# Block oriented processing of Relational Database operations in modern Computer Architectures

Sriram Padmanabhan Timothy Malkemus Ramesh Agarwal
Anant Jhingran*
Data Management Group, IBM T.J. Watson Research Center, Hawthorne, NY
{srp,malkemus,ragarwal,anant}@us.ibm.com

## Abstract

*Several recent papers have pointed out that database systems are not well tuned to take advantage of modern superscalar processor architectures. In particular, the Clocks-Per-Instruction (CPI) for rather simple database queries are quite poor compared to scientific kernels or SPEC benchmarks. The lack of performance of database systems has been attributed to poor utilization of caches and processor function units as well as higher branching penalties. In this paper, we argue that a block oriented processing strategy for database operations can lead to better utilization of the processors and caches generating significantly higher performance. We have implemented the block oriented processing technique for aggregation, expression evaluation, and sorting operations as a feature in the DB2 Universal Database system. We present results from representative queries on a 30 GB TPC-H database to show the value of this technique.*

## 1 Introduction

A number of recent papers have studied the detailed performance of database workloads on modern superscalar processor architectures and concluded that database systems are not effectively utilizing all the capabilities of these architectures [2, 3, 14]. One important measure of the performance of a new architecture is the Clocks-Per-Instruction (CPI) metric when executing a benchmark. Ailamaki *et al* [2] report that four of the major commercial database systems have rather high CPI values (between 1.2 and 1.8) for very simple database queries. In layman terms, this means that the processor is able to retire one instruction every 1.2-1.8 clock cycles on the average. Modern processor architectures use simple RISC style instruction sets where most instructions are completed in one cycle. In each cycle, the processor will invoke several function units such as instruction prefetch, load-store, fixed and floating point arithmetic units, and branch prediction. Processors typically fetch and execute several instructions (one of each type) from the instruction stream in a clock cycle which means that peak CPI rates can be a small fractional value. For instance, the CPI can be 0.5 if the processor is able to retire two instructions per cycle. However, a processor may not be able to invoke all the function units at every cycle for various reasons such as memory stalls, lack of instructions, or dependencies. The low CPI value for simple queries indicates that database systems are particularly ineffective in taking advantage of modern superscalar processor capabilities.

The results from [2] extend and corroborate the observations of several other architecture performance studies based on database workloads in the recent past [3, 10, 11, 14]. These papers have individually studied processor performance (including multiprocessors) on OLTP and DSS workloads. They are in general agreement that database workloads incur larger number of stalls and have poor CPI values compared to scientific workloads. In particular, some of the studies point out that DSS workloads are affected by stalls due to data cache misses in Level 2 cache, instruction cache misses in Level 1 cache, or functional unit resource stalls.

In related work, several papers have proposed new cache-conscious techniques for specific database operations and indexing [5, 15, 17]. Techniques such as blocking, loop fusion, and data clustering were found to be important for improving cache locality and performance of aggregation and join algorithms [17]. [5] presents memory-resident data structures and cache-conscious radix join techniques. A variety of algorithms for fast sorting have been proposed for optimizing cache performance and processor utilization [1, 12, 13]. In particular, [1] describes a high-performance sort using radix techniques that tries to take advantage of superscalar processor features and cache memory. The above papers have tended to focus on specific operations while studying techniques for performance improvement. However, there has been no effort so far towards a general purpose technique of processing different operations in a query that is effective for modern computer systems. Note that the above papers are most relevant but by no means exhaustive. Earlier work on main memory database implementation techniques [7], database machines, and extended vector processors [18] are also influential. However, these papers are less relevant when considering modern superscalar processor architectures.

In this paper, we propose a block oriented processing technique for database operations based on the observations from past work [1, 17]. The block oriented scheme is particularly motivated by the DSS and complex query workloads used in data warehouses and data marts. We use the *Sort*, *Aggregation* and *math expression evaluation* operations as the initial basis for our methodology. These block

*Current Address: IBM Almaden Research Center, San Jose, CA, USA



1063-6382/01 $10.00 © 2001 IEEE

oriented operators and query processing schemes have been implemented in IBM's DB2 Universal Database (UDB) system [9]. The main feature of this technique is to construct a block of intermediate records that can be used by one or more database operators. A block descriptor data structure is used to describe the contents of the block in terms of input and output fields. We also propose block oriented algorithms for processing database operations. Once the block memory area is full, these new algorithms are invoked to perform the appropriate operations. We contrast this scheme with the current processing strategy that is typical of most database systems. Performance experiments are conducted using a 30 GB TPC-H benchmark[6] database system to study the improvement resulting from the new processing strategy. Performance is improved because of better utilization of processor and memory resources due to (i) better instruction pipelining and branch avoidance, (ii) cache cognizant processing using tight loops to minimize data cache misses, (iii) reduction in the instruction pathlength and function-call overhead since certain parts of the code are invoked once per block of records, and (iv) better instruction cache utilization since the same instruction stream is repeated for a whole block of records.

