttling faster scans and by scan-group based

```
1 proc IXSCAN( startKey, endKey )
2    loop l from loc(startKey) to loc(endKey)
3       perform operations on page(l);
4       release page(l) with priority p;
5    endloop;
6 endproc;
```

Figure 1. High-level logic of IXSCAN operation

prioritization of buffer pages have been discussed in [5]. However this work is applicable for table scans only.

In addition to cache or page buffer algorithm improvements, other methods to reduce disk access costs for multiple concurrent queries with overlapping data accesses have been investigated. These methods include multi-query optimization [20] (which requires all queries to be known in advance) and query result caching [21]. Due to being at a high level of the query execution hierarchy, the latter may miss out on sharing potential for queries that have very different predicates but still end up performing scans on the same table, for example.

Zukowski et al. [22], Sacco et al. [23], and Chou and DeWitt [24] introduce smarter buffer managers that are used to optimize page replacement under multiple running queries in order to maximize buffer locality. Their approaches require significant modifications of the caching system. This paper, on the other hand, views the caching system as a "black box" and limits modifications to a few extra function calls in the index scan code.

3. INDEX SCAN OPERATORS

This section gives an overview of IXSCAN (standard index scan on B+ trees) processing, including the concept of "location" of a scan, and how concurrent index scans may interact with each other.

3.1 Index Scan Overview

An access plan consists of a number of operators, used to satisfy a given query, as determined by the optimizer. The operators represent processing steps such as table scans, joins, predicate evaluations, etc. One such operator is IXSCAN, or index scan. An IXSCAN includes various attributes, such as which index to scan, whether there are predicates to apply, etc.

A standard index scan consists of reading leaf pages from an index, processing each entry (i.e., a key and a row identifier, or RID) sequentially, and (possibly) following each RID to retrieve the corresponding record from the underlying table. There may be a start key and/or an end key to limit the scope of the scan, depending on predicates in the SQL query.

Figure 1 shows a high-level pseudo-code algorithm of an IXSCAN. Assuming a start key is given, it is used in a tree search, starting with the root of the index, to find the child page of each non-leaf page that may contain the key value specified, until such a child is a leaf page. If there is no start key, the index scan begins at the first leaf page in the index. This is done in the `loc()` function in line 2 of the algorithm. Once the starting page has been determined, the scan begins, with a loop over the leaf pages, and a loop over each entry (line 2). The page corresponding to the RID is read into the bufferpool[1] (or located in the bufferpool, if it

[1] The *bufferpool* is a memory region used by DB2 to cache pages during query execution.

is already there), and the record specified by the RID is located. That record is then handled (line 3) in whatever way is needed for the query processing (predicates, aggregations, sorting, etc.).

After reading and processing the relevant parts of a table page, the page gets released in the bufferpool (line 4) so that the space can be used for other pages if needed. The priority assigned to the released page is an indicator to the caching algorithm for deciding which pages to discard first when space is needed. In the IXSCAN algorithm this priority is typically fixed during a scan (though it may change from one scan to another). We show later how we adjust this parameter to increase bufferpool sharing.

Once the page is released, the next iteration of the loop will find the next RID in the index entry, if any. If there is an end key, each entry encountered in the loop is compared against that value for determining the end of the scan. If there is no end key, the scan ends with the last entry in the last leaf page.

3.2 Index Scan Location

At any time during an index scan, it is said to have a particular "location". This is the indication of which key (index entry) and RID is currently being processed. Whenever the index scan moves "forward", the next RID of the same key value is accessed until no more are available. Then, the scan switches to the next key value (in increasing or decreasing order) and the first RID for that key.

This is in contrast to a table scan whose location is described simply as the current RID and the next location is obtained by increasing or decreasing the RID. This distinction is important because only the key values are in increasing or decreasing order during an index scan. The RIDs may not be in any specific order. This can lead to many expensive disk seek operations if the RIDs are poorly distributed across the scan range. Improved caching can mitigate this problem.

3.3 Concurrent Index Scans

Often, there will be multiple applications performing the same index scan at the same time. There may or may not be overlap in the ranges of values covered by the concurrent scans. The scans may have the same values for start and end keys, or the start and end key ranges may overlap in some way, or they may be disjoint.

If there is overlap in the ranges specified for start and end keys, then the scans may benefit each other in terms of bufferpool usage. The first one to read a particular page has paid the price of the physical I/O, and the second one can process the page that is already in the buffer pool, avoiding a second physical read. However, perhaps the second scan starts after some time has passed. In that case, there is a chance that the buffer manager has already had to victimize that page, and the second scan will have to do a physical read again. When this happens, both scanners will have to do physical reads of the same pages, resulting in roughly twice the number of physical reads as for only one scan. In other words, if the scan locations are close, the number of physical page reads needed is oftentimes reduced.

3.4 Block Index Scans

A block index is essentially the same as a RID index, except that each index entry consists of a key and a list of Block IDs (BIDs) instead of Row IDs (RIDs). A *block* is a contiguous set of pages in the table that contain records of the same key value. The block size is constant for a given table, and is determined when the table is created. Block index scans are used very frequently on MDC tables [1][2] in decision support systems that require slicing and dicing of large datasets. We implement our prototype initially for block index scans for this reason. All other properties discussed for RID index scans also apply to block index scans with the scan location now being described as key/BID instead of key/RID.

While "chance" buffer sharing between regular index scans may occur occasionally, it is rare. We therefore discuss next a new type of index scan that actively "seeks" buffer sharing.

4. NEW SISCAN OPERATOR

In this section, we introduce a new index scan operator SISCAN (for "sharing index scan"). This operator can be inserted at most places of the query plan where a IXSCAN operator would be applicable and enables the corresponding index scan to actively share bufferpool content with some other ongoing index scans. We will discuss cases when sharing between index scans should not be permitted, in Section 4.1.

In a query plan, the IXSCAN operator may be associated with a start and/or end key specifying the range to be scanned. As discussed in Section 3, the corresponding index structure would then be traversed at runtime starting from the start key and ending with the last identifier matching the end key.

The SISCAN operator also traverses the index structure, accessing every record between start and end key but the traversal logic is as follows (cf. Figure 2):

1. pick a start location *startLoc* (i.e., key and RID)
2. scan index from *startLoc* to end key
3. scan index from start key to *startLoc*

By allowing the SISCAN to start at any location in step 1, bufferpool page sharing can be improved by starting a new SISCAN at the location of an ongoing SISCAN. If both have a similar speed, they will subsequently read the same index entries and thereby the same base table pages, even if the RIDs are "randomly" distributed across the scan range.

Due to its nature, the SISCAN operator has additional parameters for controlling various aspects of bufferpool sharing. These parameters are the *scan speed estimate* and the *scan amount estimate*. The speed estimate characterizes the index scan's speed in number of pages read per second and scan amount estimate characterizes the overall number of pages to be read between start key and end key. These parameters are supplied by the costing component of the query compiler and can be based on table statistics and/or past measurements.

