SPEEDING UP MAIN MEMORY TABLE SCANS IN MYSQL

MICHAEL LODBERG SAMUEL

&

ANDERS UHL PEDERSEN

ABSTRACT

The Memory table structure in MySQL 4.1 is built upon a heap accessible through a tree structure. During table scans, this tree structure is repeatedly traversed by use of division and modulo operations that are inherently expensive.

When a given record has been located in the heap, it is copied to a memory location accessible by the query processor, a step that seems unnecessary for many operations as heap tables are already RAM-based.

In this paper, we examine the initial performance of table scans on Pentium 4 and Itanium 2, and suggest and implement improvement ideas.

We benchmark the changes and investigate the possible reasons for differences in the benchmarks – both CPU-architectural-, compiler- and MySQL-design issues.

Based on these investigations, we conclude that by removing the repeated copying of data in RAM the table scan running time can be reduced by a factor of up to 8. However, optimizing the utilization of the tree structure has marginal effect despite the fact that we managed to speed up the time it takes to locate a leaf node by more than 100%.

CONTENTS

1. INTRODUCTION

MySQL has engines supporting several types of tables, both disk- and memory-based. The disk-based are MyIsam (or variations thereof), BDB and InnoDB. The memory-based engine uses Memory tables[1], which were initially developed for internal use by the query processor and supported only hash-indexes, but later extended to be available for the database developers and enhanced to support B-Tree indexes as well.

As the Memory tables have no durability, speed is the most important feature for them. While keeping the tables in RAM automatically gives a significant increase in speed, we still identified a number of possible optimizations for the main memory component in MySQL 4.1 in [1].

In this new paper we keep focus on MySQL 4.1 and examine internal tree traversals as part of table scans, and compare the expected theoretical improvement with the actual benchmark results from the implemented code changes that include both reducing the number of required tree traversals as well as optimizing the traversal by using bit shifting rather than division and modulo operations.

It is a well established fact in optimization of code that numeric division (integer or floating-point) is among the most expensive operations that can be performed and if a situation exists where divisions can be easily eliminated it is almost always worth the effort to do so.

In MySQL 4.1, the method used for locating a leaf in a Memory table uses division and modulo operations to traverse the tree and does so repeatedly for every leaf that needs to be found. As mentioned in [1], the tree traversal must be expected to be especially costly in table scan since every leaf is fetched by tree traversal. Our changes address both the division and the repeated tree traversal.

We are aware that the two changes don't really complement each other, since we expect the elimination of excessive tree traversal to diminish the speedup gained by removing the expensive division operations, but the changes are relatively simple to make, and the overall result should still be that less clock cycles are wasted.

Additionally, we have identified that MySQL has a structure that requires it to copy data between the storage engines' buffer pools and the query processor. With a disk-based storage engine the cost of this is absorbed by the much more expensive copying from disk to RAM, but when the storage engine is RAM-based the approach seems wasteful.

As a result we have also modified this structure for SELECT-commands from Memory tables. This complements the other changes perfectly, as it removes a time-costly task, allowing the benefits of the others to be more clearly seen.

[1] Previously named HEAP tables

The rest of this paper is organized as follows. In the following section we describe the data structures and algorithms in the Memory storage engine. In section 3 we present the improvement ideas. The experimental framework is discussed in section 4. We present the experiments in section 5 and discuss the results in section 6. We conclude in section 7 and wrap up the paper with ideas for future work in section 8.

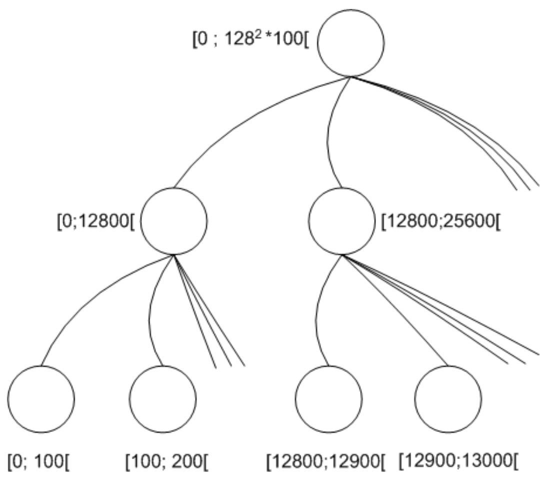
2. CURRENT HEAP IMPLEMENTATION

This section describes the data structures and algorithms that are affected by our improvement ideas, which we present in section 3. The design of the data structures in the heap reflects the original purpose of a Memory table, which was to act as a temporary table when the query processor needed to store intermediate results. As of MySQL 3.23, the Memory storage engine also has a public interface, making the heap available to the end user as an independent table type. The basic assumption underlying the heap data structures is that the field size, key size and consequently the record size are fixed.

2.1 THE MEMORY TABLE

A Memory table is stored in a tree structure with records at the leaf level. An inner node contains up to 128 child pointers. The maximum tree height is 4, thus the maximal number of leafs is 128^4. A leaf is simply a fixed size chunk of raw memory and it contains nothing but records. The location of a record inside a leaf is determined by an *offset*. Knowing that the first record in a leaf (*offset* = 0) is stored at address *leaf_start*, the record with some *offset*-value must be stored at address *leaf_start* + *offset* * record size.

A record is fetched by supplying its record position pos_{rec}. The assignment of a pos_{rec} to a record is based on the illusion that all records are stored in one large array, i.e. the pos_{rec} is simply an array index. Since the records are actually stored in a tree structure each pos_{rec} must be translated to the corresponding location in a leaf. If x records are stored in each leaf, the leftmost leaf contains the records with a pos_{rec} in the range [0 ; x [. The next leaf contains the records with a pos_{rec} in the range [x ; $2x$ [and so forth. With tree height h, the tree structure is thus capable of storing $128^h * x$ records.



Find record at pos_{rec} 12902

$$\left\lfloor \frac{12902}{12800} \right\rfloor = 1 \text{: go to 2nd child}$$

12902 mod 12800 = 102: the new pos_{rec}

$$\left\lfloor \frac{102}{100} \right\rfloor = 1 \text{: go to 2nd child}$$

102 mod 100 = 2: the 3rd record in leaf

fig. 2.1 - *The numbers indicate which record positions each node covers.*

The example tree in fig. 2.1 illustrates the assignment of record positions to the leaves based on the assumptions that the tree is full, the tree height is 2 and each leaf contains 100 records.

Finding the record with e.g. pos_{rec} 12902 in such a tree requires traversing the tree from root to leaf. Each child of the root has 12800 records in its sub-tree. The record with pos_{rec} 12902 must consequently be in the sub-tree of the second child of the root (12902 div 12800 = 1).[2] As a result of moving down one level, the pos_{rec} is now decreased to 102 (12902 mod 12800 = 102). The second child of the root has 128 children each containing 100 records. Consequently the record with pos_{rec} 102 must be in the second child (102 div 100 = 1). Furthermore it must be the 3rd record in this leaf since the *offset* is 2 (102 mod 100).

From this example it should be clear that traversing the tree from root to leaf consists of a series of modulo and division computations.

2.2 OPERATIONS ON THE MEMORY TABLE

On insertion of a record the next free pos_{rec} is identified. This might lead to allocation of a new leaf node in case no record positions are unused in any of the existing leaves. By doing a tree traversal the in-leaf location corresponding to the pos_{rec} is identified, and the record is subsequently inserted.

Scanning a Memory table using the tree structure above is quite trivial:
Fetch the first leaf, i.e. get the *leaf_start* address corresponding to the record with pos_{rec} 0. Read each record in the leaf. Fetch the next leaf, i.e. get the *leaf_start* address corresponding to the record with pos_{rec} *x* (*x* records pr. leaf) and so on (2x, 3x,…). Since the leaves are not joined together in a linked list, fetching the next leaf requires a tree traversal. When the in-leaf location of the next record to be fetched has been computed, the record is copied to the *current record*-buffer in the query processor. The copying takes place, because the record format in the storage engine may differ from the one used in the query processor.

2.3 HASH INDEX

As the hash index is indirect each bucket element consists of a pointer to the record it represents and a pointer to the next element in the bucket. The hash key for a record is translated to a bucket number (pos_{buck}) using modulo. The bucket numbers are equivalent to the record positions (pos_{rec}) mentioned above and this makes it possible to store the bucket elements in the very same tree data structure as is used for records. As a result of this strategy the number of potential buckets equals the number of records. Thus an ideal hash function would only need to store one element in each bucket (assuming unique record keys).

[2] Because of 0-indexing, 0 corresponds to first child, 1 corresponds to second child etc.

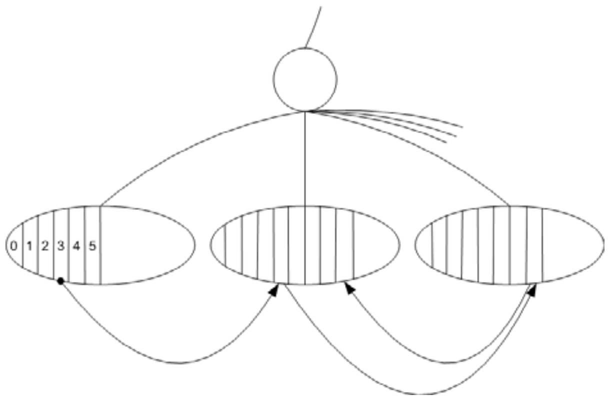


fig. 2.2 - *Four elements in bucket with pos_{buck} 3. Only the first element in the bucket is stored at pos_{buck} 3. The other elements are found by following the pointers.*

In fig. 2.2 a bucket with pos_{buck} 3 is illustrated in the lower left part of the tree structure. The hash key in each of the four elements all map to pos_{buck} 3 though only the first element in the bucket is actually stored at that position.

When inserting bucket elements the tree grows, as more leaves are allocated. Obviously the number of potential buckets (pos_{buck}) increases when more leaves are allocated. This will likely affect the mapping of a hash key to a bucket number because modulo is used. It naturally leads to situations in which the hash key for an already inserted bucket element now maps to a different bucket (pos_{buck}) and consequently the bucket element must be moved to another bucket. In [1] we have described how this issue is efficiently dealt with.

