How to Barter Bits for Chronons: Compression and Bandwidth Trade Offs for Database Scans

Allison L. Holloway[1], Vijayshankar Raman[2], Garret Swart[2], David J. DeWitt[1]
{ahollowa, dewitt}@cs.wisc.edu, {ravijay, gswart}@us.ibm.com

[1]University of Wisconsin, Madison
Madison, WI 53706

[2]IBM Almaden Research Center
San Jose, CA 95120

ABSTRACT

Two trends are converging to make the CPU cost of a table scan a more important component of database performance. First, table scans are becoming a larger fraction of the query processing workload, and second, large memories and compression are making table scans CPU, rather than disk bandwidth, bound. Data warehouse systems have found that they can avoid the unpredictability of joins and indexing and achieve good performance by using massive parallel processing to perform scans over compressed vertical partitions of a denormalized schema.

In this paper we present a study of how to make such scans faster by the use of a scan code generator that produces code tuned to the database schema, the compression dictionaries, the queries being evaluated and the target CPU architecture. We investigate a variety of compression formats and propose two novel optimizations: tuple length quantization and a field length lookup table, for efficiently processing variable length fields and tuples. We present a detailed experimental study of the performance of generated scans against these compression formats, and use this to explore the trade off between compression quality and scan speed.

We also introduce new strategies for removing instruction-level dependencies and increasing instruction-level parallelism, allowing for greater exploitation of multi-issue processors.

Categories and Subject Descriptors: H.2 [Database Management]: Systems

General Terms: Design, Performance, Experimentation

Keywords: compression, bandwidth trade offs, Huffman coding, difference coding

1. INTRODUCTION

In the past, scan speeds have been limited by the I/O data path and the layout of the data on external storage; thus speed ups in scan performance have resulted from storage and file system improvements. Recently, changes in technology have made table scans more interesting to database designers. This is because:

- Table scans give predictable performance over a wide range of queries, whereas index and join-based plans can be very sensitive to the data distribution and to the set of indexes available.
- Disk space is now cheap enough to allow organizations to freely store multiple representations of the same data. In a warehouse environment, where data changes infrequently, multiple projections of the same table can be denormalized and stored. The presence of these materialized views makes table scans applicable to more types of queries.
- Scan sharing allows the cost of reading and decompressing data to be shared between many table scans running over the same data, improving scan throughput.
- A table scan is embarrassingly parallelizable; the data can be spread over many disks and the scan can be performed by many processors, each performing its own portion of the scan independently. As hardware has gotten cheaper, it has become possible to tune the level of parallelism to control the response time.
- With the advent of parallel database appliances that automatically manage the individual processing elements inside the appliance, scale-out parallelism (*e.g.,* clusters), which has long been much cheaper to purchase than scale-up parallelism (*e.g.,* SMPs), has become competitive in terms of management overhead.
- Large addressable memories on inexpensive 64-bit processors allow entire databases to be partitioned across the main memories of a set of machines.
- Advances in database compression allow databases to be compressed to between 1/4 and 1/20 the original sizes in practice. This allows larger databases to fit into main memories, and it allows for a proportional effective increase in the data access bandwidth.

Additionally, compression is a key technology for table scans. One reason compression has generated so much interest recently is that, with the advent of multi-core processors, the rate at which arithmetic operations needed for decompression can be performed has been increasing more quickly than the data access bandwidth rate [8]. This trend is expected to continue for many years and suggests that, eventually, we will be willing to pay for any amount of additional compression, even if it comes at the expense of additional arithmetic operations.

But, programs are not written to run "eventually:" they are written for and run on a particular target system; thus, the amount of compression must match the current system.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'07 June11–14, 2007, Beijing, China.
Copyright 2007 ACM 978-1-59593-686-8/07/0006 ...$5.00.

In this paper we explore how to generate code that optimizes the CPU speed at which data in a particular format can be scanned. Depending on the bandwidth and capacity of the data storage and the CPU speed, the format that best trades off compression and CPU utilization can be chosen.

There are presently three schools of thought on database compression:

- *Neo-compressionists:* This work focuses on substantially reducing the space used for representing data in the database, using techniques like run-length coding, Lempel-Ziv coding, entropy coding, delta codes, and extensive pre-compression analysis of the data (for example, [17, 15]). In particular, a recent paper [15] describes a method to compress a relation down to its entropy, and shows compression factors of 8 to 40 on projections of TPC-H data. These researchers do not use the standard slotted page format, instead they build compression blocks which have to be decompressed as a unit.
- *Strict vectorists:* These researchers match the compression format to the capabilities of the hardware to allow exploitation of multi-issue and vector capabilities. The data is stored column-wise, with each field, or small sequence of fields, encoded in a fixed-length machine data type [12, 3]. Domain coding is used extensively to reduce long and variable length items to a small fixed size. This allows very fast queries and random access on the data, but the compression ratio is not as good as the entropy coding techniques. While this format can be updated in place because of its fixed length nature, most implementations do not emphasize the speed of updates.
- *Slotty compressors:* Most commercial DBMSs have adopted a simple row-at-a-time compression format that builds on the traditional slotted page format but utilizes an off-page dictionary. These DBMSs support update in a straightforward way and have reduced the engineering cost by decompressing pages when they are loaded from disk into the buffer pool [10, 14]. These formats are not optimized for scanning, but instead are designed to support update in place. We do not consider these formats further.