From an implementation perspective, adding piecewise scans is not difficult as most RDBMSs already provide similar facilities for farming out scans to multiple processors. Also, breaking an index scan into only two phases rather than some other more

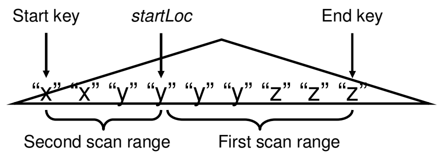

Figure 2. Two index scan ranges

complex scan pattern reduces both implementation complexity and the required memory footprint for scan status information.

4.1 SISCAN Operators in Query Plans

The query optimizer chooses an index scan operation over some other access type if the cost caused by the index scan is lower than the cost caused by the other access types. This may be due to selectivity or due to sorted results needed later in the query plan, making an index scan desirable.

From the optimizer's point of view, a SISCAN operator can occur anywhere an IXSCAN operator is used. However, an important difference between IXSCAN and SISCAN operators is that IXSCAN returns identifiers in monotonic key order (either increasing or decreasing) while SISCAN may not due to the modified key traversal order shown in Figure 2. Therefore, if the query optimizer decides to use an index scan for getting records ordered on the index key value, it can only use IXSCANs. Fortunately, in practice, there are many cases where index scans are used for reduced access time rather than for key ordering.

Similarly, we are not discussing replacement of "index only" IXSCANs (i.e., scans that only generate a RID list without processing the underlying table data) with SISCANs, even though scans on large indexes may also benefit from sharing of index pages in bufferpool or CPU caches.

4.2 SISCAN Process Logic

As visible in Figure 3, the logic of a SISCAN is very similar to the logic of a regular IXSCAN. The main difference is that (1) a SISCAN can start in the middle of its key range, and (2) the SISCAN periodically calls a new component, the *index scan sharing manager* (ISM). The ISM is the central ingredient to our bufferpool sharing mechanism and will be discussed in depth in Section 5. For now, it suffices to know that the ISM keeps track of ongoing SISCANs and uses this knowledge to influence index scan speeds and bufferpool replacement decisions.

At a high level, the interaction of a SISCAN with the ISM is given as pseudocode in Figure 3. The lines in bold font mark the differences with the IXSCAN logic in Figure 1. SISCANs first register with the ISM (line 2), providing its start and end key and the expected time needed to finish the scan. The ISM then determines whether the new SISCAN should start at the first key/RID of the scan range or whether it can join some other ongoing SISCAN. We will discuss new index scan placement in detail in Section 6. The start location is returned to the caller.

The SISCAN then starts scanning the index like an IXSCAN but instead of starting from the start key, it starts from the location *startLoc* assigned to it by the ISM (line 3). The operations for each index entry are similar to IXSCANs except that SISCANs may periodically call the ISM to update their scan location (lines 5 and 10). These calls can be performed at every page or they can be performed at every *x* entries read. In general, the more frequent the calls are made, the more accurate the location information in the ISM but the higher the additional overhead.

In line 6, the processed page is released in the bufferpool. Contrary to IXSCANs, in the case of SISCANs, the priority is adjusted dynamically by the ISM based on the state of all SISCANs in the system and returned as result of the *ISM.pr()* call. After picking the starting location in line 1, this is the second degree of freedom that the ISM has. The idea is that this priority is chosen such that pages that are needed soon again will receive higher priority than pages not needed in the near future. Details on how the ISM chooses the priority can be found in Section 7.

```
 1 proc SISCAN( startKey, endKey )
 2    startLoc := ISM.startSISCAN( startKey,
                                   endKey, time );
 3    loop l from startLoc to loc(endKey)
 4       perform operations on page(l);
 5       ISM.updateSISCANLocation(l);
 6       release page(l) with priority ISM.pr();
 7    endloop;
 8    loop l from loc(startKey) to startLoc
 9       perform operations on page(l);
10       ISM.updateSISCANLocation(l);
11       release page(l) with priority ISM.pr();
12    endloop;
13    ISM.endSISCAN();
14 endproc;
```

Figure 3. High-level logic of SISCAN operation

In lines 8 through 12, a second index scan is performed, this time starting at the original start key and ending at *startLoc*. Finally, in line 13, the ISM is informed that the index scan has ended. When reaching this point, all index entries starting at the start key and ending at the end key have been read and processed. Conceptually, one can think of the SISCAN logic as two back-to-back IXSCANs over adjacent key ranges (cf. Figure 2).

5. INDEX SCAN SHARING MANAGER

In this section, we discuss details of the index scan sharing manager (ISM). The ISM keeps track of ongoing index scans, their locations, and their speeds with very little overhead in terms of memory and CPU usage. The ISM also determines the start location of a new SISCAN and performs scan speed regulation and bufferpool entry reprioritization based on the characteristics of ongoing SISCANs with the goal of maximizing buffer reuse. In the following, we discuss the overall architecture, attributes maintained for SISCANs and groups of SISCANs.

5.1 ISM in Architectural Context

Figure 4 shows the ISM in context of the overall architecture. During the execution of a query with index scans, SISCAN operators are spawned (step 1). These inform the ISM about their start (step 2) and determine which table pages to scan via index access (step 4). The required pages are then fetched from the bufferpool (step 5). Periodically, they inform the ISM about their current index location (step 3). When the index scan is finished, they also inform the ISM (step 2).

This architecture has several advantages. First, the ISM's interface can be kept relatively simple (we only need "start/end of SISCAN" and "update location" calls). Second, the ISM is kept separate from the rest of the architecture. There is no direct interaction between ISM and bufferpool replacement algorithms or index access functions. Only the SISCAN processes interact with the ISM. This has the advantage that the required changes to the architecture are very localized.

In the next sections, we discuss what information is kept in the ISM and how it is used to establish a partial ordering between SISCANs. This partial ordering will come in handy when we discuss where to start new SISCANs and how to control their speeds.

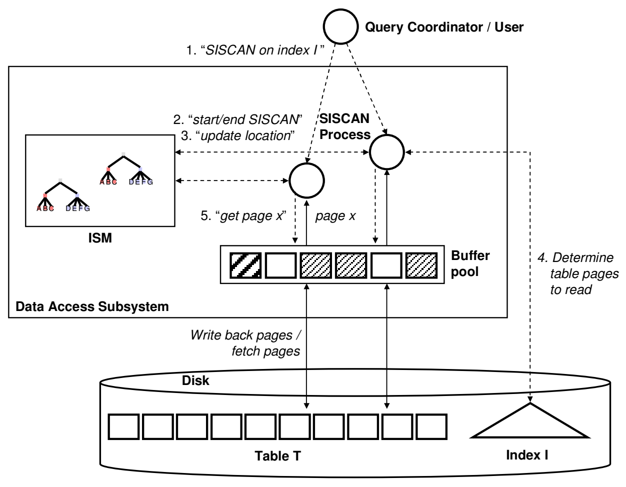

Figure 4. ISM in context of the overall architecture

5.2 Attributes Maintained by ISM

There is one ISM per bufferpool since index scans usually do not span multiple bufferpools. Each ISM maintains statistics about the ongoing SISCANs. For each SISCAN, the ISM maintains its *location* (key value and RID), *remaining pages* in scan range (this value is initialized from the scan amount estimate in the SISCAN operator, cf. Section 4), *average speed* (in pages/s; this is initialized as "(estimated pages in scan range) / (estimated scan time)" and updated during scan execution), *start and end key* of the scan range, an *anchor location* and an *anchor offset* (these attributes are needed to determine the relative ordering between SISCANs; we will return to this in the next Section). The ISM updates these attributes whenever SISCANs start, finish, or update their location.