2.4 OPERATIONS ON THE HASH INDEX

All operations on the hash index (insert, delete, update, lookup) entail accessing one or more buckets and thus one or more tree traversals (to fetch the first element in a bucket). The hash index operations are described in detail in [1].

It follows from section 2 that tree traversal is central to operations on both the Memory table and the hash index, although the copying of each record to the *current record*-buffer is the most time-consuming action. Speeding up the tree traversal is one performance improving strategy. An alternative strategy could be to minimize the number of tree traversals. We have decided to pursue both strategies in the quest for better performance.

3.1 SPEEDING UP TREE TRAVERSAL

The tree traversal consists of a number of modulo and division computations. As the division operators (also used for modulo) are quite expensive in terms of clock cycles it might be beneficial to replace them with bit-shifting instructions. This is however only an option if the divisor is always a power of 2.

The number of elements in the inner nodes is always 128, i.e. a power of 2. Hence, if we can assure that the number of records in a leaf is also a power of 2 then the divisor is always a power of 2. The leaf node size in the current implementation depends on a system variable called read_buffer_size, which is 128KB by default. A leaf then contains the largest number of records that results in a leaf node size ≤ 128KB. Hence, the number of records in a leaf currently depends on the record size. We can change it by adjusting the number of records in a leaf to be the largest power of 2 that results in a leaf node size ≤ 128KB.

Now modulo can be trivially performed using an AND and a SUB instruction. Division by bit shifting is less trivial since the number of bits that the divisor consists of must be known in advance. Fortunately the number of bits (in the divisor at initiation of the tree traversal) can be computed in advance. When doing the actual tree traversal we know that moving down one level in the tree means that the divisor must itself be divided by 128, i.e. the number of bits in the divisor decreases by 7.

Counting the bits in the divisor can be done in several ways. The slowest and simplest method is the math-library version: $\log_2(\text{divisor})+1$. A faster but still simple method is to start with a guess of 1 bit and increase until the guess is right. It is even faster to do binary guessing, i.e. guess 16 bit, then 16+8 if divisor is larger or 16-8 if smaller, then (8 or 24) +/- 4 etc. until the guess is right[3]. A preliminary benchmark indicated that binary guessing is up to 7 times faster than using log.

The bit-shifting is only an improvement if the division/modulo computations are a bottleneck during the tree traversal. To determine if this is the case, the following four issues must be considered regarding the execution of the current implementation:

1. *Does the processor reuse the result of the division computation in the subsequent modulo computation?* Since the operands are the

[3] Assuming the divisor never exceeds 32 bits

same for the division and modulo computations, only a single integer division operation is needed to compute both results. A modern processor might be able to reuse the result from the division computation and thus eliminate the otherwise needed integer division operation used to perform modulo.

2. *What is the processor-latency and the processor-throughput for the integer division operation?* The latency for an operation is the number of clock cycles that are required to complete the execution of the operation. This determines the lower bound on the time it takes to traverse the tree since the current tree traversal relies on a series of mutually dependent integer division operations. The throughput for an instruction is the number of clock cycles required to wait before the same instruction can be processed again. If no reuse occurs, two independent division operations are executed at each level of the tree traversal and in such case the division-throughput affects the running time.
3. *How expensive is a data cache miss*[4] *and how many cache misses can be put down to visiting inner tree nodes?* The division and modulo computations determine which child-pointers to follow when traversing the tree. If the child-pointer of interest is not in any data cache, a cache miss occurs and the pointer must be fetched from memory. In such case the cache miss becomes the bottleneck and the running time is thus less affected by the division improvement.
4. *Is the processor capable of out-of-order or parallel execution to minimize the delay (if any) caused by a cache miss?* While waiting for the completion of a load instruction causing a data cache miss, the processor might be able to execute load-independent instructions which will reduce the perceived latency of the division operation.

These issues constitute one part of our investigation below; another part is considering how to minimize the number of tree traversals.

3.2 MINIMIZING THE NUMBER OF TREE TRAVERSALS

When scanning a Memory table the tree is traversed once pr. leaf. The improvement to make is quite obvious: Join the leaves into a singly linked list by adding next-leaf pointers. This way, a tree traversal is only needed once - to fetch the first (leftmost) leaf. The drawback is the space overhead due to the allocation of a next-leaf pointer pr. leaf, i.e. 4B pr. 128KB (assuming the default leaf size and 32-bit addressing). This improvement makes the bit-shifting improvement insignificant for table scans but all other operations still benefit from the faster tree traversal as table scanning is the only operation visiting the leaf nodes in sequence.

The benefit of linking the leaves depends on three things:

1. *Does the lookup of the pointer to the next leaf generate a cache miss?* As mentioned, the record is copied to the *current record-*

[4] When we use the term *cache miss* it normally refers to a data cache access miss and it normally implies a memory access. Accordingly, we typically do not explicitly specify the cache level when using the term *cache hit*.

buffer in the query processor when the in-leaf location of the next record to be fetched has been computed. Consequently the *current record* is always present in the cache. At some point the last record in a leaf becomes the *current record*. Knowing this, it is wise to allocate and store the pointer to the next leaf just after the last record in the leaf. Unless the record precisely fits in (a multiple of) the largest cache line, the next-leaf pointer will be readily available in cache, because it is “accidentally” loaded into the cache when the last record in the leaf is copied to the *current record*-buffer. Using this strategy it is less likely that the pointer lookup will generate a cache miss.

2. *Does the tree traversal generate cache misses?* If the inner nodes visited during a tree traversal are in cache then the lookup time is purely determined by the speed of the division/modulo computations. In such case following a next-leaf pointer might not be much faster than a tree traversal.
3. *Is the processor capable of out-of-order or parallel execution, thereby minimizing the delay caused by a possible cache miss when looking up a next-leaf pointer?* If the processor can be kept busy during a possible cache miss due to looking up the next-leaf pointer, the next-leaf-pointer-lookup might outperform the tree traversal in any case.

With our experiments, we will shed light on these points.

3.3 REDUCING THE COST OF READING RECORDS

When a record is to be read, it is located by the storage engine and returned to the query processor. In MySQL this is done by the query processor providing the storage engine with a pointer to allocated area in RAM. The storage engine is then expected to copy its data, one record at a time, to the given area.

The rationale for this seems to be that the time it takes to retrieve the data from disks is far greater than the time it takes to copy it to a standard location, and the query processor will then have easy, standardized access to the data that will already have the structure that the query processor needs when it is copied.

This seems to be a fair solution for disk-based storage engines. In theory, it might be cleaner to let the record propagate through the layers as a variable or a return value, but it is important to remember that even a single record will in many cases be rather large, and there is a risk that the stack (and other control structures) will be put under a greater pressure than need be if the ‘clean’ solution is selected.

However, for the Memory-storage engine, this solution seems inefficient. The data is already present in RAM and allowing the query processor to access data that is already properly structured rather than copying doesn’t seem to be an unacceptable violation of encapsulation rules, especially not considering the current implementation.

Unfortunately, a large number of functions in the query processor rely on the record being present at the allocated area. The code is rash with memcpy-functions, and many methods that accesses the fields in records don't use the record pointer to calculate offsets from, but use the pointer to the allocated RAM area in the query processor to calculate absolute addresses.

The main concern with this is the fact that pointers present in the local scope of the query processor does in fact point to the record address – but changing those pointers does not affect the field-pointers – which suggests that the assumption that a pre-allocated area exists is exploited at several locations in the code. The benefit is obviously that the pointers need only be calculated once; speeding up record accesses, but the actual calculation is spread out to several functions, making it very difficult to avoid the copying.

Changing all occurrences of this is far beyond a paper as this and not particularly interesting since the partial implementation we have made exposes both strengths and weaknesses of the solution.

In practice we have decided to implement the solution such that it works for SQL-statements that select data from a single table with or without where-clauses. This is done by adding a global record pointer to the system. The pointer is updated by the heap engine instead of copying a record to the query processor. In the query processor, the address of the record pointer is used to iterate through the fields and update their individual pointers accordingly, as they would still be pointing to the original area. When this is done, the query processor can continue. Note that we only need to update a field pointer if it is part of the query. The optimal solution would be to update the field pointers in a just-in-time manner: Consider a query that contains two conditions and both must evaluate to true. If the first condition evaluates to false we do not need to examine the second and hence we do not need to update the field pointers in the second condition. Accordingly we do not need to update the field pointers in the SELECT clause unless the row matches the query conditions. Since we are only interested in the best-case (1 column) and worst-case (all columns) scenario we have not implemented this just-in-time solution.

While there would be no great computation cost by letting this record pointer propagate 'properly' through the layers, doing so would break many overloaded functions and methods which would need to be fixed for a build to work, but which would add absolutely no value to the experiments or conclusions.

4. EXPERIMENTAL FRAMEWORK

4.1 BENCHMARK STRATEGIES

The changes in performance due to the proposed tree-optimizations are expected to be small in terms of wall-clock time. In a preliminary benchmark of the current implementation we selected 100.000 records corresponding to a heap containing app. 130MB data. Scanning the entire Memory table resulted in a running time of astounding 0,15 s. on a Pentium 4 1,7GHz (excluding query-processing time and output-time).

To accurately measure the change in performance we therefore need to process large amounts of data. Since a Memory table is purely RAM-resident the amount of RAM in the benchmark computers dictates a relatively low upper limit on the data size. Faced with this problem we have identified two approaches to the performance measurement that we consider to be acceptable:

1. Implement the changes in MySQL and alter the heap to repeatedly call the modified functions until the difference in running time can be measured with sufficient reliability.
2. Simulate the heap-functions outside the MySQL-context.

The first option ensures that the changed functions interact with a real database environment and that the measured times are as authentic as possible. The second option on the other hand, makes it easier to rule out any influences from the MySQL-environment and to accurately control parameters such as the number of cache misses, the tree height etc. and hence determine the direct effect of the changes on the used architectures.