This paper explores the spectrum of formats in between the maximum compression formats and the vectorized compression, while keeping the focus on scan speed. In our experiments we measure the CPU cost of the scan: the number of cycles consumed for the scan, select, project, aggregate, and group by, with the data stored in main memory. Our experiments start only after the data has been read into main memory. The effect of I/O on scans [8] has been studied and data from a given I/O system could be folded into our CPU results.

Our scanning approach follows in the footsteps of MonetDB [3]: custom C code is generated for each query, rather than using interpreted code. This is needed to generate the best possible scan for a given implementation, and it is especially important when operating over compressed data because of the gains that can be made by customizing code, not just based on the data types and the query, but on the data encoding method and the actual contents of the dictionaries.

We start the body of this paper (Section 3) with a detailed analysis of compression formats. We classify the variety of relation compression ideas that have been used in the past into a spectrum, ranging from highly compressed (and hard to scan) to lightly compressed (and easy to scan). Two of the recurring ideas in the compression of relations are difference coding and variable length (Huffman) coding. We explore several relaxations of these two that trade off the extent of compression for scan speed. These relaxations form the spectrum of compression formats for our experimental study.

Prior researchers have made several assumptions about operating on compressed data, such as (a) bit alignment is too expensive and fields should be byte aligned [1], and (b) variable length fields are too expensive to decode [1, 21]. Two contributions of this paper are:

- to experimentally verify these claims; our results (Section 5) suggest that the first is not valid, unless the scan is highly CPU bound.
- two new optimizations, *length-lookup tables* and *length quantization*, for computing lengths of variable length entities. Our experimental results suggest that with these two optimizations, Huffman coding does become viable for many queries.

Section 4 addresses scan generation. Superficially, a scan is simple, but the performance depends on a careful implementation of each detail. Accordingly, we discuss the ideas behind generating code to do a scan, and we identify several details that are vital for good scan performance: special handling of variable and fixed length fields, techniques for short circuiting, and techniques to minimize conditional jumps.

As part of this we also discuss how to generate code that exploits the parallelism available on modern CPUs: code that exploits simultaneous multi-threading (SMT), multi-core and multi-issue instruction scheduling capabilities. We also present a new technique that increases the exploitable Instruction Level Parallelism (ILP) in a scan over a compressed table, which is called *interleaving*.

Finally, Section 5 presents our performance study. The experimental results are for two architectures, Sun Ultrasparc T1 and IBM Power 5, to illustrate the compression-speed trade off of various formats. Based on these results, we provide our main contribution: an outline of the rules and the data needed to determine the level of compression to use on systems with a given ratio of CPU speed to data bandwidth.

2. BACKGROUND AND RELATED WORK

Work on database compression has been around for almost as long as databases themselves; even INGRES used null value suppression and some form of difference coding [18]. However, much of this work has tended to focus either on heavyweight compression with high compression ratios or lightweight compression with lower compression ratios, with the focus on both improved I/O system performance and decreased storage capacity.

Much of the early work focused on compressing entire pages of the database. Though this led to good compression, page decompression was very expensive, particularly if only a few tuples were needed from the page. Graefe and Shapiro [7] were among the first to suggest tuple and field-wise compression and operating on still-compressed data; they noted that many operations, such as joins, projections and some aggregates, could be performed on compressed tuples.

Dictionary based domain compression, a lightweight compression method where data values are mapped to fixed length codes, is used in many database implementations and

has been the subject of extensive research [2, 11, 14]. Entropy coding techniques, including Huffman encoding [9], are considered heavy-weight techniques, since decompressing variable length entropy encoded data is more processor intensive. Both Huffman and arithmetic encoding have been studied and modified to work better in databases [16, 5, 15, 20].

Early compression work mainly focused on disk space savings and I/O performance [6, 5, 10]. Later work acknowledges that compression can also lead to better CPU performance, as long as decompression costs are low [16, 19, 13]. Some work has been done to examine the effect of compression on main memory databases, and most look at choosing one compression scheme and finding the execution time savings from it [11]. However, some recent work notes that query optimizers need to include the effects of compression in their cost calculations [4, 1].

Commercial DBMS implementations have generally used either page or row level compression. IBM DB2 [10] and IMS [5] use a non-adaptive dictionary scheme. Oracle uses a dictionary of frequently used symbols to do page-level compression [14].

The MonetDB project has explored hardware compression schemes on column-wise relations [21], but the work assumes that the records must be decompressed in order to query them, while we try to avoid decompressing. Their work focuses both on compression and decompression performance while our work does not include the costs of compressing and tries to decompress as little as possible. The MonetDB compression work also presents a technique they call double-cursor, which has goals similar to what we call thread interleaving, increasing the available instruction level parallelism, but it is used to encode the records, and it does not physically partition the records.

C-Store [17] is a system that does column-wise storage and compression. It also delta codes the sort column of each table. Work has also been done to use compression in C-Store in order to decrease query times [1]. However, much of the work focuses on light-weight compression schemes, such as run-length encoding, which are more suited to column stores than row stores. Their work supports our belief that it is best to decompress as little of the data as possible. They have also explored how expected query workloads should affect the column encoding and what the optimizer needs to know about the implications of direct operation on compressed data.