5.3 Partial SISCAN location order

Compared to table scans, it is difficult to determine distances between index scans by only inspecting their scan locations because the RIDs are not necessarily accessed in any monotonic order and so the distance is not simply the difference between two SISCANs' scan locations. Consider the example in Figure 5. The location of index scan A is (key "y", RID 5) and the location of index scan B is (key "z", RID 8). The distance between the two scans in index order is not necessarily 3 pages as the RIDs seem to indicate. It could be more or less, depending on how the data pages are laid out on disk. In this example, the distance is 5, since the RIDs in index order between A and B are 5, 3, 4, 7, 9, and 8.

However, in order to determine the potential for bufferpool sharing and necessary adjustments to the index scan speed to

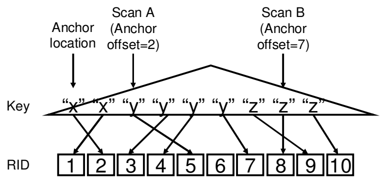

Figure 5. Anchor location and offsets

increase sharing, it is necessary to estimate the distance between two index scans. For this reason, the ISM maintains information about the distance of a SISCAN to a fixed known location – the *anchor location*. Whenever a new SISCAN A starts, it may either start by itself or it may start at some other SISCAN B's location. In the first case, the ISM sets A's anchor to its start location and the anchor distance to 0. In the second case, the ISM sets A's anchor to the location of B's anchor and A's anchor offset to B's offset. Whenever A or B move, their anchor offset is updated with the moving distance but their anchor remains the same. By knowing the distance of A and of B from the anchor, the ISM can calculate the distance between A and B.

For the example above, let us assume the common anchor of scans A and B is (key "x", RID 2) and scan A's anchor offset is 2 and scan B's anchor offset is 7 (cf. Figure 5). Now the distance between A and B can easily be calculated as the difference between the anchor offsets, namely 5. It should be noted that the maintenance of anchor locations and offsets can be done while regarding the index as a black box. No changes are necessary in the way the index is managed and in fact, the same anchor/offset technique works for any type of index that has a deterministic ordering.

It is useful to think of the anchors and offset as defining a *partial ordering* between SISCANs. In the example in Figure 6, two anchors were created with four SISCANs (shown as black dots) having one of the anchors in common and two SISCANs having the other anchor in common. We will refer to SISCANs that have an anchor in common as *anchor groups* from now on. Since we know the relative locations between scans within anchor groups but not between anchor groups, we obtain a partial ordering between SISCANs as A<B, B<C, C<D, E<F where "<" denotes the (transitive) partial order defined by anchors and offsets.

Next, we discuss an algorithm for choosing a starting location for a new SISCAN. We will see how the partial SISCAN ordering becomes useful in the practical implementation of this algorithm in Section 6.3.

6. SISCAN PLACEMENT

Whenever a new SISCAN starts, it calls the *startSISCAN* function in the ISM (line 2 in Figure 3). The ISM then determines where the new SISCAN should start by inspecting its scan statistics data structure. If there are no other SISCANs, the ISM may start the new scan at its start key/RID. If there are some other ongoing SISCANs, the ISM has to decide the placement of the new scan based on the other scans' speeds, locations, and scan ranges. The overall objective is to maximize bufferpool sharing between all SISCANs by selecting the starting location of each scan based on the available statistics.

If, for example, there is one ongoing scan A with scan range [key "a", key "f"] and its current location is at key "b" and the scan range of a new scan B is [key "d", key "g"], B cannot share bufferpool pages by starting at A's location because key "b" is outside of B's scan range.

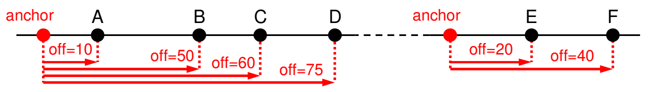

Figure 6. Two anchor groups

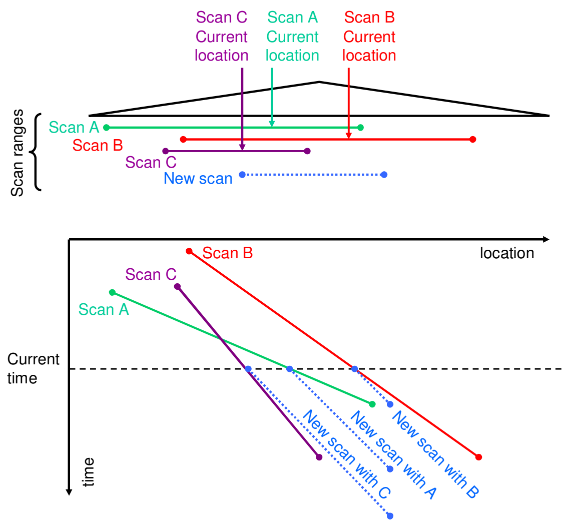

Figure 7. Placement of new SISCAN

Even when ruling out cases where the location of ongoing scans is outside of the scan range of the new scan, there may be many scans that can be selected as starting location. Consider the example in Figure 7. There are three ongoing SISCANs, A, B, C. The figure on the top shows their scan ranges as horizontal lines. The current location of each scan is marked with an arrow. The new scan's key range is shown as a dotted line below.

By inspecting only the amount of overlap between the scan ranges, it is tempting to assume that scan B would be a good choice to share bufferpool contents with. However, when taking into account the time dimension, as shown in the graph below, it becomes obvious that this is not a good choice. The lower plot shows the scan location (in index order) on the x-axis and the time on the y-axis, with the current time marked with a dashed line. Different speeds of the scans show up as different slopes of the lines representing the scans. The current location of the scans is the intersection point of the scan lines with the line of the current time. For simplicity, we assume the speed of the scans is constant and that past speed is a good indicator for future speed. Furthermore, we assume that we can estimate the speed of the new scan, indicated by the slope of the dotted lines.

The figure shows three potential starting points for the new scan: at the location of scan A, scan B, or scan C. When starting at scan A's current location, A and the new scan drift apart quickly, due to their different speeds. Sharing of bufferpool pages is therefore only of short duration. Once the distance between A and the new scan reaches a threshold, the other ongoing scans will cause the pages read by A to be discarded from the bufferpool, causing additional I/Os for the new scan that has to re-read them, thereby increasing the drift even further.