While some of the questions in sections 3.1 and 3.2 can be answered with sufficient certainty in a simulation, other questions can be more suitably answered in the MySQL-environment. Consequently we have decided to pursue both strategies.

4.2 HARDWARE PLATFORM

We have decided to benchmark the optimizations on the two very different platforms described below.

	Pentium 4 1600MHz	Itanium 2 900MHz
System:	Microsoft Windows XP	Debian GNU/Linux 2.4.25
Addressing:	32-bit	64-bit
Processors:	1	2
RAM:	256MB	4GB
L1 cache size/line:	12KB / 128B	16KB / 64B
L2 cache size/line:	256KB / 128B	256KB / 128B
L3 cache size/line:	None	1,5MB / 128B
Compiler:	MS Visual C++ 6.0	Intel C/C++ 8.0.066

The P4-platform is interesting because the *CPU* is “intelligent” in the sense that it diminishes pipeline stalls by out-of-order instruction execution. The Itanium 2-platform has been chosen because the CPU is capable of parallel instruction execution and because the *compiler* is “intelligent” as it diminishes pipeline stalls by bundling instructions that can be executed in parallel. In the remainder of this section we elaborate on these features and other relevant differences between the platforms.

That a shift operation is many times faster to perform than a division is basic knowledge. [2] However, CPU designers have put a lot of effort into both speeding up division and generally diminishing the effect of computation latency.

In P4, the optimizations performed can be explicit by the compiler (i.e. by using the assembler instruction XOR rather than CWD or CDQ for sign extension in certain cases [3]), but in many cases, the processor itself has ways of improving real throughput of division operations - most notably through the “Out-of-order Execution Logic” (OEL).

The OEL specifies that if several operations are independent of each other, the processor will attempt to execute these in parallel, delaying the results from them to make it seem as if they have been executed in order. This means that in optimal cases[5], only the actual use of the Arithmetic Logic Unit (ALU) will delay execution of a division operation, and this should reduce the overall latency for this operation. The ALU even has a pseudo-pipeline, as it can do a so-called “staggered add” where the first 16 bits of an addition operation is passed on to dependent units in the processor before the next 16 bits are computed.

In the Itanium processor, the approach is somewhat different.

First of all, the Itanium 2 (I2) processor/ALU has no native division operation [4], so any division operation has to be done by algorithms implemented in software.

Second, the ALU is fully pipelined and can start a new operation every clock cycle, which greatly improves the overall throughput, but does little to help the speed in applications that doesn’t use the ALU intensively, since the latency for each operation remains constant.

Finally, through use of Itanium’s “Explicitly Parallel Instruction Computing” (EPIC) it is possible for the compiler to bundle certain operations so that the processor will be able to execute these in parallel. It is important to note that only certain operations can be bundled together, and that this is done entirely by the compiler – the processor does not rearrange at runtime like the P4, and the dependencies that decide what can run in parallel and what can’t are distinctly different from those on P4.

[5] And with a warm pipeline

4.3 MEASUREMENTS

Just as Intel's own documentation has reservations regarding the perceived cost (duration) of the various instructions, our measurements have issues to be aware of.

Most notably, the OEL makes it virtually impossible to determine the effect of our changes in a real environment, with a full-scale database server that serves many users.

Therefore, it is important to distinguish between our improvements and those done automatically by compiler and CPU. The compiler can largely be controlled by the use of flags, but the CPU improvements are illustrated by running tests that only differ in the dependency of the variables – effectively a throttle to control the level of parallelism possible.

This means that we can test the theoretical best-case improvement of our changes directly, and show that the changes does ease the load on the ALU, which improves the general throughput even if the perceived effects in a live environment might be somewhat diminished on a Pentium 4 due to OEL and other factors.

We find our theoretical best case by creating performance cases that focus on the simple, raw achievements, but in a way that mimics the existing code.

In many hardware based implementations of division, the ALU will have the remainder of an integer division present. Even in advanced division algorithms and hardware, the ALU will have an (unshifted) remainder present in all iterations. However, since the I2 relies on a software algorithm to do its division, it also relies on the software to provide the remainder; the processor itself cannot be expected to have any knowledge of the usability of values in the registers used by the algorithm.

4.4 MEASURING ACHIEVEMENTS

In order to determine the effects of our changes, we need to find a reliable set of measurements. According to Douglas W. Jones [5] it is important to ensure that the computer used for benchmarking is exclusively used for the benchmarks. However, due to the complexity of MySQL as well as of modern OS it is difficult to control the CPU-time allocated for benchmarks and obtain reliable results.

Also, this statement was made in 1986, where the way computers were used was distinctly different from today. Jones' requirement remains interesting in order to determine the raw effect of changes made, but this won't necessarily reflect the effect in a live system. Unfortunately, it is close to impossible[6] to reliably benchmark the effects in a live system with hundreds of tables, users, etc., but it remains important to remember and compare

[6] And certainly beyond the scope of this paper

the effects on the live system (or an approximated version of it) with the isolated tests.

One solution to this is to simply run the benchmarks so many times that one is confident that any difference in the average run time before and after the changes displays the actual improvement (or deterioration) incurred by the changes. The obvious advantage of this method is that it benchmarks the changes in a simulation of real usage – and hence gives an indication of what users can expect if the changes were implemented.

The best metrics for this benchmark is wall-clock time, time spent in the changed functions, number of times given functions are called, etc.

As discussed in 4.1, another solution is to isolate and – if possible – extract the code that is being changed to a small test-program that doesn't do anything else than run the code with and without the changes. This has to be done carefully in order to be reliable. First, if more than one change has been made, the changes should be benchmarked individually as well as combined in order to see what effect (if any) they have on each other. Second, compared to a full-fletched MySQL, a test-program with only one running thread can have a false positive effect on the instruction cache, might have effects on data cache, and might be easier for compiler and CPU to optimize due to the reduced complexity. But not all side-effects increase speed: On the P4, the OEL is so good at diminishing the effects of division latency that a small loop won't have sufficient instructions to keep the CPU-pipeline full while waiting for a division to complete. See sections 5.5 and 6.5 for a further discussion on how we examine the effect of this.

Wall-clock time and time spent in the changed functions are also interesting metrics for this benchmark, but should be complemented by CPU-statistics, cache-statistics, etc. Especially cache-statistics should be interesting, since one of the changes we've made changes the access pattern when scanning Memory tables.

These statistics are very difficult to obtain hard measurements for, since both system-wide and application-wide statistics will tend to include too much noise. In the following sections we will describe how the different measurements are designed to also uncover the changes in cache utilization.

4.5 CHOICE OF TIMERS

In MySQL we intend to use the built-in timer functionality, which is based on the system timer with a precision of 10-55 ms. In the simulations on the Pentium 4-platform we can use the high-resolution Windows-timer QueryPerformanceCounter with a resolution in the order of nanoseconds. In the simulations on the Itanium 2-platform we intend to use the gprof-tool with a 10ms-resolution.

While not altogether optimal, these timers suffice because we can ensure sufficiently long running times to eliminate any influence from a timer with ms-resolution.

As mentioned in section 4.1 we have decided to both benchmark the changes in MySQL and simulate the changes in order to shed light on all relevant aspects.

The performance improvement due to the two changes mentioned in sections 3.1 and 3.2 depends on a number of factors. The experiments must be designed in a way that allows us to control or at least predict the effect of these factors. It is furthermore essential that the experiments make it possible to draw conclusions regarding the extent of the speedup in a realistic setting.

The factors affecting performance can be extracted from the questions in sections 3.1 and 3.2:

1. Reuse of the division remainder in the modulo computation
2. Division operator speed vs. bit-shifting speed
3. Effect of out-of-order/parallel execution
4. Cost of cache misses

Before discussing how to measure the effect of each factor it is useful to investigate the instruction dependencies in the tree traversal.

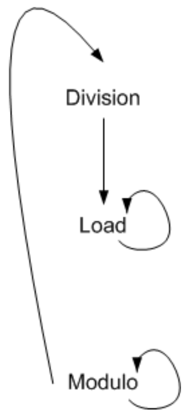


fig. 5.1 - *Dependency graph for tree traversal*

Each iteration in the tree traversal loop consists of a division operation, a load operation and a modulo operation. It follows from fig. 5.1 that the load depends both on the result of the division and on the result of the load in the previous iteration. After moving down one level in the tree structure a new division computation is needed and it depends on the result of the modulo computation in the preceding iteration. Furthermore the new modulo depends on the modulo computation of the previous iteration.

5.1 REUSE OF DIVISION COMPUTATION IN MODULO COMPUTATION

In a simulation of the tree traversal we can alter the algorithm such that the divisor or dividend in the division computation differs from the divisor or dividend in the modulo computation. If the running time changes as a result of this modification we know that the processor reuses the value in the remainder register when computing modulo.

If the processor is capable of reusing the remainder value from division in the modulo computation, the dependency graph must be modified as illustrated in fig. 5.2.

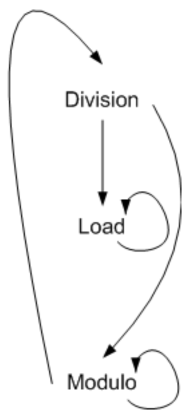


fig. 5.2 - *Modified dependency graph for tree traversal*

5.2 DIVISION OPERATOR SPEED VS. BIT-SHIFTING SPEED

According to the Intel P4-processor documentation [6] the integer division instruction IDIV has a latency of 56-70 cycles and a throughput of 23 cycles. The Intel C++ Compiler on the other hand converts the division operator to a series of smaller floating-point instructions, which leads to a latency of 43 cycles and a throughput of 5.5 cycles. In any case, if no cache misses occur during the tree traversal, the division operation is the bottleneck in the tree traversal loop because the other tree traversal instructions depend on the result of the division.

In a simulation of the tree traversal we can exclude all cache misses and compare the running time of the version using a division operator with the version using bit-shifting. This shows the maximal speedup due to bit-shifting.