The careful work on scanning performance done by Harizopolous et al.[8] has shown the effect of compression on disk layout performance, showing how the cost of joining data from multiple disk streams can be compared with the cost of skipping over data from a single stream. Compression's effect on the results was analyzed and, for many work loads, it changed the system from being I/O bound to being CPU bound. A very interesting aspect of this work was the use of the cycles-per-byte system ratio to generalize the results beyond their test system and to understand the momentum of future systems. This influenced our work greatly.

3. DATA REPRESENTATION

There are many different ways to represent data to allow efficient scanning; this section discusses a number of alternative formats and when they are useful. Multiple formats are examined because no single format wins in all cases: there is a tension between formats that allow for more efficient processing of the data and formats that represent the data more compactly. Depending on the relationship between where the data is stored and the access bandwidth to that data, and the processing power that is available for the data, one format or another will be more efficient. The consensus is that over time the ratio will tilt in favor of instructions, and this trend leads us to consider formats that are more heavily compressed and more expensive to process today.

This section begins by categorizing data formats on three dimensions, ordering the options within each dimension by decreasing processing speed and increasing representation efficiency. These dimensions are: column coding, the way that individual column values are coded to form column codes; tuple coding, the way that the column codes are combined to form a tuple code; and block coding, the way that multiple tuples are combined to form a compression block.

Individual database systems implement a particular point from each dimension that represents the trade offs made by the designers. For example, in the entropy-compressed format of [15], the implementors have chosen the most highly compressed form in each dimension. The more lightly compressed format chosen by MonetDB [3] uses intermediate values in each dimension. Today, traditional databases, most of them designed when the compute-to-bandwidth ratio was skewed toward the bandwidth side, have focused on the easiest-to-process format.

We believe that, ultimately, the choice of data format will become an integral part of physical database design: this would involve analyzing the dataset, computational environment, and workload to determine the best data format. For example, on a certain set of columns, variable length entropy coding might save such a small amount that fixed length domain encoding might be preferable, while on other, highly skewed columns, the reverse decision might be made.

Figure 1 illustrates the three dimensions of data formats, whose components are described below. Note that, in combination, this amounts to over one hundred formats, and the experimental section only gives results for those viewed to be most promising.

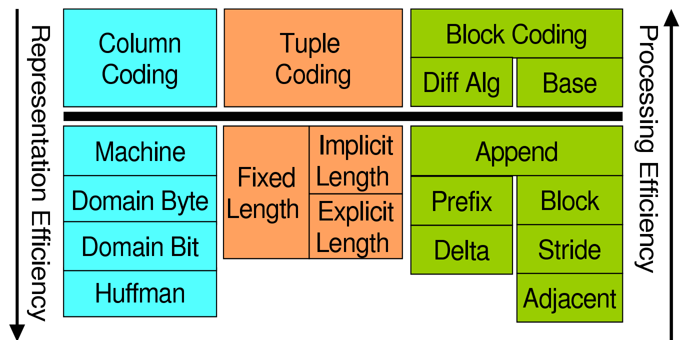


Figure 1: Spectrum of compression formats from most to least compressed

3.1 Column Coding

A column code is the string of bits that is used to represent a particular column value. We consider several ways of translating a column value into a column code.

Machine coded data represents each column value as a data type understood directly by the machine instruction

set, that is, as records and arrays of fixed length data items represented as one, two, four, or eight byte integers, characters, and floating point numbers. Values can be loaded into registers with a single instruction and some machines have special SIMD registers and instruction sets that allow operations on arrays of small values to be performed in one vector operation. Using the machine data types minimizes the work needed to get the data into the registers of the machine in an efficient format. In addition, it is possible to randomly access into an array of such rows, which is important for following physical references to data items. This representation also entails the smallest amount of CPU overhead, since the data can be accessed with little more than a load and an array increment. However, this format is the most inefficient in space and, thus, bandwidth because it aligns column codes to machine boundaries, adds extra zero bits to the high end of words, and pads variable length strings to their maximum lengths.

Domain coding assigns a fixed length field to each column according to the cardinality of the column, not the value of the column. The mapping between the column code and the column value is maintained arithmetically, where applicable, or by using a dictionary and storing the index. The mapping used is stored in the meta-data of the representation. For example, if a CHAR(15) column can take 7 distinct values, it can be coded into a 3 bit domain code. We can potentially improve the efficiency of accessing domain codes if the code length is extended to a byte boundary. For example, instead of coding the character string above with 3 bits, it could be padded to 8 bits (one byte).

Huffman coding assigns each column value a variable length code chosen so that more frequent values within the column are assigned shorter codes. The important implication for scan efficiency is that fields are now variable length and their length has to be determined before the next field in the tuple can be accessed.

Canonical Huffman [15] coding is a reordering of the nodes of a Huffman tree so that shorter codes have smaller code values than longer codes and codes with the same length are ordered by their corresponding value. This format results in the same amount of compression as normal Huffman, but it supports both efficient code length computation as well as equality and range checking, without decoding or any access to a large data structure. We want to avoid accesses to large data structures since they may result in a cache miss, and cache misses are very expensive on modern processors.

Finding the length of a canonical Huffman code is important, but can be difficult. Once the length of the code is known, range queries can be efficiently performed directly on the code, the code can be mapped into an index in a dense map of the values (for decode) or the same index can be used to access an array representing the results of an aggregation GROUPed BY the column.