If the new scan is started at scan B's location, the drift is smaller because scan B's speed is more similar to the new scan's speed. However, the sharing duration is limited by the fact that the new scan's key range has very little overlap with scan B's remaining scan range. Once the new scan reaches the end of its range, sharing of bufferpool pages between the two scans ends.

Finally, if the new scan is started at scan C's location, sharing of bufferpool pages is maximized because scan C has a similar speed and scan C's remaining scan range overlaps sufficiently with the new scan's key range. Therefore, even though not obvious when inspecting the key ranges by themselves, starting the new scan at scan C's location is the best choice in this example.

We learn from this example that placement of a new SISCAN depends on (1) the speeds, and (2) the remaining scan ranges of the ongoing SISCANs. We also note that in order to find the optimal placement, it is not enough to consider ongoing SISCANs individually as we have done in the example above. In many cases, a new scan may first share bufferpool pages with one ongoing scan and then switch to share with another scan due to speed changes and drift. An algorithm to find the optimal placement should therefore also consider starting locations that do not lead to immediate sharing, if they lead to more sharing later on.

6.1 Estimating sharing potential

Each starting location of a new SISCAN has a different sharing potential based on the other ongoing scans. How can this potential be estimated? Let is consider the example in Figure 8. Again, the x-axis represents the location of the index scans and the y-axis represents time increasing towards the bottom. Current time is the top-most dashed line. There are three ongoing SISCANS: A, B, and C with different scan ranges and speeds. We also know there will be a scan D in the future. This may be the second phase of scan A (assuming A was started in the middle and has to perform a second phase as explained in Section 4.2), or a scan that is the inner of a nested loop join and therefore repeated multiple times.

Given this state of the system, assume a new SISCAN E is about to start. Its scan range and expected scan time is shown as the dotted box in the center (the width of the box corresponds to its scan range and the height corresponds to the time to scan the range).

Let us consider starting E at the beginning of its scan range (Figure 8). This will result in a trace as shown by the dotted line marked with "E". Between the current time T0 and time T1, there are four ongoing SISCANs (A, B, C, and E). Only C and E are close enough to share some bufferpool pages (as shown by the hashed area between C and E). However, sharing is only possible if C and E are close to each other since otherwise A and B may cause bufferpool pages to be displaced. Between time T1 and T2, only three scans (C, D, E) are running, thereby allowing for sharing even when C and E are further apart (cf. larger hashed area between T1 and T2). However, at some point, the distance between C and E will be too large due drift caused by their different speeds, and sharing will end.

The sharing potential for this scenario can now be calculated by counting how often pages in E's scan range had to be read and re-read. In the key range before C and E start sharing, there are three scans that read those pages at different times (C, D, and E), so each page in this range is read three times. The second key range is when only C and E are active and sharing, so each page here is read only once (marked in red because the number would be higher without sharing). In the next range, A is active in addition, so each page is read twice (also marked in red). Then C and E stop sharing, so each page is now read three times. And in the last key range, B, C, and E are active but none is sharing, so each page

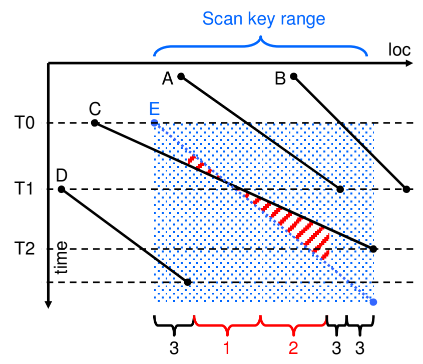

Figure 8. Sharing potential when starting new SISCAN E at beginning

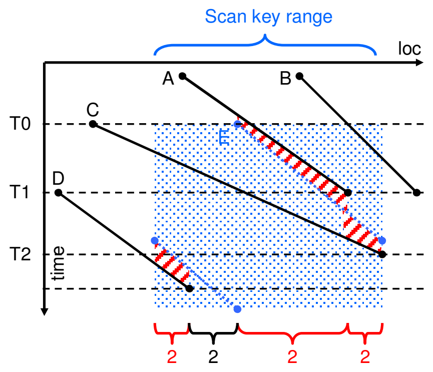

Figure 9. Sharing potential when starting new SISCAN E near scan A

is read three times. If for example, the sizes of the ranges are 15, 30, 15, 20, and 10 pages respectively, then the overall amount of page reads "within the dotted area" would be 15*3 + 30*1 + 15*2 + 20*3 + 10*3 = 195. This compares to the worst case without sharing which is 15*3 + 30*2 + 30*3 + 5*3 + 10*3 = 240. So, the potential reduction in I/Os through sharing is 1-195/240 or 19%.

Let us now consider a different starting point for E (Figure 9). This time, E is started near ongoing scan A. It will then scan until it reaches the right end of the scan range, "wrap around", and start from the left end of the range (second phase). Therefore, we see two dotted lines corresponding to scan E. In this case, E can share with A right from the beginning all the way until A ends, since both have similar speeds and stay therefore close together. After sharing with A, E can share pages with scan C until the end of the scan range is reached. Finally, after starting in phase two, E can share with scan D until D ends.

We calculate the sharing potential the same way as before. In the first key range, in which D and E are sharing and C is not, we read each page twice. In the second range (C and D ongoing, not sharing), we read each page twice. In the third range (A and E sharing, C not), again each page is read twice. And in the last range (C and E sharing and B not), we again read each page twice. For range sizes 15, 20, 40, and 15, respectively, we have 15*2 + 20*2 + 40*2 + 15*2 = 180 page reads. The potential I/O reduction is therefore 1-180/240 or 25%. Starting E near scan A is therefore preferable.

```
1  fct calculateReads( scan t, scanset S )
2     R := empty set;
3     for each time interval Ti defined by
      scan start/stops
4        n := number of ongoing scans in Ti;
5        determine key ranges K during which t can
         share with scans in S based on n;
6        add K to R;
7     endfor
8     reads := 0;
9     for each key range r in R
10       reads := reads + reads(r)*pages(r);
11    endfor
12    return reads;
13 endfct;
```

Figure 10. Algorithm for calculating the number of page reads necessary for a given SISCAN start location

More formally, the algorithm for estimating the number of page reads for a given SISCAN start location and a given set of ongoing SISCANs is given in Figure 10. The function reads(r) in line 10 determines the number of reads and re-reads for any page in the range r and the function pages(r) determines the number of pages in range r. Since the number of time intervals is proportional to the number of ongoing scans and the number of key ranges in R is proportional to the number of scans as well, the running time of this function is O(|S|).

Now that we know how to calculate the expected page read cost for a given SISCAN start location, we will discuss next how to pick the start location that leads to a small read cost.

6.2 Finding the best starting location

One naïve way to find the best starting location would be to try every possible start location and pick the one with maximum sharing potential. Since this requires O(p) steps where p is the number of pages in the scan range, this is not very practical. Fortunately, it suffices to inspect only the locations where we can expect local optima. In the following, we call such locations "interesting locations".