The rest of the paper is organized as follows. Section 2 describes the current record-oriented, iterative processing model and its deficiencies in more detail using a TPC-H query example. Section 3 describes the block-oriented processing methodology and the architecture cognizant features of this scheme. Section 4 describes the results of our performance experiments comparing the new and old schemes. Finally, Section 5 presents a conclusion and a discussion of future work.

## 2 Motivation

Typically, relational database management systems have implemented a *demand-driven*, pipelined query processing model that can be executed by a single operating system process or thread[16]. This approach has several advantages such as avoiding inter-process communication between operators, avoiding process synchronization and scheduling, minimizing data copies, keeping only the current data items in memory, and performing lazy operator evaluation. Below, we illustrate this processing strategy using a complex query example and then list a number of deficiencies of the method for modern processor architectures.

We illustrate the query execution tree and the processing strategy using a sample query (query 1) from the TPC-H decision support benchmark[6].

Figure 2 shows the execution plan for this query in DB2 and it is fairly typical of most DBMSs. The leaf operator of this tree is a SCAN operator on the base table LINEITEM. The LINEITEM rows are filtered by the PRED operator which applies the selection constraint on L_Shipdate. The qualifying rows are also projected to remove unnecessary attributes and forwarded to the SORT operator. The SORT operator is included in this query plan to enforce the GROUP BY constraint on L_ReturnFlag and L_Linestatus. Note that an alternative technique to SORT is Hash[8]. The output of the SORT (or Hash) based Grouping is fed to the MATH and subsequently to the Aggregation operators. In the figure, we have expanded the Math to show the actual math operations that are performed in this query. We have also expanded the AGG block to show the individual aggregation operations. Note that the "Average" Column Function is obtained by dividing the Sum by

```
Select
 L_Returnflag, L_Linestatus, Sum(L_Quantity)
                            AS SUM_QTY
 Sum(L_Extendedprice) AS SUM_BASE_PRICE,
 Sum(L_Extendedprice *(1-L_Discount))
                            AS SUM_DISC_PRICE,
 Sum(L_Extendedprice *(1-L_Discount)*(1+L_Tax))
                            AS SUM_CHARGE,
 AVG(L_Quantity) AS AVG_QTY,
 AVG(L_EXTENDEDPRICE) AS AVG_PRICE,
 AVG(L_Discount) AS AVG_DISC,
 COUNT(*) AS COUNT_ORDER
From LINEITEM
Where L_Shipdate <= DATE '1998-12-01' -
                                INTERVAL '60' DAY
GROUP BY L_Returnflag, L_Linestatus
ORDER BY L_Returnflag, L_Linestatus;
```

**Figure 1. TPC-H Query 1**

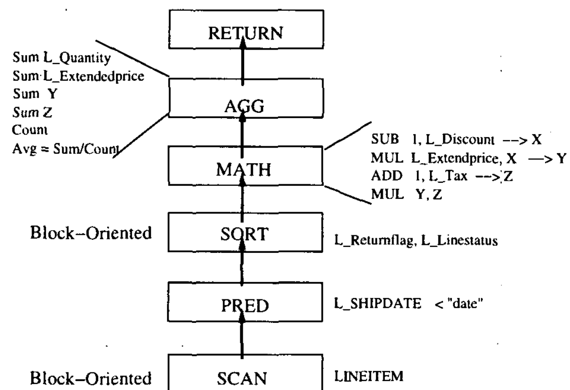


**Figure 2. Query Execution Plan for TPC-H Query 1**

the Count. All the aggregation operations are performed for each distinct group, i.e., the Sum, Count, and Average accumulators are reset at the beginning of each distinct group. Finally, the results of the aggregation are returned to the application.