There are two ways to compute the length of a canonical Huffman code. One option, presented in previous work [15], is to search in the Huffman mini-dictionary, which is a list of largest tuple codes whose first Huffman code has the given length. This can be done in time proportional to the log of the number of distinct code lengths. For example, if a canonical Huffman dictionary has codes: 0, 100, 101, 110, 1110, and 1111, the mini-dictionary will store the largest code for each code length left-shifted and padded with 1 bits to the word length of the machine, i.e. {01111111, 11011111, 11111111}. Thus, to find the length of the different fields in the tuple code 11101000, one would begin by comparing 11101000 to each of the three entries in the mini-dictionary, stopping when the mini-dictionary's entry is no longer less than or equal to the tuple code, in this case at 11111111. The length associated with that entry is 4, so the first field is of length 4.

Length-Lookup Table: In this paper we introduce an alternative technique for computing the length using a *lookup table* indexed by a prefix of the tuple code. A Length Lookup Table replaces these multiple comparisons with a single array lookup using the code itself. The challenge is to keep this array small (ideally, to fit in the L1 cache).

A simple solution is to have an array LEN of 2^k elements, where k is the longest code length. For the example dictionary, k=4, LEN = {1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 4, 4}. LEN[0]..LEN[7] are 1 because indexes 0..7 have a common prefix '0' in base 2. LEN[8]..LEN[13] are 3 because the indexes have prefixes 100/101/110. Now, the length of the code prefix in 11101... = LEN[4-bit prefix of 11101...]

To reduce the array size, let j be the second longest code length (3 in the example). Observe that codes of length j have to differ from codes of length k in the first j bits itself (because it is a prefix code). So, the array only needs 2^j elements: LEN = {1, 1, 1, 1, 3, 3, 3, 4} (*e.g.,* LEN[0-3] = 1 because the indexes are 000-011 base 2, sharing prefix '0').

To tokenize the tuple code 11101000101: index into LEN with the 3-bit prefix 111, giving a code length of 4, and column code '1110'; shift left by 4 to form 1000101; index into LEN with 100 (or 4), so the length is 3, and column code is '100'; shift left by 3 to get 0101; LEN[010] = 1, so the next column code is '0'; shift left by 1, and LEN[101] = 3, so the last column code is '101'.

The advantage of table lookup over binary search is that it involves no branches; mispredicted branches are quite expensive on modern architectures. The disadvantage is that the table can get long; table lookup is only efficient when the entire table fits in the L1 cache. So, as part of reading in a canonical Huffman dictionary, the mini-dictionary is always constructed, but the look up table is only constructed if it would be much smaller than the L1 cache.

Compression must take into account the processing cost, and dictionaries must be constructed to optimize efficient length computation and compression efficiency. The experiments presented in this paper use the lookup table because of its better performance. To allow full exploitation of the length lookup table, a canonical Huffman dictionary generator was developed that matches a given histogram and has only one code length greater than 8. This limits the size of the length look up table to 256 bytes. The resulting code does not result in optimal compression, but for many column distributions it is quite good.

3.2 Tuple Coding

Once the column codes have been constructed, the next step is to combine the columns into a larger code that represents the entire tuple. There are two competing approaches for combining columns: a column store and the more traditional row store.

In a column store, each column is stored separately from the others; the correspondence between columns in a tuple is maintained either by using the same order of values between columns, or by augmenting each column value with a tuple

identifier. The latter adds a substantial amount of extra size to the relation, and is not considered further. The former is essentially a *column major* layout of a table projection. Predicates are evaluated separately on each column, with bitmaps storing the results. The bitmaps are then anded together to form the results of the tuple predicate. Recent work comparing column stores and row stores [8] has shown that, for simple queries, storing the columns separately only adds a small amount of overhead, even when all columns are required, and can save a great deal because of saved I/O when the query only requires a subset of the columns.

If the query workload is predictable, disk space is free, and a high update rate is not required, there is a another alternative to either row stores or column stores: studying the workload and storing views of the more heavily used fields, and querying the projection that has the least number of extra columns, as is done in C-store [17]. This is the approach taken in this paper. The results in this paper use projections that are stored in row major order; we believe that our results should be applicable to either approach.

Because a tuple is formed by combining the column codes for each attribute, fixed length fields are best placed at the front of the tuple. This allows the initial fields to be examined without waiting to determine the length of the previous variable length column codes. There is some tension between this rule and the desire to order the columns to improve the efficacy of tuple difference coding. Tuples that are entirely comprised of fixed length fields allow for greater parallelism within a scan, as the next tuple can be processed without waiting to determine the length of the current tuple.

When processing queries with low selectivity, there are many times when a predicate will fail based on an initial column code, which means the rest of the tuple does not need to be processed. In such a case, the evaluation of this tuple should be *short circuited*. But short circuiting requires knowledge of where the next tuple starts, which is non-trivial for tuples with many variable length columns that must have their lengths individually decoded and summed. To ease this process, we have added a variation on tuple coding where the total length of the tuple is explicitly and redundantly encoded as part of the tuple header, through a technique termed *fixed length quantized (FLQ) compression*, which is described below.

3.3 Block Coding

Compression blocks are sequences of tuples that have been combined to form a unit so they can be updated and coded on a per block basis. Each compression block must be able to be interpreted independently in order to increase parallelism. Compression blocks are processed in parallel using both thread-level and instruction-level parallelism. The compression blocks are sized so that thread processing overhead (*e.g.,* pthreads) is dominated by the cost of scanning the block.