Consider Figure 11. This is an example with one ongoing scan A and a new SISCAN that starts at time T0. Knowing how many other scans are active between T0 and T1, we can draw an "envelope" around A (shown as hashed area) that indicates the area in which there can be sharing between A and another scan. If another scan is outside the envelope, there is no sharing because they are too far apart. There are three interesting starting locations for the new scan: E1, E2, and E3. E1 is the location when the new scan touches A's envelope on the left. E2 is the location when the new scan goes through the center of A's envelope. And E3 is the location when the new scan touches A's envelope on the right. The reason why these locations are the only ones that may have local optima is because between interesting locations E1 and E2 the sharing potential is monotonically increasing and between E2 and E3 it is monotonically decreasing.

In this example, the solution would obviously be the starting location E2 as it has maximum sharing. However, for many ongoing scans, there may be many local optima. Even then it still holds that the monotonicity of the sharing potential function may change from increasing to decreasing (or vice versa) from one interesting location to the next while being monotonic (actually even linear) between interesting locations. In other words, local optima can only occur at interesting locations.

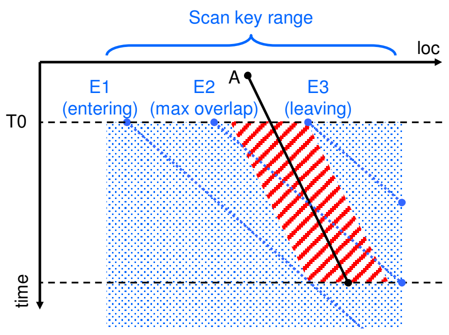

Figure 11. Determining "interesting" SISCAN starting locations

This yields a simple algorithm for finding the best starting location: (1) determine all interesting points, (2) for each such point, compute the sharing potential with the `calculateReads` function (Figure 10), (3) pick the point with minimum reads (maximum sharing). Step (1) requires to find the three interesting locations for each scan "envelope" in each time slice. There are O(|S|) time slices and O($|S|^2$) envelopes for |S| scans. Since each sharing potential calculation requires O(|S|) cost, we have an overall cost of O($|S|^3$).

6.3 Practical implementation

While the previously discussed algorithm will find the starting location with optimal sharing potential, it has two shortcomings for practical purposes: (1) the cost is relatively high (even though this cost only occurs at SISCAN start time) and (2) linearly ordered locations for all SISCANS as assumed in the time/location graph in Figure 11 may not be available without access to the index structure (as we had discussed earlier, the RID component of the index scan location may not be ordered in any way). For this reason, we now discuss a modified version of the previous algorithm that has lower cost and treats the index structure as a black box. The high-level idea is to compute the sharing potential for anchor groups that overlap the new scan's key range based on the partial ordering between SISCANs discussed in Section 5.3.

For purpose of presentation, let us assume one anchor group of three SISCANs A, B, and C as shown in Figure 12. The current time is again indicated as T0. We also assume the current location of A, B, and C (the intersection of their traces with the dashed T0 line) are inside the new scan's key range (shown as shaded region), as is the end key of A and B and the anchor. C's end key is assumed to be outside of the new scan's range. Contrary to the previous algorithm, we now have only partial knowledge about the relative locations of scans and their end keys. We know the distance of A's current location from the anchor (30 in this example), the distance of B's location from the anchor (60 in this example), and the distance of C's location from the anchor (75 in this example). In addition, we know the past (and assumed future) speed and the expected number of remaining pages of A, B, and C (denoted by "remaining(A/B/C)"). Scan speed and remaining scan pages are maintained for each SISCAN (cf. Section 5.2) and give us the approximate end location of a scan relative to its start location (as shown in Figure 12).

This still leaves the question of where the start of the new scan range is located (we only know it is before the anchor) and where

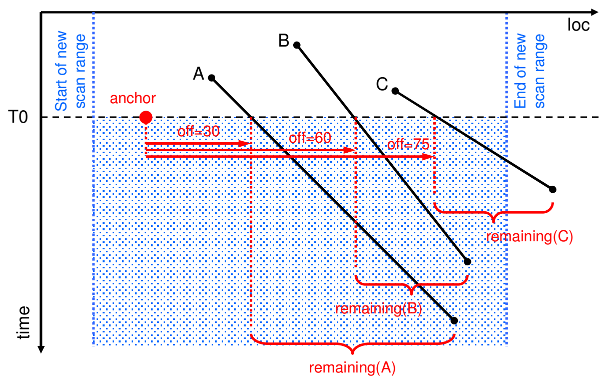

Figure 12. Finding best SISCAN starting location with anchors and offsets

the end of the new scan range is located (we only know it is between the end of B's scan range and the end of C's scan range). Since the location of the new scan start key is needed to estimate where a "wrapped" SISCAN continues, we will estimate the sharing potential only for the "pre-wrap" phase. In addition, to be conservative, we set the end location of the new scan to the smallest location (in this case, the end location of B).

We now have nearly all the parameters needed to apply the `calculateReads` function to this scenario: we know the current locations and the end locations of A, B, C (and their trace slopes), we know the new scan's expected speed (from the *average speed* attribute), and we have fixed the new scan's end key location. The only parameter missing is the new scan's start location. In the previous section, we picked it from among the "interesting locations". In this scenario, we have only three locations to pick from: the current location of A, B, and C. We propose to calculate the sharing potential for each of these. The best one gives us the best starting location for this group of scans. We then repeat the same procedure for other anchor groups that overlap the new scan's key range and pick the location with best overall sharing potential among all groups as the new scan's starting location. The pseudocode algorithm is shown in Figure 13. Since we need to evaluate `calculateReads` once per scan, the overall cost for this algorithm is O($|S|^2$).

One interesting special case in the algorithm is in line 2. If there are no ongoing SISCANs, we start the new scan at the location of the last SISCAN that finished. This way, the new scan can reuse any bufferpool pages left behind by the scan that finished. Technically, we should start the new scan several pages before the last scan's location, depending on how many pages of this scan we expect to be left in the bufferpool. In the location/time diagram

```
1 fct findStartLoc( scan t, scanset S )
2    if S is empty, return most recently finished
     scan's location;
3    G := set of SISCAN groups that overlap t's
         scan range;
4    if G is empty, return t's regular start loc;
5    for each group g in G
6       loc(g)  := best start location for t in the
                   group (as discussed in text);
7    endfor
8    return loc(g) with best sharing among all g;
9 endfct;
```

Figure 13. Finding SISCAN starting location

in Figure 8, such terminated scans would show up as vertical traces (since they are no longer moving forward) and with a wide (hashed) sharing area (since no other SISCANs are running). So, by keeping the last terminated SISCAN in the data structure with speed set to 0, this special case could be handled by the general algorithm.

Once the new SISCAN's location is determined, its anchor is set to either its start key/RID (if it cannot start with some other SISCAN) or to some other SISCAN's anchor (if it starts at that other SISCAN's location) with the offset being 0 or the other SISCAN's offset, respectively.

7. SISCAN LOCATION UPDATE

In this section, we discuss how a SISCAN location change influences the various SISCAN attributes, how the speed of SISCANs can be adaptively controlled, and how the bufferpool page priorities are influenced by the scan states.