DB2 already supports an improved execution strategy for processing this query, called *early aggregation* [4, 8]. In this scheme, the MATH and Aggregation operations are performed as a Predicate (PRED) of the SORT operation by identifying the group for the record in SORT. A Final Aggregation step is used to perform merging of grouping streams and computation of Averages from Sums and Counts if required. This strategy reduces the SORT cost by removing duplicates inside sort at insert time. Thus, the intermediate aggregation results are likely to fit in sort memory and there will be no need for I/O and merging.

At present, most database systems perform block based operations only when I/O is involved. The Scan and Sort operations are able to work on a page or block of pages at the time of performing the I/O. However, most of the other steps even in these operations are based on individual records.

This query is one of the longer running queries in TPC-H. The complete list of TPC-H benchmark results are available at the TPC web site [19]. A quick review of the

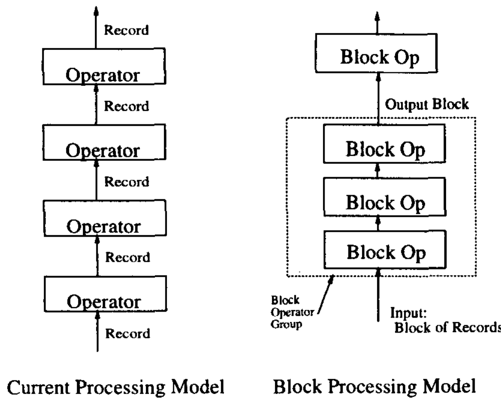


**Figure 3. Comparison of the record-oriented and block-oriented processing strategies**

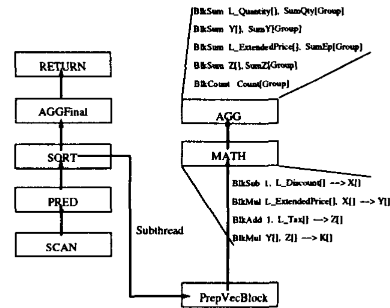


**Figure 4. Block Operator Query Plan for TPC-H query 1**

current reuslts will show that the execution time of query 1 is usually a factor of 1.5 larger than the arithmetic mean of all the query execution times. Typically, the query has a high CPU utilization (usually 100%) due to the intensive set of math and aggregation operations and also requires a fairly high I/O throughput due to the scan of the large LINEITEM table. Thus, any technique that can improve this query tends to have a real performance value for decision support and complex query applications.

We identify the following as important deficiencies of the current query execution schemes. First, the data and instruction cache utilization can be poor especially when the query tree contains a long pipeline of operators. These query plans could result in repeated loading and flushing of per-operator data structures and instructions as each record is processed. Second, the branching overhead tends to be high and leads to frequent processor stalls. Each operator checks for nullability, datatypes, comparison, overflow, underflow, error conditions, etc. per record. This will lead to wasted computation as well as branch stalls. Finally, the instruction counts are high because they are repeated per record. The processing strategy also generates significant function call overhead for each record.

We believe that there is room for improvement in the iterator model of processing complex queries by working at the level of a block of records and describe this methodology in more detail in the following section.

## 3 Block Oriented Processing

Let us reconsider the query plan for the Example TPC-H query 1 (Figure 1) as illustrated by Figure 2. Notice that the dataflow is from the leaf node (or nodes in multi-table query plans) to the root of the tree. Suppose, each lower-level, *producer* operator generates a block of intermediate tuples before returning control to the parent, *consumer* operator. Then, the parent can operate on the entire block of records and generate a new or modified block of records for its ancestors in the tree. The current database operations need to be redesigned to operate on blocks of records. To accomplish this, we borrow the concepts of tight loops and stride based processing from the scientific and vector computing disciplines. Vector style processing, wherever possible, can amortize condition checking and branching overhead resulting in very effective utilization of the CPU and memory hierarchies. The technique is also very effective for multiprocessors since each processor can work on its own partitioned block of records. Figure 3 compares the old and proposed processing strategies to highlight these issues. For the block processing strategy, we introduce the notion of a *block operator group*. All the operators in a group manipulate the same block of records and pass the updated block to its consumer. Each operator can modify existing columns or generate new columns. Note that the space for new columns should be pre-allocated in the block. Reusing the same block across a number of operators is an important performance criterion since it reduces or avoids unnecessary data copies.