In our implementation, compression blocks each hold the same number of tuples. This makes it easier to scan several compression blocks simultaneously within a single thread, as will be described in the next section. Short compression blocks may be handled through a less optimized path.

The simplest and fastest approach to block coding is just to append the tuples codes. Since each tuple is either of fixed length, or the length can be determined by looking at the tuple, this is acceptable.

To increase compression between tuples, difference coding can also be applied. To perform difference coding, first the tuples are sorted lexicographically. The sorting and the differencing does not take into account any column boundaries. To maintain block independence, no differencing is done between blocks; the beginning of a block assumes a zero valued previous tuple.

The main idea of difference coding is that instead of storing the actual tuple codes, a compressed version of the difference between this tuple and the previous tuple is stored. Difference coding m tuples saves approximately $log_2 m$ bits [15]. For a terabyte of random data (a few billion tuples) the savings is 30-35 bits per tuple. In practice, difference coding correlated data can save even more bits per tuple.

There are two parameters that define difference coding: the difference operator and which previous tuple is used as a base for the differencing. The difference operators we support are prefix coding (also called XOR coding) and delta coding (which uses arithmetic subtraction). Difference coded tuples have the form: the first tuple, the number of consecutive zeroes at the head of the difference between the first and second tuples, then the tail (the numerical difference between the tuples), continued for all tuples in the block. For instance, the difference between 1023 and 1000 using prefix coding would be 1111111111 xor 1111101000 = 0000010111. The number of zeroes at the head of the difference is 5, and the numerical difference between the tuples would be 10111. Thus, the two tuples difference coded would be 1111111111 0101 10111. The number of leading zeroes has been padded to 4 bits to include all the possible numbers of leading zeroes. We do not have to explicitly store the first '1' at the head of the tail since, by definition, the tail begins at the first '1' in the difference. This optimization would lead to the code: 1111111111 0101 0111.

The number of zeroes at the head of the difference are normally encoded in order to save more space; these could be encoded using a Huffman code, but that slows down decode. To speed up the differencing decode, we use the same type of coding we use for the tuple length: a FLQ code, described below.

Our analysis and experience shows that delta codes are two bits shorter than prefix codes over random data. To see the advantage of delta coding over prefix coding, consider the coding of the numbers 1 to m. With delta coding the differences between codes is uniformly 1, which can be coded given the schema above in one bit: a zero length code and a one bit tail. With prefix coding, the expected length of the tail will be zero bits half of the time, one bit a quarter of the time, two bits one eighth of the time, and so on, which works out to 1 bit on average. Using a similar sum for the length of the code, the code will be two bits long on average, which results in a 3 bit code.

Even though prefix codes are a bit longer than delta codes, they are nicer for processing than delta coding for two reasons: multi-precision arithmetic is more complex than the multi-precision masking, and knowing the number of bits in common between two tuples allows us to reuse processing that was done on the first tuple. For example, if by examining the first 57 bits of a tuple we are able to determine that this tuple does not meet the selection predicate, then if the length of the identical prefix is 57 bits or longer, we can throw out the new tuple without any processing.

However, any form of difference coding cuts down on po-

tential parallelism within the processing of a block, because the next tuple cannot be decoded until the previous tuple has been decoded. To mitigate this problem, reference coding has been proposed and implemented in MonetDB [3]. In reference coding, instead of representing the difference between pairs of tuples, we compute our differences from each tuple and the initial tuple in the block. In this way, after the first tuple has been computed, all the other differencing operations can go on in parallel. We generalize this notion and call the reference tuple the *base* of the differencing. Using a block level base instead of the adjacent tuple as a base can cost, on average, $log_2 k$ bits if the block is of length k (follows from Lemma 1 and 2 of [15]).

To split the difference between instruction level parallelism and compression, we propose using an intermediate base, we call the stride. A stride is a small sequence of tuple codes, typically 2, 4 or 8. Each tuple in the stride uses the same base, which is the last tuple code of the previous stride. Using this value costs slightly less than $log_2 stride$. If the stride is chosen to match the level of loop unrolling or vectorization, then stride coding can be done with no extra overhead. Using a stride in this range allows for plenty of instruction level parallelism while not costing nearly as much compression efficiency.

3.4 Fixed Length Quantized codes

Even with the advantages of canonical Huffman, variable length columns still cost more than fixed length columns to decode in terms of increased instruction count and reduced instruction level parallelism. For this reason we would like to use fixed length codes to represent the tuple lengths and prefix lengths. These values have a large range, so how can we use a short fixed length field to represent them? The answer is by quantizing the value, using what we call a Fixed Length Quantized code or FLQ code. Quantization works because we don't actually have to represent the exact bit count for these fields: we are trading off compression for easier processing. We can always add more padding bits, as long as we know that is what they are. So, if we had a 45 bit prefix when all the rest of the prefixes were 48 bits, we could just add an extra three bits at the end.

As part of compression we run an optimizer that, based on a histogram of lengths, determines the optimal number of bits to use for the FLQ code and the optimal FLQ dictionary. An FLQ dictionary of a given length is an ordered list of values that are accessible by an index; thus, decode is just a fixed length bit extraction of the index followed by a table lookup. The FLQ dictionary is computed using a recursive dynamic programming algorithm, taking into account the length of the FLQ code and the extra bits that have to be included due to quantization errors. Since the FLQ dictionary rounds the value represented up or down, we call it quantized compression.