5.3 OUT-OF-ORDER/PARALLEL EXECUTION

If an instruction causes a data cache miss two scenarios are possible. Either it causes the processor to stall and not resume until the data is available in cache or out-of-order/parallel execution is utilized in which case the processor executes other instructions that are independent of the result of the instruction causing a cache miss. As the illustrations above show, the subsequent modulo and division computations are independent of the result of the load and can be executed while the load instruction waits (assuming a cache miss). Speeding up the division computation in this scenario is less beneficial since even the slow division operation might be able to finish during a cache miss. Thus, bit-shifting is only worth the effort when the load results in a hit.

We can test the out-of-order/parallel execution capabilities of the processor by adjusting the number of cache misses in the simulation. If the difference in running time between the division-operator version and the bit-shifting version is unaffected by the number of cache misses, then no out-of-order/parallel execution has occurred. If the difference shrinks when adding more cache misses then it has likely occurred.

5.4 CACHE MISSES

We have already discussed how to determine the effect of cache misses on tree traversal. The optimization consisting of following a pointer to the next leaf instead of doing a tree traversal is expected to be beneficial because the allocation of the next-leaf pointer just after the end of the leaf ensures that it is present in cache when needed. Thus, we do not expect cache misses to affect the next-leaf pointer optimization. Benchmarking table scan in MySQL will assist in clarifying this issue.

5.5 LOCATING THE NEXT LEAF IN MYSQL

Table scan, record insertion/update/deletion and hash index operations are all affected by tree traversal. Hence we want to measure how our changes affect the time it takes to locate the next leaf in MySQL.

The query processor requests one record at a time from the Memory storage engine. If the record matches the condition(s) of the query it is further processed, otherwise it is discarded. Since we only focus on locating leaves, we want to disable all further row processing in the query processor. Consequently the query processor will discard all rows in the benchmarks and no rows will be sent to the client. Furthermore we intend to exclude the pre-processing of the query (parsing, execution planning etc.) from the measured running time. When the Memory storage engine has identified the location of the next record it copies the record to the *current record*-buffer in the query processor. We exclude this operation from the benchmarks as well. Hence the benchmark will only measure the time it takes to locate each leaf in the Memory table, i.e. the part of the heap affected by our changes.

Normally several records are stored in each leaf in the tree structure. To maximize the number of tree traversals we have modified the heap such that only one record is stored in each leaf and with a record size of 8 bytes. This way the amount of memory in the benchmark computers does not severely limit the size of the tree structure and the (performance-affecting) tree height. We intend to repeatedly fetch all the leaves until the total response time reaches a level that makes it possible to rule out timer resolution as the cause of the difference in response time.

It should be noted that the row processing of matching rows in the query processor affects the instruction cache, data cache, TLB etc. Disabling row processing might artificially speed up the heap response time in general and this might be in favour of the original version and in particular the bit-shifted heap because it is expected to benefit the most from improved data cache performance.

Eliminating the copying of a record to the *current record*-buffer might also affect the benchmark validity. When copying, the record also becomes available in cache and some of the inner nodes in the tree structure might consequently be "swapped out" of several or all cache levels. Once again our benchmark strategy improves cache performance and thus favours the original version and in particular the bit-shifted version. In addition, running the benchmarks in a single-user environment further improves the cache utilization.

On the other hand our benchmark strategy makes it possible to rule out influence from the MySQL-environment, which we consider important as we expect the differences to be small in terms of wall-clock time.

5.6 TABLE SCAN IN MYSQL (TREE TRAVERSAL OPTIMIZED)

Table scan is the only operation in MySQL that relies heavily on the tree traversal algorithm. Consequently we want to benchmark how our changes affect the total running time of table scan, without costs incurred by external factors such as I/O and communication between the server and client.

In contrast to the leaf-locating benchmark this benchmark displays more realistic cache behaviour because all parts of the table scan execution is included. As we expect the total running time to be affected by the record size we want to measure the speedup due to our changes while varying the record size. To make the figures comparable we need to keep the number of leaves constant, i.e. we have to adjust the number of records when adjusting the record size. We furthermore ensure that the number of records pr. leaf is a power-of-2 in all three versions because we want to exclude differences in records pr. leaf as the cause of differences in running time. Once again, we intend to repeat the table scan until "random noise" can be ruled out as the cause of the difference in response time.

5.7 SIMULATION VALIDITY

Before running any of the described tree traversal simulations we need to verify that the running times in the simulations are sufficiently close to the running times in MySQL. Specifically, we need to verify that the *difference* in running time between division-operator-tree-traversal and bit-shifted-tree-traversal in the simulation is comparable to the difference in running time in MySQL.

The simulation of the tree traversal differs from the MySQL-version only in the data access pattern. While the MySQL-version reads data from *a node* at the calculated position, the simulation reads data from *an array* instead. When reaching the end of the array the simulation moves to the start of the array and rereads the elements. Only every 32nd array element is read as this ensures the elements are not already available in cache[7]. Furthermore the array is sufficiently large to ensure that the first array elements are swapped out of cache before the simulation moves to the start of the array

[7] Otherwise it could have been part of the cache line when reading the previous element

again. This way we do not need to allocate large amounts of data in RAM and we can specify precisely how many cache misses should occur.

In our MySQL-benchmarks we expect each inner node to generate a few cache misses until the entire node is available in cache and we expect that when an inner node is present in cache it will stay there while needed because it is unlikely to be "swapped out" as mentioned in section 5.5. Given these assumptions we are able to accurately calculate the expected number of cache misses during a full table scan and we can distribute these misses such that at most one miss occurs pr. tree traversal in our simulation.

5.8 TAKING TIMER-RESOLUTION INTO ACCOUNT

The duration of a single tree traversal is so brief that we have no choice but to measure the time it takes to perform a large number of tree traversals. By repeatedly calling the tree traversal and scan functions we are able to obtain measurements in the order of seconds. This essentially eliminates the need for high precision timers.

5.9 TABLE SCAN IN MYSQL (MEMORY COPYING OPTIMIZED)

In the original version the current record is copied from storage engine to a *fixed* location in the query processor. Consequently the query processor does not have to rely on offset computations when accessing the fields in the current record. Instead field pointers containing an exact memory address are used in all parts of the query processor. Without the copying of the current record to a fixed location we have to compensate by updating the field pointers in the query processor such that they point to fields in the current record located inside the storage engine. Note that we only update a field pointer if it is part of the query.

We want to examine the following factors that determine the benefit of having removed the copying of records:

1. The fraction of the total number fields that is part of the query
2. The size of the fields that are part of the query

In these benchmarks we once again intend to exclude costs incurred by external factors such as I/O and communication between the server and client.

Unless otherwise indicated we have run the tests with 300000 leaves. Each table scan was repeated 100 times and hence we have divided all running times by 100. We have compiled MySQL with the same options and flags as is used to compile the official MySQL-binaries [7] unless otherwise indicated. The simulations have been compiled with compiler-default release flags on both platforms. The only exception is that the simulations on Itanium were compiled with the –O1 option (compiler-default is –O2) because this is the optimization level used when compiling the official MySQL-binaries on Itanium. On P4 we compiled with the MySQL default -O2 option.

6.1 VALIDITY OF SIMULATIONS

We initiate this section with an evaluation of the validity of the simulations since it form the basis of most of the subsequent benchmarks. We have measured the difference in speed between the original MySQL-implementation and the bit-shifted MySQL-version (DIFF-M) with the number of leaves as the independent variable. These results are compared to the difference in speed between the *simulation of* the original MySQL-implementation and the *simulation of* the bit-shifting optimized MySQL-version (DIFF-S). We tested the validity with both processors: Pentium 4 (P4) and Itanium 2 (I2).

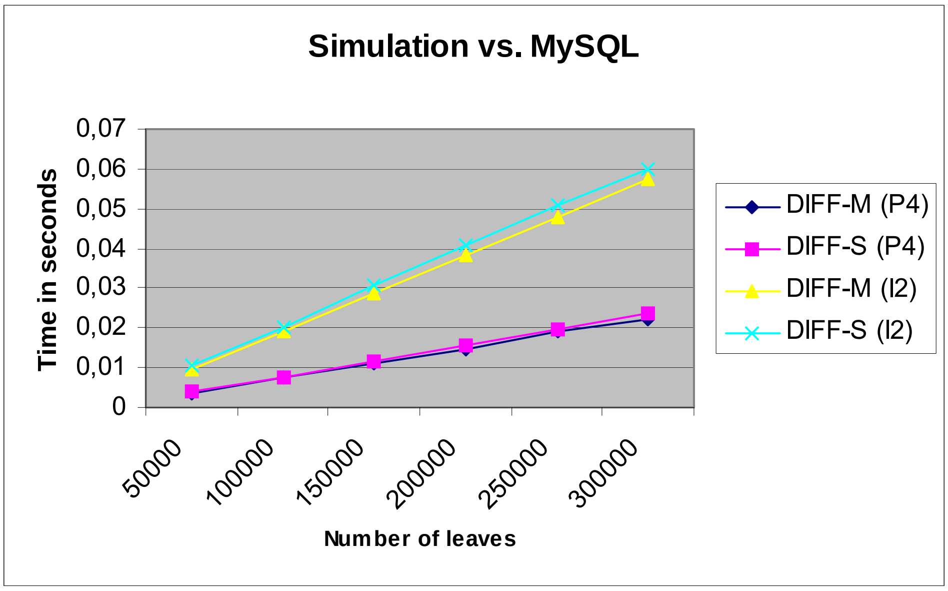


fig. 6.1 - *Comparing simulations to MySQL*

The simulations appear to be a good approximation at least when the number of leaves is in the range [50000; 300000]. The number of leaves is within this range in all the subsequent benchmarks and thus the tree height is 3 in all benchmarks. The difference in the simulation is 6% larger than the difference in MySQL on both platforms. By proper use of constants we have been able to take this into account in the measurements below.

6.2 REUSE OF DIVISION COMPUTATION IN MODULO COMPUTATION

In this simulation benchmark of the original MySQL-implementation (ORI-S) we ensured that the data access instruction always requested the same data and hence we practically excluded all cache misses without excluding any instructions from the simulations.