We illustrate the main ideas using the MATH and AGG operators in TPC-H Query 1. Figure 4 shows the new query plan with the details of the sub-thread performing the math and aggregation operations. In the new strategy, the SORT operator invokes the PREPBLOCK operation to generate a block of tuples and initialize a data structure called the *Block Descriptor* appropriately. The block is passed as an operand to the *Block* MATH operators and subsequently to the *Block* Aggregation operators. The result of the last Block AGG operator is a tuple for each grouping value and this could be inserted in a new block since the size reduction is likely to be substantial. Note that we are able to prepare a single block and use it to process a group of operations.

A block area is simply a storage space for a group of intermediate records. The records are stored consecutively in this space. Each record is of a fixed length and contains the input attributes and space for intermediate result attributes. The SQL compiler will reuse existing column space for intermediate results whenever the column is not needed for future computation. In the case of our example query, the block area will contain (i) input columns such as L_Extendedprice, and L_Quantity, (ii) intermediate expression results such as (1 − L_Discount), and (L_Extendedprice × (1 − L_Discount)), and (iii) pointers to specific aggregate counters based on the Group By columns.

The block area is allocated from the Sort memory heap in this example. Both performance and available space

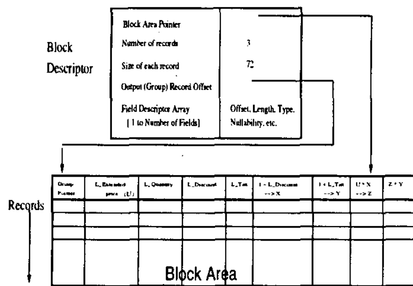


**Figure 5. Block and Descriptor for Aggregation in TPC-H Query 1**

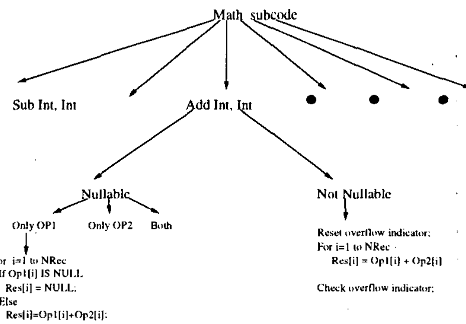


**Figure 7. Code execution tree for Block Math operations**

considerations are important in choosing the block size. Performance is crucial and we will show in Section 4 that the Level 1 (L1) data cache size is an important factor in determining an optimum block size. In most of our discussion, we assume that the block is utilized in *row-major* fashion, i.e., the fields of a given record are contiguous. However, in certain situations, a record might be very large and the optimum block size may not be able to contain many records. In these situations, a larger vector block can be allocated while still maintaining good cache utilization if the space can be utilized in *column-major* order, i.e., a column vector is assigned contiguously. Since a block operation usually operates on two input columns and generates one result column, only the vectors for the three columns need to fit in the cache for that operation.

### Block Descriptor

The block descriptor is a data structure that describes the schema of the block. It is shown in Figure 5. The descriptor contains a pointer to the block, the size of the block, the size of each record, number of records in the block, and details of the fields and their offsets within each record. The descriptor is initially set up by the SQL Compiler and is updated by the block initialization routine. The descriptor is passed to a consumer operation which might in turn create a new block or modify the existing one in case the same block is to be used for subsequent operations.

### Block oriented Operators

As the name implies, a block oriented operator expects one or more vectors of data elements as input and can generate a vector or scalar as output. Figure 6 describes the format of the Block MATH instruction that we implemented in DB2 UDB. The instruction consists of the specific *opcode* which could be a datatype dependent flavor of Add, Subtract, Multiply, Divide, or unary Negation. The operands consist of the block descriptor, offsets of the two input fields, and the offset of the result field in each record. The flag fields associated with each operand indicate whether the operand is a constant. One of the operands of the math operation could be a constant(e.g., X+5) in which case its offset descriptor indicates the location of the constant value.

The operation is implemented as a specific loop over the elements of the input columns with emphasis on minimizing overheads. Ideally, there is a loop for each combination of attributes that could be specified: operand data types, operation type (math, sum, count, etc.), nullability. Figure 7 shows some of the key execution steps in the block-oriented Math operation. First, the Math subcode is used to invoke the specific routine. This routine, e.g. Add two integers, will check for Nullability of its operands. Note that this condition is checked only once per block of records. If both operands are Not Null, then the specific inner block (say integer Addition) which is a very tight loop over the block is invoked as shown. Each loop is coded as tightly as possible, to make good use of the machine's instruction pipeline and to minimize the cycles needed for each iteration. We have found that for architecture supported datatypes such as integers, the loop for block oriented math (Figure 7) can be accomplished in 10 machine instructions and fewer than 10 cycles (RISC System/6000) per iteration. It is not possible to reduce the instruction count below this number in current processor architectures. The instruction count varies slightly for different cases depending on the data types of the operands, nullability, overflow/underflow, etc.