For example, in our example TPC-H dataset the lengths of the tuple codes averaged 82.1, hitting every value between 81 and 110. Coding the length straight would have consumed 7 bits and using fixed length domain code with arithmetic would have consumed 5 bits. Instead, using an FLQ we coded the length in two bits, representing the quantized tuple code lengths 82, 83, 90, and 110. We then paid an average of 1 bit per tuple in quantization errors for a total cost of 3 bits per tuple. The cost for a Huffman code would have been 2.5 bits per tuple.

Similarly, the average length of the differential prefixes was 72.3 bits on average and ranged over 41 values between 8 and 90. This would require 6 bits as a domain code. FLQ used 3 bits to represent the prefix length, and paid an additional 0.7 bits per tuple for quantization errors which brings the total cost per tuple to 3.7, the same as the Huffman code cost.

4. SCAN GENERATION

This section describe how to generate code to scan the tuples inside a compression block. Section 4.2 use this skeleton as an atom to do a parallel scan over a table composed of multiple compression blocks.

At scan generation time, a lot of information is known about the data to be scanned and the query to be run. All of this information is hard-coded into the generated scan code:

- number of tuples in each compression block
- number of columns in the tuple
- the dictionary used to encode each column inside the tuple. This includes information about the relative probability of the values.
- the maximum size of the tuple code, computed by adding the maximum size of each column. This size is used to determine the number of 64-bit words needed to store a tuple, usually two or three. These words are declared in the scanner as separate automatic variables of type 'unsigned long long.' Having them as variables increases the chance that the compiler will keep them in a register.
- the query. We currently implement only a subset of SQL queries. Specifically, we consider only conjunctions of equality and range predicates on the columns and sum and count aggregates, with group-by. Queries always return at least one aggregate, since we wanted to focus on the CPU cost of the scan, and not the I/O cost of producing output tuples.

Figure 2 shows a skeleton of generated code for a scan over a table with 5 fixed and 3 variable length fields. We will use this to explain the generation logic.

Our general scan model involves repeatedly fetching one tuple after another into local variables, and processing the tuple one constituent field at a time. The processing of fields within a tuple is hard coded (*e.g.,* process field C and then F, G, and H) rather than loop-based – this is the advantage of a generated, as opposed to interpreted, scan.

4.0.1 Fetch Next Tuple

Within the main loop, the first operation is a call to *nextTuple* to fetch the next tuple in the compression block. nextTuple() is specialized to both the kind of block coding used in the compression block and the maximum length of the tuple[1]. If the block is coded via arithmetic deltas, nextTuple() reads the delta and adds it to the previous tuple. If it is coded via prefix deltas, nextTuple() shifts the delta and XORs it with the previous tuple. Both the addition and the XOR are multi-precision operations, but we have found that XOR is much more efficient than addition since there is no

[1]Even though the fields within a tuple are themselves compressed (via domain or Huffman coding), for most schemas the tuple takes up more than one machine word (64 bits).

```
FOR i = 1 TO NUM_TUPLES_PER_BLOCK DO
    /* Fetch tuple into local variables tRegs */
   nextTuple(tOffset, tRegs, &tupleLength &sameBits);
    /* Operate on field C */
     fldOffset = CONSTANT_OFFSET_OF_FLD_C;
     fcode3 = getField(tRegs, fldOffset, &flen3);
     /* work on fcode3 */
    /* Operate on field F */
     fldOffset = CONSTANT_OFFSET_OF_FLD_F;
     vcode1 = getField(tRegs, fldOffset, &vlen1);
     /* work on vcode1 */

    /* Skip field G */
     fldOffset = CONSTANT_OFFSET_OF_FLD_F + vlen1;
     vcode2 = getField(tRegs, fldOffset, &vlen2);
     fldOffset += vlen2;
    /* Operate on field H */
     vcode3 = getField(tRegs, fldOffset, &vlen3);
     /* work on vcode3 */
    fldOffset += vlen3;

    /* go to next tuple */
    tOffset += fldOffset;
```

Figure 2: Generated code for scan over an 8-column table. Arrows denote instruction dependencies.

need to check for carries, which eliminates the potential for branch misses.

nextTuple() takes in a tuple offset and optionally returns the tuple length (if the format embeds the length). The tuple offset is set to 0 for the very first tuple in the block. For subsequent tuples, the offset is incremented in one of two ways:

- by summing the lengths of each of the fields as they are processed.
- by adding the embedded tuple length to the starting offset.

The latter is the basis for short-circuiting a tuple after processing only a subset of its fields (Section 4.1).

4.0.2 Processing Fields within a Tuple

nextTuple() is followed by a series of operations on fields within the fetched tuple. All the compressed formats considered place fixed length codes first. Thus, all the fixed length codes, *plus the first variable length codes* are at constant offsets within the tuple. For these fields, the scan generator outputs a block of code for each field referenced in the query (C, F, H in Figure 2).

But the rest of the variable length fields are at variable offsets, and have to be processed in order, even if they are not referenced in the query (like field G). Furthermore, the offset calculation means that operation on these fields is serialized, as shown by arrows in Figure 2.

Extracting a Field: The first step in operating on a field is to extract it from the tuple (still leaving it compressed). For a fixed length field, getField() is just a shift and mask with constant. For a variable length field, the mask depends on the length. getField() computes the length by either searching the mini-dictionary or through a table lookup, as discussed in Section 3.