7.1 SISCAN attribute updates

As discussed in Section 4.2, the ISM is called periodically to update the SISCAN location in its data structures. This call can be performed every time a new table page is accessed or at every *x* page accesses. There is a tradeoff: if the updates are too frequent, the accuracy of the SISCAN location in the data structure is high but there is extra overhead at every page change. On the other hand, if the updates are too infrequent, this overhead is lower but the location accuracy decreases, leading to less optimal decisions for SISCAN start locations and speed control. In our implementation, we update locations at fixed intervals (every 16 32k pages, to be precise) but we plan to investigate more adaptive schemas in future work.

At every location update call, the following SISCAN fields are modified: location, remaining pages, speed, and anchor/offset information. In our implementation, we set the speed to (pages read since last update)/(time since last update). Since this speed estimate depends only on the near past, it can capture speed fluctuations caused by interactions with other ongoing scans.

The anchor offset is simply incremented with one important exception. When a SISCAN A's new location is equal to some other SISCAN B's anchor, A sets its anchor to B's anchor and sets its anchor offset to (A's offset)+(B's offset). This way, the anchor/offset-based partial SISCAN order discussed in Section 0 now also has an order between A and B. Or in other words, A and B are now in the same anchor-group.

7.2 SISCAN speed control

No matter how smart the starting location of a SISCAN is picked, scans that were perfectly aligned at the beginning, may start drifting apart over time. This is caused by different index scan predicates and disk access interference. Without actively keeping SISCANs aligned by adjusting their speeds, the drift will lead to reduced bufferpool sharing over time.

We have addressed the drift problem in our previous work on bufferpool sharing for table scans [8]. We will therefore just summarize this work here, as the translation to index scans is very straightforward. In [8] we propose to group table scans together based on similar speeds. The table scan in the front of the group (in scan direction) is designated as the "leader" and the scan in the back of the group is designated as the "trailer" of the group. In order to reduce drift, we measure periodically whether the distance between leader and trailer becomes larger than some threshold (typically less than two prefetch extents). If it does, we insert some wait operations for the leader to let the trailer (and the rest of the group) catch up. The wait duration is determined by the measured speeds of the table scans in the group and the distance between leader and trailer. Technically, the wait is inserted during an `updateSISCANLocation` call; the call therefore simply appears to take a longer time to the calling SISCAN.

We now show how the partial ordering on SISCANs can be used to determine leaders and trailers (Figure 14). Let us consider the groups in Figure 6 again and assume the bufferpool size is 50 pages. We know the partial order between the scans is A < B, B < C, C < D, and E < F. We also know the scan distances are d(A,B)=40, d(B,C)=10, d(C,D)=15, and d(E,F)=20. We first sort the scan pairs by distances and then add them in increasing order to form larger scan groups until the sum of the extents of all scan groups reaches the bufferpool size (lines 3-8). Here, we first add (B,C), then (C,D), resulting in the group (B,C,D), and finally we add (E,F) which does not merge with the other groups since no scans are in common. The final result are the groups (A) with extent 0, (B,C,D) with extent 25, and (E,F) with extent 20 and an overall extent of 45 (which is smaller than the bufferpool size). For each of these groups we have a leader (the largest by location) and a trailer (the smallest by location). A is leader and trailer of the first group, B is trailer and D is leader of the second group, and E is trailer and F is leader of the third group (lines 9-11). Thereby we have mapped this scenario to the scenario of table scan groups with leaders and trailers in [8] and can now apply the throttling technique discussed there.

Slowing down a scan operation in order to improve query response time may seem counter-intuitive at first. However, let us consider what would happen if the leader was not slowed down. In this case, the distance between the leader and other scans in the group would keep increasing until the distance is so large that bufferpool pages can no longer be reused between them. Once this happens, every page that the leader has read, needs to be re-read by the other scans in the group – the I/O cost has doubled. This additional I/O in return also affects the leader itself negatively since its I/O requests get delayed more due to a busier disk.

Of course, there is a limit to the amount of throttling of a scan. If all scans in a group are progressing very slowly relative to the leader, it may not make sense to keep slowing down the leader. For this reason, we keep track of the accumulated slowdown of each SISCAN (this can be stored in an additional field in the ISM data structure). If a SISCAN was slowed down for more than 80% of its estimated total scan time, it is not slowed down anymore until it finishes. By limiting the per-scan delay, we can enforce some amount of fairness in that no single scan may get delayed

```
 1 fct findLeadersTrailers( scanset S )
 2    R := empty set;
 3    while sum of extents of groups in R <
      bufferpool size
 4       pick a pair (x,y) not in R with x<y and
                   d(x,y) minimal;
 5       if (w,x) in R, replace it with (w,x,y)
 6       elsif (y,z) in R, replace it with (x,y,z)
 7       else add (x,y) to R;
 8    endwhile
 9    for each group (x, …, y) in R
10       mark x as trailer and y as leader;
11    endfor
12 endfct;
```

Figure 14. Classifying SISCAN leaders and trailers

indefinitely in order to improve bufferpool sharing for other scans. The "80%" threshold is based on our experience with various workloads. One interesting extension would be to make this threshold dynamic by taking into account query priorities and machine characteristics.

7.3 Adaptive bufferpool page prioritization

As discussed in Section 4.2, each SISCAN releases pages that are processed with a priority determined by the ISM rather than with a fixed priority. The idea is to assign a high priority to pages in the bufferpool that are needed soon by members of the group and a low priority to pages that will not be needed soon. Similar to [8], we propose to use the leader/trailer status (as determined in the previous section) to decide on the priority. Leader SISCANs assign a high priority to their finished pages because other SISCANs that need the same page are following. Trailer SISCANs assign a low priority to their finished pages because there is a big gap until the next following SISCAN and so the page would get discarded anyway at some point in time before the following SISCAN can read it.

8. EXPERIMENTAL RESULTS

This section discusses the experimental results obtained from our prototype implementation in IBM's DB2 Universal Database (UDB). The implementation of SISCAN operators and the ISM is built on top of the table scan bufferpool-sharing infrastructure for DB2 UDB described in [8]. That infrastructure allows normal table scans' speed and bufferpool page prioritization to be controlled via a table scan sharing manager (much like the ISM). The table scan sharing manager provides functions similar to the ISM: "start table scan", "end table scan", "update table scan location", and "get page priority". We extend these functions to also accept index scans. Internally, the table scan sharing manager keeps track of groups of table scans. In the new prototype, the ISM keeps track of groups of SISCANs and their anchors and offsets. Since we view SISCANs just as another type of scans, we can reuse all of the grouping, leader/trailer classification, throttling and page prioritization algorithms. The implementation effort is mainly limited to the algorithm for determining the SISCAN start location and the modified SISCAN process logic (cf. Section 4.2).

While the presented algorithms work for any type of index-based scans, we have decided to implement our prototype first for MDC block index scans [1] [2] since these are increasingly popular in customer settings. The block size is configurable. In our experiments, we set it to 16 pages with a page size of 32 Kbytes. We perform calls to `updateSISCANLocation` at every extent boundary.