The pseudo-code in figure 7 shows one example of amortizing condition checks; in this case for overflow (or underflow). Typically, overflow is required to be checked for each math operation. Naively, this can be done by checking the overflow condition inside the loop during each execution of the math operation. However, this introduces a condition check in the loop and degrades performance for two reasons. First, it discourages compilers from performing *loop unrolling*. Second, the instruction prefetching and pipelining can be adversely affected due to branch prediction misses. We recognize that overflow occurs very rarely in practice. Hence, we can amortize the overflow checking cost by performing it only once for the whole block. Usually, overflow will not occur and we have improved the execution efficiency of the loop. If overflow is detected, then another loop can be executed with the overflow check inside it in order to detect all the overflow operations.

Figure 8 shows the instruction format of the Block Aggregation operation. All the standard column functions, viz., SUM, COUNT, MIN, MAX, can be executed in a block-oriented fashion. Also note that AVERAGE can be computed from the SUM and COUNT. The operands include the input block descriptor, the offset of the input field

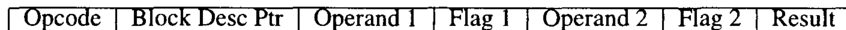


**Figure 6. Block Math Instruction Format**

**Figure 8. Block Aggregation Instruction Format**

for aggregation, the offset of the group by pointer, and the offset of the aggregation counter in the output block. The output block is the result of group by aggregation. Suppose the operation is SUM. It is performed by looping over the input block and using the grouping column index to add the specific input to the accumulator in the corresponding group record in the output block. Once again, there are specific sub-blocks of code to deal with datatype and nullability.

Comparing query plans between Figure 2 and Figure 4, the reader can notice that all the math and aggregation operations have become block-oriented in the sub-thread of SORT.

### Performance Discussion

This methodology is intended to utilize modern computer architecture features and address the issues we raised in Section 2.

**Cache Consciousness** Block operations can maximize the effective utilization of data caches by carefully choosing the block size. Also, by reusing the same block of records across several operations, the effect of cache locality is further enhanced. Instruction cache locality is also enhanced significantly since the same inner loop of the code is executed repeatedly for a block of records.

**Better Instruction Pipelining** Database operations are required to check a variety of factors such as datatypes, nullability, operation status, error conditions, etc. Usually, this makes the code fairly complex and requires a lot of condition checking constructs. It is important to perform the condition checks only once for a group of records and then route to the appropriate function that performs the core operations. The block operation functions are coded with close attention to minimizing branches inside loops. Thus, inner loops of operations are mostly devoid of branch conditions enhancing instruction prefetching and pipelining in the CPU.

**Minimizing Instruction count and Function calls** Unlike the record oriented processing model which switches operation context repeatedly, the block processing style reduces the number of function calls significantly. Also, the overall instruction count for these operators is reduced significantly and leads to very streamlined code.

The block processing methodology can be applied in a similar fashion to other database operations such as Join, Predicate processing, and set operations. We are currently investigating robust block processing techniques in a real DBMS context for these other operations and should be able to report on their performance in the future.

| Experimental Setup | |
|---|---|
| System | IBM RS/6000 Model J40 |
| Number of CPU | 8 |
| CPU | 112 Mhz PowerPC 604 |
| L1 Cache | 16K (data) 16K (instruction) |
| L2 Cache | 1 MB |
| Memory | 2 GB |
| Storage capacity | > 100 GB |
| Disk | IBM SSA 4.5GB disks |
| Operating System | AIX 4.1.5 |
| RDBMS | DB2 UDB V5.2 |
| Database | 30-GB TPC-H |
| Sort Heap | 8 MB |

**Table 1. Important Experiment Configuration and Parameters**

## 4 Performance Experiments

We used an 8-way IBM RS/6000 Model J40 system and the DB2 UDB Version 5.2 database system to conduct performance experiments. Table 1 describes some important system and experiment parameters. The experiments were conducted using a 30 GB TPC-H database. Queries were issued against the LINEITEM table of TPC-H which is the largest and consists of 180 Million Rows and occupies approximately 22GB of data space in this database.