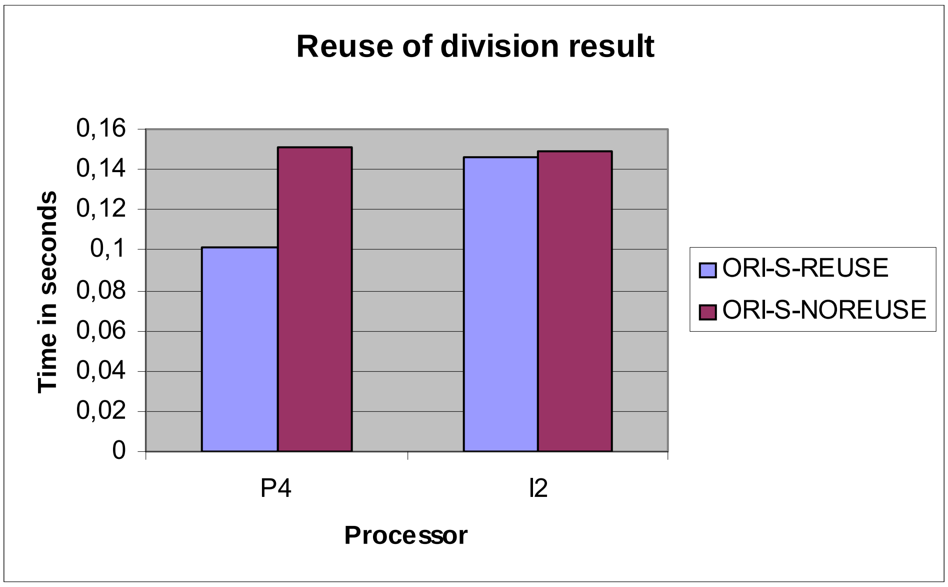


fig. 6.2 - *Reusing the division remainder to compute modulo*

The result is somewhat surprising. The P4 processor reuses the result of the division computation in the subsequent modulo computation and it is clearly beneficial. On the other hand, the Itanium 2-processor does not benefit from reusing.

To determine the reason for this, let us first look at the no-reuse case in the P4 processor. Since no cache misses occur, the division computation is the bottleneck. The integer division instruction IDIV is used for both division and modulo and it has a *throughput* of 23 cycles in the P4 processor [6], i.e. the processor has to wait for 23 cycles before it is able to process another, *independent*, IDIV instruction. The processor cannot be kept busy for 23 cycles between the division computation and modulo computation. Consequently it stalls for some time before initiating the modulo computation. When looping (next iteration), it has to execute division again. Now the division computation *depends* on the result of the modulo computation as illustrated in fig. 5.2. Thus the processor stalls again due to the IDIV *latency* of 56-70 cycles.

Now let us look at the reuse case in P4. The modulo computation has to wait for the result of IDIV which takes up to 56-70 cycles but then the modulo computation only takes a few cycles and after looping the next IDIV can start straight away. Consequently it is beneficial for the P4 processor when the divisor and dividend are the same.

After inspection of the assembly code we have verified that the Itanium 2-processor does reuse when possible. Thus in the reuse case, the processor waits for the division computation to finish and reuses the result to immediately compute the result of modulo.

In case of no-reuse the processor can execute the division and modulo computations partly in parallel because the processor only has to wait for 5,5 cycles from initiation of the first division operation to the initiation of the second division operation (that calculates modulo). Hence, the Itanium 2-processor does not really benefit from reuse because of its near-optimal division throughput.

6.3 DIVISION OPERATOR SPEED VS. BIT-SHIFTING SPEED

Once again the data access instruction always requested the same data in this benchmark. Hence the difference in speed between the simulation of the original MySQL-implementation (ORI-S) and the bit-shifting implementation (BIT-S) indicates the *maximal* possible speedup due to our change. In a MySQL-context this corresponds to the unlikely situation in which all inner nodes are available in cache at all times.

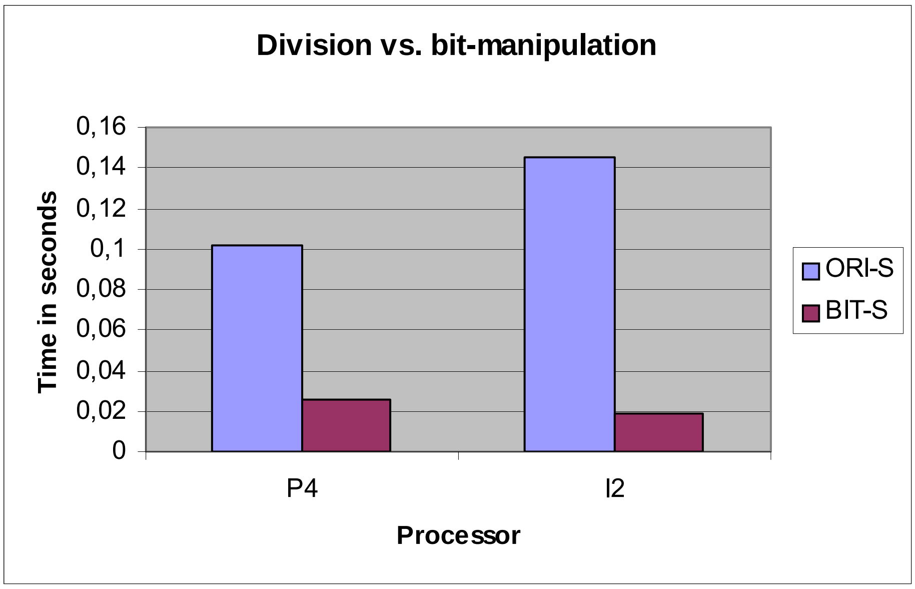


fig. 6.3 - *Maximal speedup due to bit-shifting*

While the maximal speedup on the platform using the P4 processor is app. 440%, it is no less than app. 660% on the platform using the Itanium 2 processor which indicates that the division operation is quite slow in the Itanium 2 processor.

6.4 OUT-OF-ORDER/PARALLEL EXECUTION

We now turn to the opposite case in which every data access generates a cache miss. In this situation the difference in speed between the simulation of the original MySQL-implementation (ORI-S) and the bit-shifting implementation (BIT-S) indicates the *minimal* possible speedup due to our change. In a MySQL-context it corresponds to the just as unlikely situation in which a cache miss happens every single time an inner node is visited.

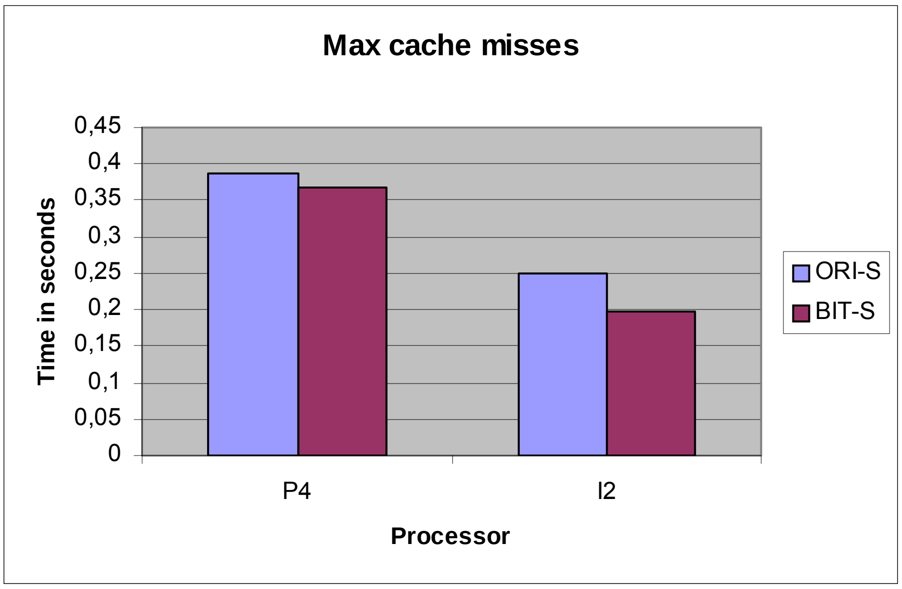


fig. 6.4 - *Minimal speedup due to bit-shifting*

With the data cache access as the constant bottleneck only a small speedup can be expected on a computer using the P4 processor because the division instruction almost manages to finish during a cache miss. This reveals that cache misses are quite expensive and that the P4 processor is obviously capable of utilizing the cache miss time through out-of-order execution by, for instance, performing a division operation.

On the Itanium, the cache miss reduces the benefit from bit-shifting, but a speedup is still expected because both division and modulo cannot finish during the cache miss. The benchmark does however indicate that the Itanium 2 processor is capable of parallel execution.

6.5 LOCATING A LEAF IN MYSQL

On the basis of the previous benchmarks we are at last prepared for the interpretation of the results from the MySQL-benchmarks. The benchmarks have been run on both platforms with the number of leaves as the independent variable. The query processor and the Memory storage engine were modified as described in section 5.5 and the benchmarks only contain table scan measurements.