A useful common-case optimization is for equality predicates. If the only operation on a field is an equality check with a literal, the mask is always a constant, even for a variable length field, because it is determined by the length of the literal.

Operating on a Field: At this point, the field is Huffman or domain coded. If the query needs the field only for a predicate (equality or range), or for a group-by, coded field itself is operated on. For aggregations, the field is decoded.

Equality comparisons on Huffman coded fields are simple: the codes are checked for equality. Range comparisons exploit the order-preserving property of canonical Huffman codes, as suggested in [15]: at each level in a canonical Huffman tree, the values at the leaves are sorted in ascending order from left to right, thus, their Huffman codes are then also in ascending order. This means that if values v_1 and v_2 code to the same length, then $code(v_1) \leq code(v_2)$ iff $v_1 \leq v_2$. So, to calculate if $field > literal$, the scan generator pre-computes an array $minCode$ of the smallest code at each length (*i.e.,* level of the Huffman tree) that satisfies the literal. Then the predicate is evaluated as $code(field) > minCode[field.length]$. A search of $minCode$ is not needed because the field's length can easily be calculated from the length-lookup table.

Grouping and aggregation are done by maintaining an array of running aggregates, one for each group. In the absence of compression, group identification is typically done by a hash function. But group identification is *easier for compressed fields*, because the domain for canonical Huffman codes is dense within codes of each length.

For example, one might want to know the volume of sales grouped by the week of the year, to see which week has the most business. The aggregate statement would be SUM(sales) group by week, and the resulting calculation would be:
$sum[groupByIndex(week)] += decode(sales)$.
The groupByIndex maps the field code for week into a spot in the sum array. In this case, the mapping would be very simple, subtracting one from the week number (1-52). The field code for sales would also have to be decoded into its number value.

Canonical Huffman makes determining the index very easy since codes of the same length are consecutive integers. To compute the $groupByIndex$, the length of the code is determined, as described in Section 3. The length of the code is then used as an index into a small array storing the precomputed count of the number of codes of shorter length. This value is then added to the index of this code within codes of the same length to get the result, *e.g.,*
$groupByIndex[c_f.length][c_f]$ and $decode[c_e.length][c_e]$,
where $groupByIndex[len]$ is an array of the aggregate-array indices for codes of length len, and $decode[len]$ is an array of the values for codes of length len.
A domain coded field is treated the same as a Huffman code where the Huffman dictionary contains a single length. All the operations are exactly as for Huffman codes, but are implemented more efficiently because the length (and dependent quantities like minCode[len] and groupByIndex[len]) are constants within the generated code.

4.1 Short Circuiting

As discussed in Section 3, difference coded tuples may or

may not embed the tuple length in the sequence of tuples. However, knowledge of the tuple length greatly impacts the scan. Without knowing the tuple length, difference coding merely adds an extra step before the standard processing of the tuple, i.e. there will be an *undifference* tuple step after the nextTuple statement above, because the other fields' lengths must be found in order to move to the next tuple. But if the tuple's length is known, the next tuple can begin processing as soon as the current tuple fails the predicate, or the last aggregate is performed. The process of skipping to the next tuple as soon as possible is commonly referred to as short-circuiting.

The program flow in Figure 2 would include an "if" statement after each applied predicate to determine if the tuple needs to be processed further. Also, the tOffset increment would change to add the embedded length, rather than the fldOffset.

Short-circuiting increases the number of branches (since at least some predicates will fail), but the cost of mispredicted branches may be less than that of decoding the rest of the fields in the tuple. Also, short-circuiting difference coded tuples can allow us to re-use already computed information from the previous tuple, depending on how similar the previous and current tuples are, as was discussed in Section 3. This optimization comes at the cost of increased code complexity.

4.2 Parallelism

Today's processors are highly parallel: most new and upcoming processors have multiple threads (most often of a type called SMT) and/ or multiple cores on one die (CMP). While processors and compilers are good at extracting instruction level parallelism, they cannot split a stream of data into multiple parallel streams of data (partitions); it is something we, as programmers, must do for it.

The records are partitioned using one of two methods: evenly spaced cursors into the data or compression blocks.

If the tuples are only Huffman coded, the records can simply be partitioned by providing different pointers into the file. However, if the tuples are difference coded, compression blocks must be formed, since one tuple is dependent on the previous tuple, thus starting in the middle would render the compression unparseable.With compression blocks, difference coding must periodically be started anew. Each new start of difference coding signals the start of a new compression block, and all but the last compression blocks are forced to have the same number of records.

4.2.1 Interleaving

Breaking the records into independent blocks allows us to simultaneously perform the same actions on each block of records, all in one thread. This parallel action on independent blocks within one processing core, or *interleaving*, allows us to exploit the superscalar properties of processors: they can execute multiple instructions per cycle, as long as they have instructions to execute. If only one block of data is being processed, dependences within the data (for instance, a control dependence from a branch, or a data dependence on the length) limit ILP. But, if there are multiple blocks to work on - say 1-3 more, the processor can have more instructions to execute, thus reducing stall time. The example scan in Figure 2 only has one interleave. To modify it to include multiple interleaves, the same code block would be repeated multiple times within the loop, each working on a different sequence of tuples.