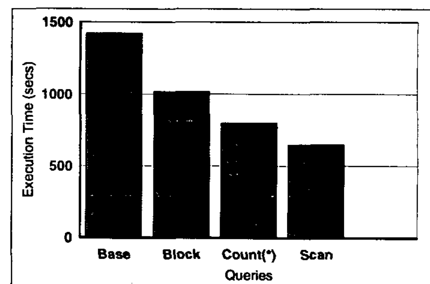


**Figure 9. Execution times for Base, Block, Count, and Scan queries on a 30GB TPC-H database**

Our first experiment is a comparison of four queries

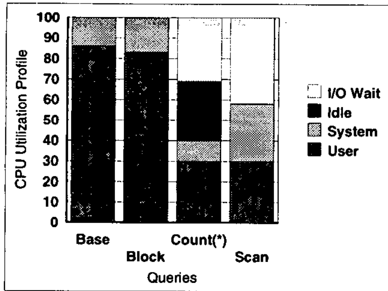


**Figure 10. CPU Utilization Profile of the baseline queries**

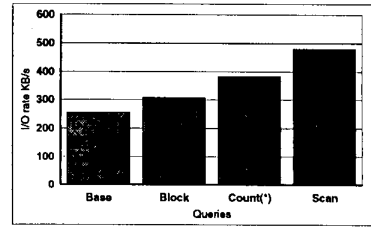


**Figure 11. Stabilized I/O Rate per Disk for the baseline queries**

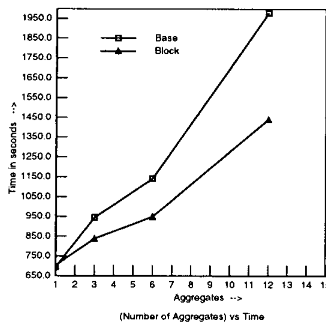


**Figure 12. Comparison of Block and Base query plans while varying the number of aggregations and expressions**

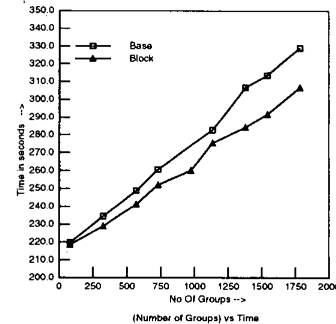


**Figure 13. Comparison of Block and Base query plans while varying the number of groups**

on the LINEITEM table to study the I/O and CPU utilization as well as to validate the performance of the block-oriented processing technique. The four queries are:

- Query 1 of TPC-H (Figure 1) using the enhanced but record oriented (*Base*) query processing strategy described in Section 2.
- Query 1 of TPC-H using the block oriented (*Block*) processing scheme described in Section 3.
- A simple Count query on LINEITEM which also applies the L_Shipdate predicate specified in TPC-H query 1. This query introduces the extra CPU processing for applying the predicate and counting all the rows that are selected.
- A scan of LINEITEM with a predicate that eliminates all rows. This shows the basic I/O time required to scan the table.

The scan and count queries are used as controls to study the baseline I/O and CPU processing requirements. Figure 9 shows the execution times of the four queries. The execution time for the Base query plan is 1426 secs while that of the Block plan is 1020 secs. The block processing strategy is able to shave off more than 400 secs in the execution time resulting in a 28% improvement. The execution times of the baseline Count and Scan queries are 800 and 650 secs, respectively. The difference between the baseline times and the Base query times are an indication of the math and aggregation complexity in query 1. Aso, note that the difference in time between the count query plan and the Base/Block plan is an indication of the CPU utilization for the math and aggregation operations in query 1. Using this metric, one can observe that the difference between the Block and count times (220 secs) is almost 3X better than the (626 secs) difference between Base and count.

Figure 10 shows a stable snapshot of the CPU utilization profiles of these four queries. Both the Base and the Block query plans peg the CPU at 100% utilization. This confirms one of our earlier assertions that CPU processing is an important aspect of complex queries. The CPU and I/O activities are completely concurrent since there is no I/O wait in these profiles. In contrast, both the baseline queries incur I/O waits. Some of the CPUs are also idle for the Count query indicating that the extra predicate and count processing reduces prefetching or it introduces additional waiting overhead. For the scan query, the CPUs are in I/O wait for 40% of the time since they have limited processing to perform.