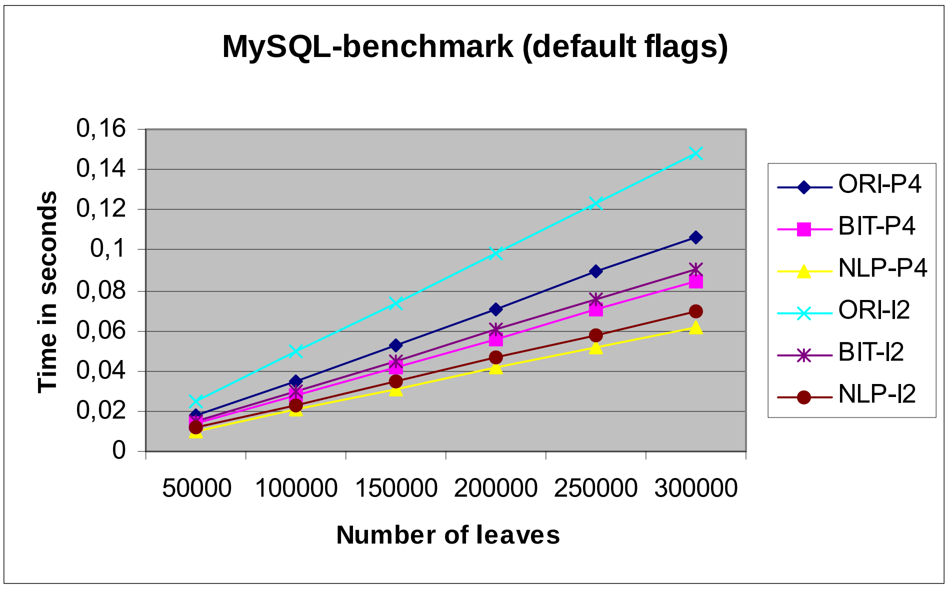


fig. 6.5 - *Comparison in MySQL on both Pentium 4 and Itanium 2*

	Pentium 4	Itanium 2
Speedup due to bit-shifting	26%	64%
Speedup due to next-leaf-pointers	72%	113%

table 6.1 - *Speedup of locating a leaf in MySQL due to changes*

On both platforms the MySQL-version using next-leaf pointers (NLP) is the winner. This indicates that looking up a pointer to the next leaf does not generate a cache miss.

The cache utilization in this benchmark might be slightly better than in a real MySQL-setting as mentioned in section 5.5. This might have sped up the two tree traversal versions (the bit-shifting version in particular) because the risk of an inner node being “swapped out” of cache while still needed is minimal. Consequently the relative NLP-speedup might be even larger in a real setting.

Section 6.2 revealed that the Pentium 4 benefits from reusing the division result in the modulo computation. Thus the bit-shifting operations have to compete with a single division instruction and this somewhat limits the potential speedup due to bit-shifting as we also showed in section 6.3.

Although the official MySQL-binaries are compiled with the –O1 optimization level on Itanium we decided to rerun the MySQL-benchmark after compiling with the compiler-default –O2 level. Only the bit-shifted tree traversal benefits from this.

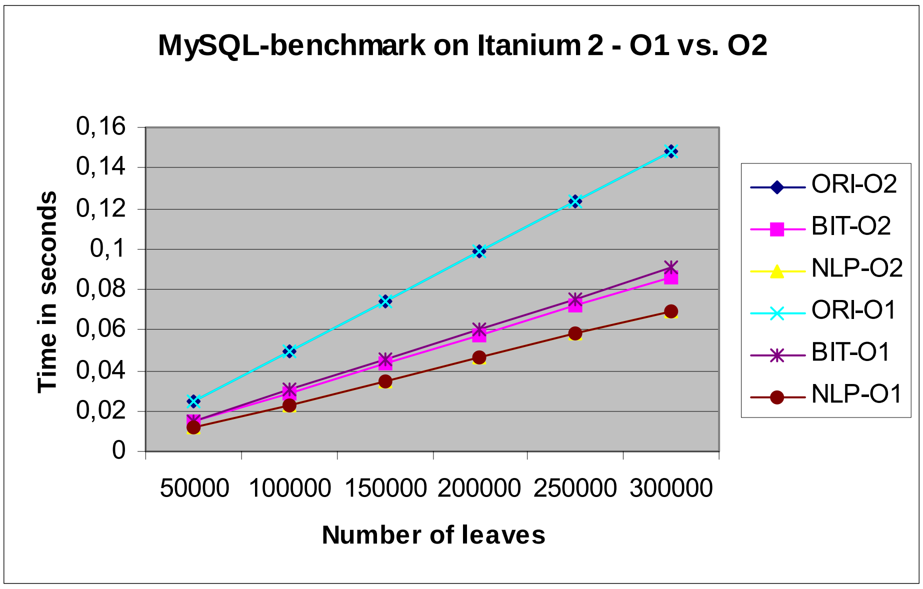


fig. 6.6 - *Comparison in MySQL on Itanium 2. Compiled with –O2 vs. –O1*

Itanium 2	O1-flag	O2-flag
Speedup due to bit-shifting	64%	71%
Speedup due to next-leaf-pointers	113%	113%

table 6.2 - *Compile-flag dependent speedup in MySQL on Itanium 2*

The speedup due to bit-shifting has increased from 64% to 71% on Itanium 2 while the speed of the original tree traversal and following a next-leaf pointer is unchanged.

The software pipelining optimization comes into play when moving from O1 to O2 [8]. This optimization explains why the speedup due to bit-shifting increases from 64% to 71%. By software pipelining we refer to the concept of overlapping the loop iterations, i.e. to execute instructions from two or more iterations concurrently as illustrated in fig. 6.7.

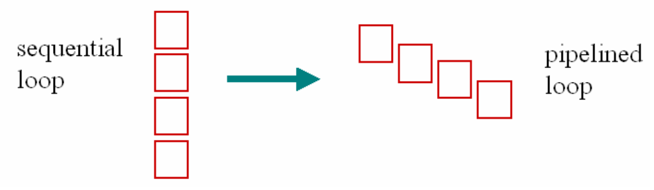


fig. 6.7 - *Software pipelining*[8]

[8] As seen at `http://www.cs.nyu.edu/courses/fall04/G22.2243-001/lectures/lect7-1up.pdf`

In the original MySQL-version the division operation must be executed prior to the load and modulo operations as both depend on the division result (as verified in section 6.2). The division in the next loop iteration has to wait for the completion of the modulo operation in the current iteration as was illustrated in fig. 5.2. Thus software pipelining cannot occur.

In the bit-shifting version software pipelining is possible. The bit-shifting operations used to perform modulo do not depend on the division-by-bit-shifting operation as illustrated in fig. 5.1. Furthermore the Itanium 2-processor is capable of executing the bit-shifting operations in parallel. As illustrated in table 6.3 the division and modulo operations in iteration x+1 can execute concurrently with the load operation in iteration x. The table clearly simplifies the execution but it illustrates the concept.

Cycle number	Instructions executed in parallel		
0	Division_1	Modulo_1	
1	Load_1	Division_2	Modulo_2
2	Load_2	Division_3	Modulo_3
…	…	…	…
X	Load_x		

table 6.3 - *Software pipelining the bit-shifted loop*

As none of the versions benefit from compiling with optimization level O3 we have not included the results from that benchmark.

6.6 TABLE SCAN IN MYSQL (TREE TRAVERSAL OPTIMIZED)

Having estimated how the changes affect the time it takes to fetch the next leaf, the next and final step is to investigate the impact of our changes on the total running time. The running times in this benchmark correspond to table scans in which no records satisfy the selection criterion. The number of leaves (3125) is held constant because we double the number of records when the record size is halved. Hence the amount of data is constant as well (app. 400MB). The table scan has been repeated 1000 times to ensure sufficiently large differences in running time.

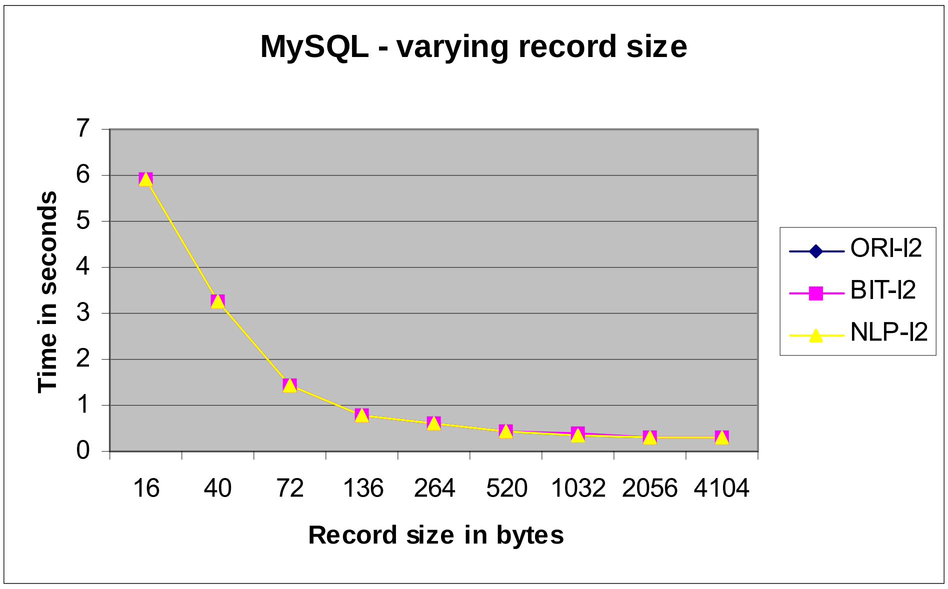


fig. 6.8 – *Total running time in MySQL for different record sizes*

The graph in fig. 6.8 indirectly indicates that the running time is nearly doubled when the number of records is doubled – despite the fact that the record size is divided by two. This is primarily due to the record-size independent iteration overhead (return to query processor; examine record; request next record). The running times are virtually identical for all three versions of MySQL.

By "zooming in" on the running times in fig. 6.8 we can examine the differences between the three versions of MySQL. We add a constant to each running time such that the running time of the original implementation is always 1.