We evaluated our prototype on two hardware platforms: (1) an HP Integrity rx5670 server powered by 4 Intel Itanium 2 processors running at 1GHz on HPUX OS with 15GB main memory and FAStT storage manager and (2) an 8-node p660 AIX cluster with each node powered by 4 PowerPCs running at 600MHz with 8GB main memory and 16 SSA disks. The reason why we use two platforms is because HP-UX allows us to measure disk seek times, while AIX allows us to measure disk I/O wait times. Both are key indicators for evaluating our algorithm.

The database for the experiments is a 100GB TPC-H [25] database. The bufferpool size is about 5% of the database size.

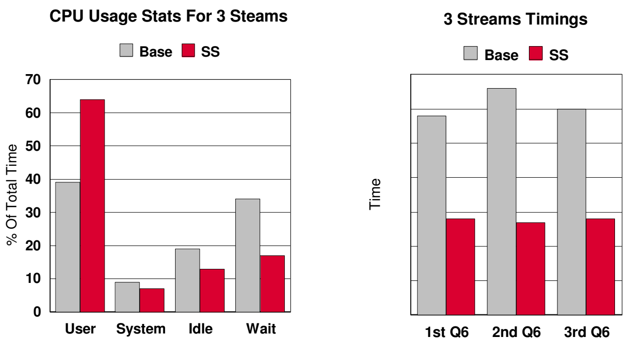

Figure 15. Timings for 3 Q6 streams (I/O intensive)

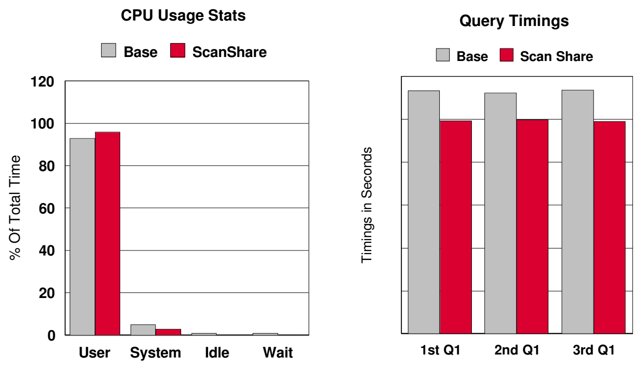

Figure 16. Timings for 3 Q1 streams (CPU intensive)

We performed three sets of experiments: single stream TPC-H runs (to investigate the overhead of our SISCAN extensions), runs with single TPC-H queries that are started in a staggered way and full TPC-H throughput runs. In each set of experiments, we obtain measurements for vanilla DB2 and our prototype over five runs and average the results. Due to space constraints we only report on the latter two experiments but note that the observed overhead in the first experiment was well below 1% of the end-to-end time.

8.1 Staggered Q1/Q6 query performance

For the single query runs we chose TPC-H queries Q1 and Q6. While the former is very CPU-intensive, the latter is highly I/O-intensive. For both tests, the queries were started 10s apart, so that the runs are overlapping. The runs were performed on the AIX box which displays the I/O waits in the `iostat` readings.

Figure 15 shows the timing measurements for Q6. The left graph shows the distribution of CPU time spent in user time, system time, idling, and in I/O wait. "Base" indicates the distribution for the plain DB2 runs. It shows that in this configuration, a large amount of time is spent in I/O wait. For our prototype (indicated with "SS"), I/O wait time is reduced by half, indicating that our algorithm is making pages available to the SISCAN processes faster by improving bufferpool sharing and reducing seek times. Similarly, the idle time is reduced showing better CPU utilization. In turn, the user time component increases, indicating that with pages now more easily available, the scan processes are able to do more useful work. The right graph shows the execution times for each of the three Q6 runs. Each of the runs gains more than 50%

due to the sharing of bufferpool pages with the middle run gaining most due to its sharing potential being highest.

In contrast, Figure 16 shows the same types of results for staggered runs of TPC-H query Q1 – a very CPU-intensive query. As can be seen in the left graph, the amount of time spent in I/O idle and wait is negligible compared to the user slice. Despite that, improved bufferpool sharing is able to reduce even these small amounts further. Also, since fewer system read calls are now being made, the system time is going down as well. All these reductions lead to more time available for the scan processes to do useful work (as shown in the user time slice). Thus, as the right graph shows, even in these sub-optimal conditions, our prototype improves the runtime of each Q1 noticeably.

8.2 TPC-H throughput performance

During a TPC-H throughput run, five query streams are started at the same time. Each stream consists of a predefined order of 22 queries. Since the order is different in each stream, different queries overlap at different points in time. In the 22 queries, there are 18 block index scans and 29 table scans.

In our experiments, we measure both the end-to-end time gains of TPC-H, as well as gains per stream and disk I/O gains. These runs were done on the HP box since HP-UX shows seek times as part of the `iostat` readings. Table 1 summarizes the overall results as improvements over the vanilla DB2 measurements. The first column shows that end-to-end runtime of the TPC-H throughput runs was improved by 21% with our prototype. These gains are largely due to better bufferpool usage between index scans. This is visible from the second and third columns of Table 1 which show that the average disk reads (i.e., the amount of data read from disk) and the average disk seeks are reduced by 33% and 34%, respectively. The reduced disk utilization may be used to scale to a larger number of streams with the same hardware.

Table 1. Performance results for 5 stream TPC-H

End-to-end gain	Avg. disk read gain	Avg. disk seek gain
21%	33%	34%

Figure 17 shows the amount of data read from disk over time. Each bar in the graph stands for a fixed unit of time and the y-axis indicates the amount of data read during that time unit. The graph entitled "Base" shows the read characteristics for the unmodified DB2 version during the TPC-H run. It shows some amount of jitter which is due to different queries being executed in the different streams over time. Our prototype (indicated by "SS") shows the same jitter but the amount of reads is lower in most time intervals and the run ends sooner.

Similarly, Figure 18 shows the progress of disk seeks (y-axis) per time unit (x-axis). As with the data reads, the disk seeks is very much reduced during most time intervals. This is because with our prototype, scans are synchronized and thus tend to reuse the pages demanded by each other. Thus, although each scan ends up demanding the same page set as with the base code, they demand it in such an order that the disk has to seek less often to satisfy them.