Figure 11 shows the stable I/O bandwidth of one of the disks in the system while processing the four queries.

| DB Size | Ratio between Base and Block Operations | | |
|---|---|---|---|
| | Num. Instructions | I-Cache Miss | D-Cache Miss |
| 300 MB | 1.187 | 1.47 | 1.162 |
| 1 GB | 1.265 | 1.79 | 1.28 |

**Table 2. Hardware counter based comparison of the Non-block and Block oriented schemes**

The Base plan's disk I/O rate is 256 KB/s while the Block plan has a higher rate of 307 KB/s. The difference of 51 KB/s per disk reflects the improved CPU utilization arising from the block oriented processing strategy. Note that the I/O rate for the simpler scan operation is approximately 480 KB/s. Processing enhancements for the other query operators (PRED and SORT) could help further reduce the execution time for query 1 and increase the I/O rate.

Figure 12 shows a comparison of the execution times of the Block and Base strategies for queries similar to TPC-H query 1 as the number of math expressions and aggregation functions is increased from 1 to 12. For a single aggregation, both strategies perform almost similarly, indicating that the block processing strategy does not impose any significant overhead. The single aggregation in this case is COUNT which is a very simple operation. We expect the block processing to perform better than the base strategy for single math expression or aggregations such as SUM rather than COUNT. Hence, it can be used for queries with small number of aggregation and math operations. As the number of aggregates and expressions increase, the difference between the block and base schemes grows significant. For a query with 12 expressions and aggregations, the difference in execution time is more than 500s and results in a 28% improvement in execution time.

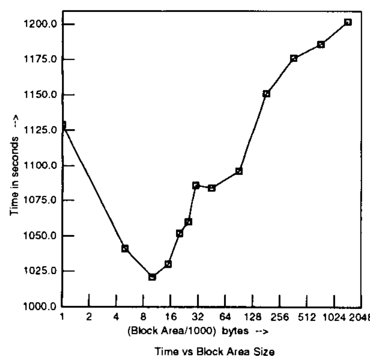


**Figure 14. Effect of cache on Block-oriented Operator performance**

The grouping columns of TPC-H query 1, L_Returnflag and L_Linestatus, are restricted to specific values and hence generate only four different groups. We would like to conduct a performance comparison of the block and base schemes for queries generating different numbers of groups. However, the standard LINEITEM table does not contain any attribute (other than date fields) whose grouping values can be varied from 1 to a few thousand values. Hence, we created a special 1GB version of the LINEITEM table with two artificial columns in them with distinct grouping values that could generate more than 2000 groups. Figure 13 shows the comparison of the execution times of the Block and base strategies for queries as the number of groups was varied from 1 to 2000. We fixed the number of aggregations and expressions to be seven for all the queries in this experiment. When the number of groups is small, the two strategies perform similarly. As the number of groups increases, the block strategy performs better and continues to improve. This result might at first seem counter-intuitive; it might appear that the block scheme will be penalized as the number of groups increases since the aggregations are group-specific. However, the results show that this is not the case even for several thousands of groups since the core operations are still being performed on blocks of records and using vector-style processing techniques.

One important tuning parameter for the new processing scheme is the size of the block. We hypothesized in Section 3 that its size is dependent on the data cache size. Figure 14 shows the execution times of TPC-H query 1 as we varied the size of the block. Note that the X-axis of this graph is plotted using a logarithmic scale. Initially, the execution times improve as the block size is increased from 1 KB to 10KB. This improvement results from better utilization of the CPU due to amortization of branching and function call overheads over larger blocks. Note that the entire block remains in L1 data cache for these sizes. From this point onwards, the execution times starts increasing (performance degrades) due to L1 data cache misses. Note that the optimum point, 10 KB, is slightly more than half the L1 data cache size of 16 KB. This indicates that the remainder of the data cache is being used for other data structures and data items. There is a small flattening of the curve around 32K. We speculate that the cache replacement is favoring the block operations briefly at this part of the curve which is about twice the data cache size. Beyond this point, the performance of the query degrades steadily as the block size is increased due to regular L1 cache misses. There might also be a Level 2 effect at larger vector block sizes but this effect is quite secondary since the performance is already very poor. This experiment indicates that the best block size is about half the size of the L1 cache.

We have also recently studied the behavior of the Base and Block algorithms using hardware monitors on an IBM RS/6000 S7A system. Table 2 shows the ratios of the instruction counts, the I-Cache misses, and the D-Cache misses between the two methods for TPC-H query 1 using two different database sizes. The results indicate that the block oriented scheme provides a significant improvement in the total number of instructions executed. Additionally, the I-cache miss rate is significantly lower for the Block scheme. We expect to report more results using the hardware counters on the CPI metric and other performance as-

pects in the future.