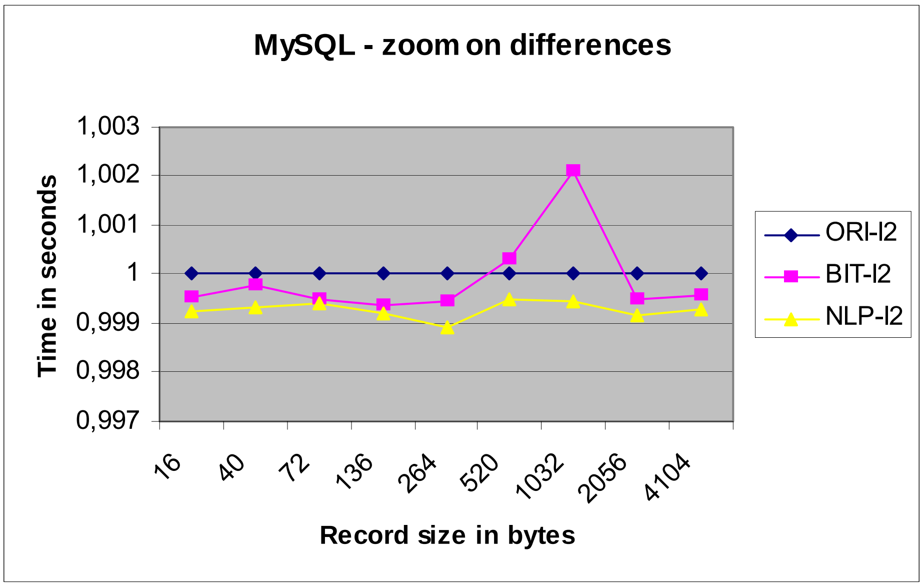


fig. 6.9 – *The differences in running time when table scanning on Itanium*

Since the number of leaves is constant one would expect the speed differences to be constant as well, i.e. unaffected by the record size. With 16 bytes as the only exception, none of the record sizes are a power of 2. Hence it is unlikely that looking up next-leaf pointers caused any cache misses as explained in section 3.2. The speedup due to following next-leaf pointers is nearly constant for all record sizes. Except for record sizes around 1000 bytes, the speedup due to bit-shifting is nearly constant as well and less beneficial than NLP. Although the irregular behaviour of bit-shifting is reproducible we have not been able to identify the cause.

The chosen record sizes may seem rather small but it should be noted that the maximal column size is 255B and hence 4104B corresponds to a table of at least 17 columns.

We expect the speed-differences to be even smaller on the P4-platform for two reasons: First, the figures in table 6.1 indicates a smaller speedup on P4 and second, the amount of RAM on the P4-platform is only 256MB leading to a maximum of app. 1000 leaves of 128KB in practice (3125 leaves in the Itanium-benchmark). Hence we consider it unrewarding to repeat this time-consuming test on the P4-platform.

6.7 TABLE SCAN IN MYSQL (MEMORY COPYING OPTIMISED)

Now we turn our focus to the memory copying optimisation and explore how the table scan running time is affected. In this first benchmark we have examined the impact of the number of columns (all of 129B) that are part of the query for a small and a large record size. The running times in the benchmark correspond to table scans in which none of 25,000 records satisfy the selection criterion.

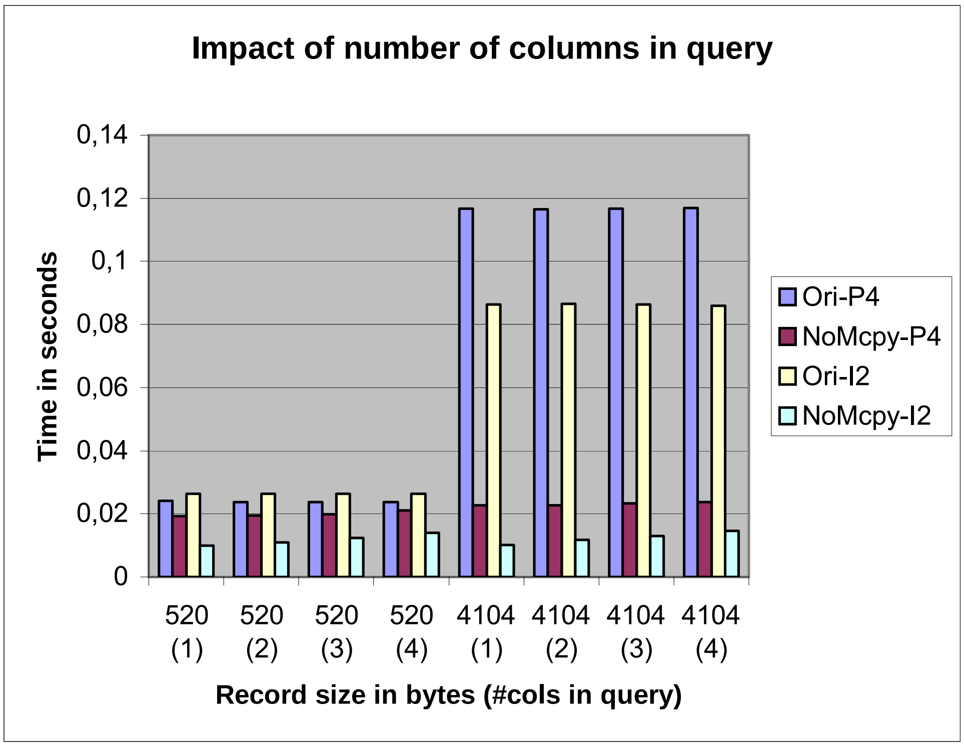


fig. 6.10 – *Table scan running times with and without memory copying*

As we expected the running times of the original version are unaffected by the number of columns but substantially affected by the record size (due to the copying). On the contrary the running times of the optimized version increase proportionally with the number of columns in the query (due to field pointer updating) and are only marginally affected by the overall record size. Adding one extra column to the query appears to have a larger impact on Itanium 2 than on P4.

In the following two benchmarks we aim for a larger perspective by studying how record and column size influences running time. We show best-case and worst-case behaviour of the optimized version by selecting a single column and all columns. The running times correspond to table scans in which none of 25,000 records satisfy the selection criterion.

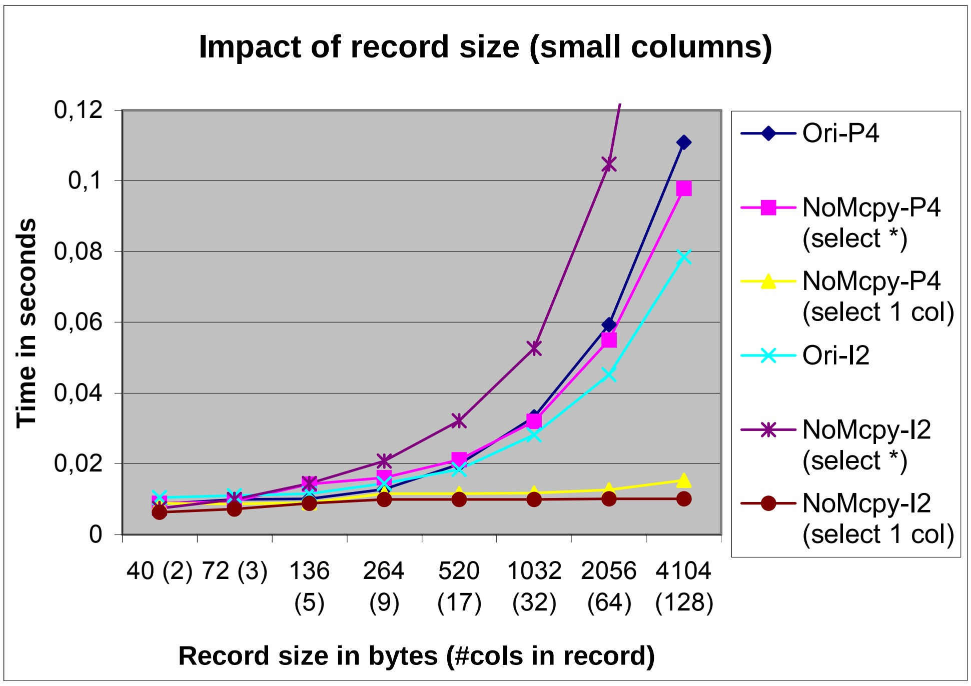


fig. 6.11 – *Table scan running times with a fixed column size of 32B*

To our surprise the worst-case behaviour (in which all columns are part of the query) on Itanium 2 is significantly worse than the original version for most record sizes. On the other hand the worst-case on P4 is no worse than original version. It strongly suggests that the current field-pointer update routine is sub-optimal on Itanium. On the positive side the best-case running time (single-column selection) is nearly constant and radically improves the running time for all but the smallest record sizes. The question is whether an “average” column size of 32B is representative. Below we have repeated the benchmark with a column size of 128B.

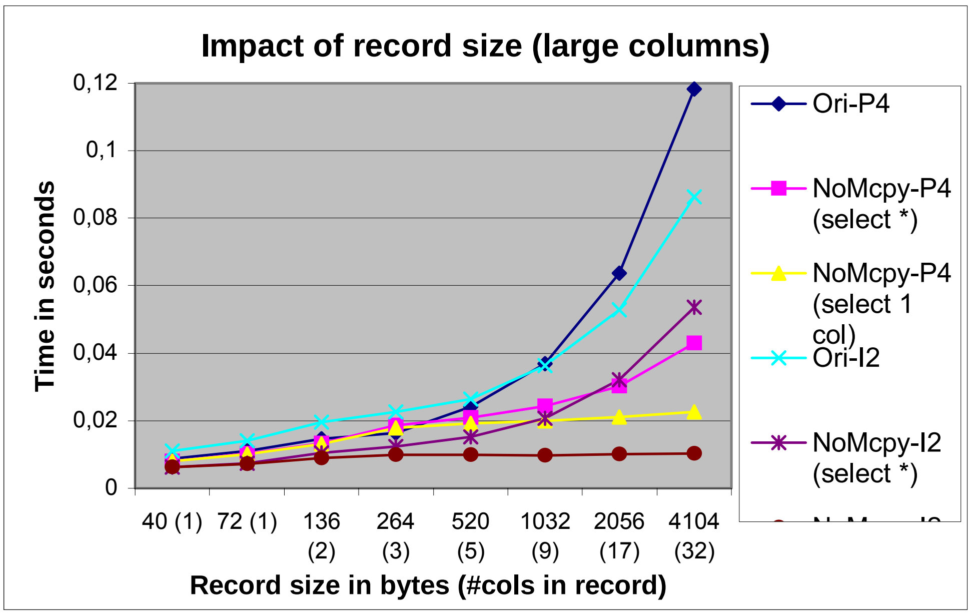


fig. 6.12 – *Table scan running times with a fixed column size of 128B*

Quadrupling the column size is evidently in favour of the optimized version since fewer field pointers need updating compared to the amount of data that needs to be copied and now the worst-case running time of the optimised version is at least as fast as the original version - no matter the choice of record size.