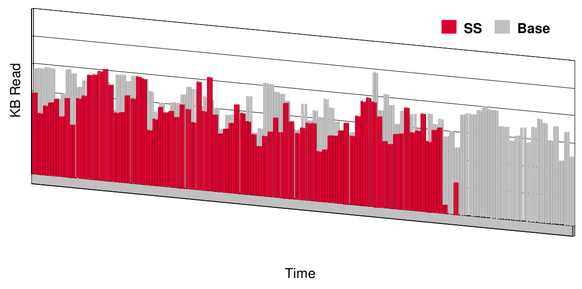

Figure 17. Disk read amounts over time for plain DB2 and our prototype

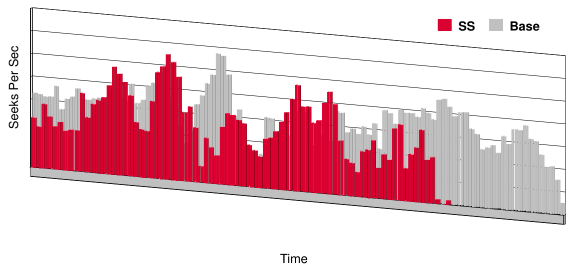

Figure 18. Disk seeks over time for plain DB2 and our prototype

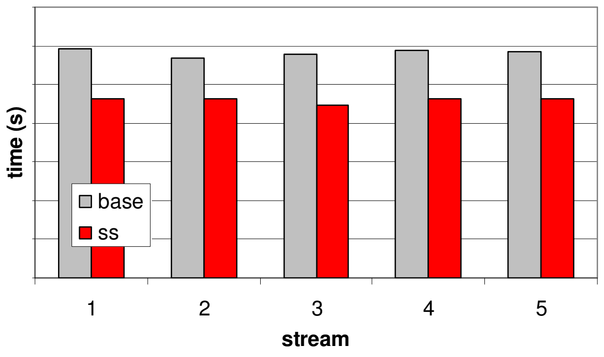

Figure 19. Per stream gains (5 stream TPC-H)

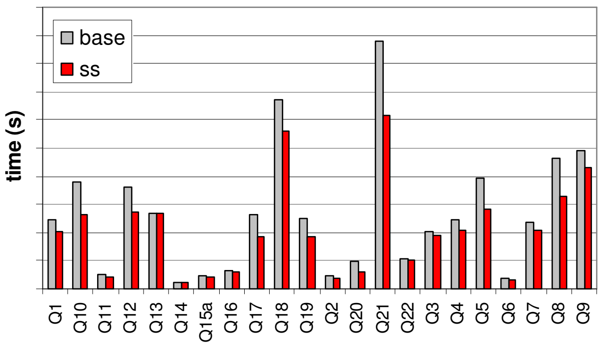

Figure 20. Per query gains (5 stream TPC-H)

In order to investigate the "fairness" of our algorithm, we measure the per-stream and per-query gains. The per-stream results are shown in Figure 19. We can see that each stream gained similarly from the improved bufferpool sharing. The per-query results are shown in Figure 20. The chart shows the average execution time for the 21 queries for the 5 TPC-H streams. While the gains from the bufferpool sharing vary with queries, no query shows a negative effect. This is remarkable given the fact that we slow down individual SISCANs whenever drift occurs. Apparently no single query's SISCANs get "punished" with throttling over the others. Instead, throttling seems to be distributed across the queries for mutual benefit. We also point out that certain queries benefit more from bufferpool sharing than others (e.g., query 21). This is due to the amount and range size of the index scans present in the queries.

9. CONCLUSION

In this paper, we presented a technique to improve caching between concurrent index scans on an RDBMS. At the core of the idea is a new index scan sharing manager that keeps track of ongoing scans and decides where to start a new index scan and how to throttle ongoing scans. Since our extensions treat the index itself as a black box and requires only few extra calls to the sharing manager, integration with an existing architecture is very easy. Our prototype in DB2 UDB shows significant gains for the TPC-H benchmark, both in terms of reduced disk activity and in end-to-end timings.

In the future, we plan to investigate sharing for other types of index scans and tighter coupling of SISCANs and the query optimizer.

10. REFERENCES

[1] Padmanabhan, S., Bhattacharjee, B., Malkemus, T., Cranston L., Huras, M., "Multi-Dimensional Clustering: A New Data Layout Scheme in DB2", Proceedings of SIGMOD 2003.

[2] Bhattacharjee, B., Padmanabhan, S., Malkemus, T., Lai, T., Cranston, L., Huras, M., "Efficient Query Processing for Multi-Dimensionally Clustering Tables in DB2", Proceedings of VLDB 2003

[3] Teradata Corporation, "Introduction to Teradata RDBMS V2R5.0", December 2002

[4] Kyte, T., "Expert Oracle", Apress, 2005

[5] Winter Corporation, "Peak Workload, DW 2005", http://www.wintercorp.com

[6] E. Grochowski; R.D. Halem, "Technological impact of magnetic hard disk drives on storage systems" IBM System Journal,Vol 42, No 2, 2003

[7] R. Bhashyam. Technology challenges in a data warehouse. In Proc. Int. Conf. on Very Large Data Bases, pages 1225–1226, 2004.

[8] Lang, C., Bhattacharjee, B., Malkemus, T., Padmanabhan, S., Wang, K.,"Increasing Buffer Locality for Multiple Relational Table Scans through Grouping and Throttling", Proceedings of the ICDE 2007

[9] http://www-306.ibm.com/software/data/db2

[10] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In Proc. ACM SIGMOD Int. Conf. on Management of Data, pages 297–306, 1993.

[11] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In Proc. Int. Conf. on Very Large Data Bases, pages 439–450, 1994.

[12] J. Robinson and M. Devarakonda. Data cache management using frequency-based replacement. In Proc. ACM SIGMETRICS Conf., pages 134–142, 1990.

[13] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. IEEE Trans. Computers, 50(12):1352–1360, 2001.

[14] N. Megiddo and D. Modha. Outperforming LRU with an adaptive replacement cache, 2004.

[15] P. M. Fernandez. Red brick warehouse: a read-mostly RDBMS for open SMP platforms. In Proc. ACM SIGMOD Int. Conf. on Management of Data, page 492, 1994.

[16] T. Walter. Explaining cache — NCR CTO Todd Walter answers your trickiest questions on Teradata's caching functionality. (http://www.teradata.com/t/page/116344/)

[17] R. Bhashyam, "TPC-D - The Challenges, Issues and Results", NCR Corporation, SIGMOD Record 25(4) 1996: 89-93

[18] www.microsoft.com, "SQL Server 2005 Books Online"

[19] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: A simultaneously pipelined relational query engine. In Proc. ACM SIGMOD Int. Conf. on Management of Data, pages 383–394, 2005.

[20] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In Proc. ACM SIGMOD Int. Conf. on Management of Data, pages 249–260, 2000.

[21] J. Shim, P. Scheuermann, and R. Vingralek. Dynamic caching of query results for decision support systems. In Proc. Int. Conf. on Scientific and Statistical Database Management, pages 254–263, 1999.

[22] M. Zukowski; P.A. Boncz; M.L. Kersten, "Cooperative Scans", CWI Report 2004, INS-E0411, ISSN 1386-3681

[23] G. M. Sacco and M. Schkolnick. A mechanism for managing the buffer pool in a relational database system using the hot set model. In Proc. Int. Conf. on Very Large Data Bases, pages 257–262, 1982.

[24] H. Chou and D. DeWitt. An evaluation of buffer management strategies for relational database systems. In Proc. Int. Conf. on Very Large Data Bases, pages 127–141, 1985.

[25] http://www.tpc.org/tpch