In summary, these results show that the block processing scheme is very effective in improving the CPU utilization of complex queries without incurring any significant penalty when processing simpler queries.

## 5 Conclusion

Relational database engines have improved significantly in their performance and functionality over the years. At the core of these systems is the relational query processing engine based on a record oriented processing model [16]. In this paper, we have argued that record oriented processing does not fully utilize modern computer architecture resources such as superscalar processing, instruction prefetching, pipelining, larger memories, and multiple cache levels. Past work has validated these observations and suggested the need for improvement in database query processing especially for complex queries. For these reasons, we have proposed a block oriented, vector style processing strategy and described the methodology in detail using math and aggregation operations. The main concept in our scheme is that the database operations take one or more blocks of intermediate records as input and generate output blocks, if required. Also, each operation is performed on the entire block of records using vector processing concepts. We have implemented this methodology in IBM's DB2 Universal Database system. Results of performance experiments conducted using a 30-GB TPC-H database show that the new techniques can produce close to 30% improvement in execution times of complex queries. The improvement is obtained due to better utilization of the CPU and memory resources as well as reduced instruction counts.

Most database operations can be implemented using the blocking and vector processing techniques. Currently, we are implementing the generic block oriented processing architecture for other database operators so that a complete query plan can be processed using this technique. A topic for future work is to enable the optimizer to freely choose between record oriented and block oriented processing of operations and sub-plans based on factors such as cardinality and cost.

**Acknowledgment:** The authors wish to thank members of the scalable data management group and the database development team for all their help and feedback.

## References

[1] R. C. Agarwal. A super scalar sort algorithm for RISC processors. In *Proceedings of the ACM SIGMOD Conference*, pages 240–247. ACM Press, 1996.

[2] A. Ailamaki et al. DBMSs on a modern processor: Where does time go? In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases*, pages 266–277. Morgan-Kaufman, 1999.

[3] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *Proceedings of ISCA 1998*, pages 3–14, 1998.

[4] D. Bitton and D.J. DeWitt. Duplicate record elimination in large data sets. *ACM Transactions on Database Systems*, 8(2):255, June 1983.

[5] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory Access. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases*, pages 54–65. Morgan-Kaufman, 1999.

[6] Transaction Processing Council. The TPC-H Benchmark Specification Version 1.2.1. Http://www.tpc.org/hspec.html.

[7] D.J. DeWitt et al. Implementation techniques for main memory database systems. In *Proceedings of the ACM SIGMOD Conference*, pages 1–8, 1984.

[8] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[9] IBM, Armonk, NY. *DB2 Universal Database Administration Guide, V6.* Http://www.software.ibm.com/data/db2/udb.

[10] K. Keeton et al. Performance characterization of a Quad Pentium Pro SMP using OLTP workloads. In *Proceedings of ISCA 1998*, pages 15–26, 1998.

[11] S. Khalafi and J. H. Moreno. Characterization of TPC-D benchmark on RS/6000 SMP machine. IBM TJ Watson Internal Report, 1999.

[12] P. A. Larson and G. Graefe. Memory management during Run generation in external sorting. In *Proceedings of ACM SIGMOD Conference*, pages 472–483. ACM Press, 1998.

[13] C. Nyberg et al. Alphasort: A RISC Machine Sort. In *Proceedings of ACM SIGMOD Conference*, pages 233–242, 1994.

[14] P. Ranganathan et al. Performance of database workloads on shared-memory systems with out-of-order processors. In *ASPLOS 1998*, pages 307–318, 1998.

[15] J. Rao and K. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases*, pages 78–89. Morgan-Kaufman, 1999.

[16] P. Selinger et al. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD Conference*, pages 23–34, 1979.

[17] A. Shatdal, C. Kant, and J. Naughton. Cache conscious algorithms for relational query processing. In *VLDB'94, Proceedings of the 20th International Conference on Very Large Data Bases*, pages 510–521. Morgan-Kaufman, 1994.

[18] S. Torii et al. Accelerating non-numerical processing by an extended vector processor. In *Proceedings of 4th International Conference on Data Engineering, ICDE*, pages 194–201, 1988.

[19] TPC-D Results. Http://www.tpc.org/New_Result/TPCH_Results.html.