According to the last figure the running time can be (best-case) reduced to 80% of the original running time on Itanium 2 and 38% on Pentium 4 for a record size of 520B. By increasing the record size to 4104B the running times can be reduced to 12% and 19% respectively. Obviously the results would be even better if the column size was further enlarged.

6.8 TABLE SCAN IN MYSQL (TREE TRAVERSAL + MEMCPY OPTIMISED)

In this final benchmark we combine the next-leaf pointer optimisation with the memory copying optimisation. We have already shown that optimising tree traversal has a minuscule impact on the total running time of table scan. The sole purpose of this benchmark is to examine whether the *relative* benefit of the next-leaf pointer optimisation is worth mentioning when memory copying is no longer part of table scan. The best-case scenario comprises large records (4104B), large columns (255B) and a single-column, no-match query. The table scan has been repeated 1000 times to ensure sufficiently large differences in running time.

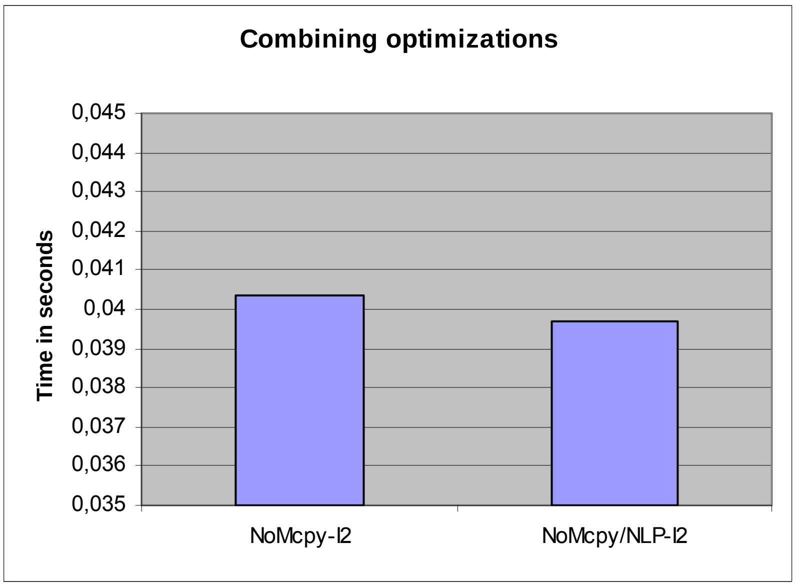


fig. 6.13 – *Table scanning 100.000 4104B-records on Itanium 2 when combining memory copying optimisation with tree traversal optimisation.*

The benchmark indicates that a speedup of no more than 1,6% can be expected in a best-case scenario despite the fact that memory copying has been eliminated.

6.9 FINAL REMARKS

The experimental results show a significant speedup of locating the next leaf expressed in percentages. However, with a table size in the order of 400MB the reduction of the *original* table scan running time on Itanium 2 is 0,25% at best and the speedup is unlikely to be more than 1,6% in the memory copying optimised version. As 50000 leaves of 128KB correspond to a Memory table of no less than 6.1 GB, the MySQL-benchmark in section 6.5 suggests that the wall-clock speedup due to next-leaf-pointers (on the Itanium 2-platform with a 6.1 GB Memory table) is 14 ms. Late in the process of writing this report we managed to get a profiling tool up and running. It confirmed that less than 0,5% of the time is spent in the tree traversal code and it furthermore confirmed that the copying of each record to the *current record*-buffer is the salient bottleneck indeed. Hence, the benefit of optimizing the leaf-locating part is hardly worth mentioning.

Memory copying optimisations are far more important as the gap between CPU speed and memory speed has been constantly growing, making the memory relatively slower over time and hence making excessive copying more costly. Even though the problem is somewhat alleviated by caches and their increasing size, the current situation where all data processed is copied in memory will flood most, if not all, caches. The best one can hope for is that where a prefetching algorithm is available, it will be strong enough to reduce the waiting period. Also, as cache-sizes grow, so will the number of primitives to explicitly change cache-behaviour. For instance, the Itanium offers an API for explicitly prefetching data into cache. This is an interesting development because cache misses are so expensive that they can severely degrade an application’s performance. At the same time, it is important to keep the cache management algorithms simple since time spent in these just add to the overall cache latency.

7. CONCLUSION

In this paper we first set out to investigate the practical value of optimizing the application of the tree data structure in the Memory storage engine in MySQL 4.1. Our optimizations consisted of speeding up tree traversal through bit-shifting and minimizing the number of tree traversals by using next-leaf pointers. Through simulations we examined the potential of the optimizations on fetching the next leaf node. Then, by benchmarking table scan in MySQL we succeeded in measuring the practical value of the optimizations. This led us to investigate the cost of what seemed to be superfluous copying between different areas in RAM.

In case of optimal cache behaviour the simulations suggested that tree traversal based on bit-shifting is 400-700% faster than the current tree traversal algorithm based on the division-operator. In practice for finding leaves in the tree, the tree traversal speedup due to bit-shifting is probably in the range 25%-75% while following next-leaf pointers appears to result in a speedup in the range 70%-120%.

Unfortunately, the total time spent in the tree traversal area is too small compared to, for instance, copying of records between the storage engine and the query processor. This means that even if our relative speedup for the tree traversal is significant, the effect on total running time is not impressive, i.e. a speedup in the order of milliseconds. Even when we remove the copying, which is the primary time-consumer, and could be expected to have absorbed some of the initial tree-scan costs, the impact on the total running time of table scan in MySQL is minuscule (a speedup of < 1,6%).

In contrast we have shown that there is great speedups to be gained if the query processor's access to records is redesigned, i.e. elimination of the copying of records between storage engine and query processor. In optimal cases the table scan running time can be reduced to 1/8 on Itanium 2 and 1/5 on Pentium 4. The worst-case running time matches the original running time on Pentium 4. However, the worst-case behaviour is awful on Itanium 2 because the field-pointer update routine is based on a ListIterator-implementation that performs poorly on Itanium 2.

As we have previously pointed out the copying of records between the Memory storage engine and the query processor is the major time-consumer when table scanning. It might be beneficial to insert explicit pre-fetching instructions in the source code or to adjust the implementation in ways that urge prefetching-enabled compilers to automatically insert data-cache prefetching instructions. This way the records will be readily available in cache when the copying takes place.

The current field-pointer update routine that we used in the memcpy-optimized version appears to be sub-optimal on Itanium 2. It might be beneficial to re-implement or circumvent the applied MySQL-ListIterator (which is commonly used in MySQL).

9. REFERENCES

1. Michael L. Samuel & Anders U. Pedersen, “MySQL in a Main Memory Database Context”, *Department of Computer Science, University of Copenhagen*, 2004

2. John L. Hennesey & David A. Patterson, “Computer architecture: a quantitative approach, 2nd edition”, *ISBN 1-55860-372-7, Stanford & Berkeley, USA*, 1996

3. “IA-32 Intel® Architecture Optimization”, *Order Number 248966-010, Intel, USA*, 2004

4. “Intel® Itanium® Architecture Software Developer’s Manual, Volume 1: Application Architecture. Revision 2.1, October 2002”, *Document Number 245317-004, Intel, USA*, 2002

5. D. W. Jones, “An empirical comparison of priority-queue and event-set implementations”, *Communications of the ACM 29, 4, 300-311*, 1986

6. “Intel Pentium 4 Processor Optimization Reference Manual”, *Order Number 248966, Intel, USA*, 2001

7. Lenz Grimmer: “List: MySQL Packagers”, *http://lists.mysql.com/packagers/179*, April 2004

8. “Intel® C++ Compiler for Linux* Systems User's Guide”, *Document number: 253254-014, Intel*, 2003

9. “Intel® Itanium® Architecture Software Developer’s Manual, Volume 2: System Architecture. Revision 2.1, October 2002”, *Document Number 245318-004, Intel, USA*, 2002

10. “Intel® Itanium® Architecture Software Developer’s Manual, Volume 2: Instruction Set Reference. Revision 2.1, October 2002”, *Document Number 245319-004, Intel, USA*, 2002

11. Joshua A. Redstone, Susan J. Eggers and Henry M. Levy, “An Analysis of Operating System Behavior on a Simultaneous Multithreaded Architecture”, *Proceedings of the 9th International Conference on Architectural Support for Programming Language and Operating Systems*, November 2000

12. “Intel® Itanium™ Processor Reference Manual for Software Optimization”, *Document Number: 245473-003, Intel, USA*, 2001

13. ”IA-32 Intel® Architecture Software Developer’s Manual – Volume 1: Basic Architecture”, *Order Number 253665, Intel, USA*, 2004

14. “IA-32 Intel® Architecture Software Developer’s Manual – Volume 2: Instruction Set Reference A-M”, *Order Number 253666, Intel, USA*, 2004

15. “IA-32 Intel® Architecture Software Developer’s Manual – Volume 3: Instruction Set Reference N-Z”, *Order Number 253667, Intel, USA*, 2004

16. "IA-32 Intel® Architecture Software Developer’s Manual – Volume 4: System Programming Guide”, *Order Number 253668, Intel, USA*, 2004

17. “A Formal Specification of Intel® Itanium® Processor Family Memory Ordering – Application Note”, *Document Number: 251429-001, Intel, USA*, 2002

18. Glenn Hinton, et al, “The Microarchitecture of the Pentium® 4 Processor”, *Intel Technoloy Journal Q1*, 2001

19. Jean-Francois Collard & Daniel Lavery, “Optimizations to Prevent Cache Penalties for the Intel® Itanium® 2 Processor”, *Intel, USA*, 2003

20. “Intel® Itanium®2 Processor Reference Manual – For Software Development and Optimization”, *Order Number: 251110-002, Intel, USA*, 2003

21. “Introduction to Microarchitectural Optimization for Itanium®2 Processors”, *Document Number: 251464-001, Intel, USA*, 2002