4.2.2 Threading

Today's code needs to be threaded in order to fully exploit SMTs and CMPs. We use pthreads and multiple compression blocks to achieve this. The scan code itself might also include interleaving, in which case more pointers into the sequence of records or compression blocks will need to be passed to the scan. Each thread consists of the code loop illustrated in Figure 2, and the necessary pthread and data structure initialization code.

4.2.3 Orthogonality

The different forms of parallelism we introduce are, in general, orthogonal to other issues. However, different configurations may have different optimal amounts of exposed parallelism, since some possibilities are so compute-intensive that no additional parallelism needs to be provided. The next section further discusses these issues.

5. EXPERIMENTAL EVALUATION

This section presents an extensive performance evaluation of our generated scans. It also discusses how much of the processors' parallelism was able to be exploited and the effectiveness of short circuiting.

Different formats have widely varying compression ratios and hence (memory/disk) bandwidth requirements: formats that need less CPU to operate over the data tend to need more bandwidth to scan the data. Any complete system that uses one of these formats would have to have an I/O and memory subsystem that can support this bandwidth: a format that compresses poorly will need a powerful memory/disk subsystem, while a format that scans poorly will need a powerful processor.

The focus of this paper is *not to present a single such balanced system*, but is instead to lay out the trade off between compression (bandwidth requirement) and scan speed (CPU requirement). The bandwidth requirement is directly calculated from the compressed tuple sizes (bits/tuple). We measure the CPU requirement by measuring the scan speeds (ns/tuple) on two different architectures. To make sure that our scans are entirely CPU-bound, the data sets were kept entirely memory resident.

Section 5.2 studies the performance of scans without predicates to measure the tokenization efficacy of each format. Section 5.3 studies scans with predicates. We investigate how well our queries can speed up using processor parallelism in Section 5.4. Finally, Section 5.5 discusses these results in terms of the number of instructions/byte that each format needs, and correspondingly which format is appropriate for which architecture.

5.1 Methodology

The experiments were performed on 2 machines: a 2-core Power 5 (running AIX) and an 8-core 1GHz Sun T2000 server (running Solaris 10). The relative performance of each compression format is almost the same on the two machines, so most results are reported for the Sun only.

We use a TPC-H projection with attributes from the natural join of the lineitem, customer, and supplier tables. (Table 1) As in [15], two kinds of skew are added: (a) the customer and supplier nations distribution are chosen per the

Schema	LPK, LPR, QTY, OPR, WK, DAYOFWK, YR, SNAT, CNAT
Q1a	select sum(LPR) CNAT from table group by CNAT
Q1b	select sum(LPR), DAYOFWK from table group by DAYOFWK
Q1c	select sum(LPR), QTY from table group by QTY
Q2	select sum(LPR), QTY from table group by CNAT where LPK=1 and CNAT=USA
Q3	select sum(LPR), QTY from table group by CNAT where LYR>1990 and LYR<8515

Table 1: Schemas and Queries (LPR is l_extendedprice, LPK is partkey, QTY is o_quantity, OPR is o_totalprice, WK, DAYOFWK, YR represent o_orderdate, SNAT and CNAT represent s_nationkey and c_nationkey).

	Length Embedded (AL)	Vanilla (A)	Delta Coded + Length Embedded (DL)	Delta Coded (D)
0V (byte): Byte-aligned, all fixed	N.A	128	43.04	43.04
0V (bit): Bit-aligned, all fixed	N.A	105	35.56	35.56
1V (bit): Bit-aligned, 1 var len	101.66	100.66	32.25	31.24
2V (bit): Bit-aligned, 2 var len	98.32	96.32	28.91	26.91
3V (bit): Bit-aligned, 3 var len	87.46	85.43	19.48	17.46
4V (bit): Bit-aligned, 4 var len	86.81	84.24	19.25	16.68

Table 2: Compressed sizes (bits/tuple) for various tuple encodings (columns) and field encodings (rows).

Column	Uncompressed	Can. Huff
LPK	20	N.A.
OPR	18	N.A.
QTY	6	N.A.
LPR	24	N.A.
WK	6	N.A.
DAYOFWK	4	2.81
YR	15	4.11
SNAT	6	1.66
CNAT	6	1.66

Table 3: Size in bits of each column either uncompressed or with canonical Huffman coding applied.

distribution of Canadian trade from the W.T.O, and (b) the dates distribution ranges until 10000 A.D., with 99% in 1995-2005, 99% of those being weekdays, and 40% of those are in the two weeks around Christmas and Mother's day. All other columns used the standard (uniform) TPC-H data distribution. Dates are stored as year, week, and day-of-week because there is often skew on weeks and day-of-week (*e.g.,* more sales on weekends and before Christmas). Thus, the schema consists of 5 fixed length fields (LPK through WK), and up to 4 variable length fields (DAYOFWK through CNAT) that can benefit from Huffman coding.

By default, the TPC-H data generator produces two kinds of correlation in this schema: LPR is functionally determined by the LPK, and the OPR is LPR × QTY.

The dataset models a size of 1B rows, but, to keep the experiments manageable, only a 32M row slice of it was generated in sorted order; queries were only run on this slice. The experiments were explicitly designed so that I/O bandwidth would not be an issue, since we wanted to measure the CPU cost of operating over each format and the bandwidth needed for each format.

We study the speed of scans over different compression formats created by 4 tuple encodings and 3 field encodings:

- The tuple encodings arise from whether the tuples in the block are difference coded (D) or just appended (A), and whether or not lengths are embedded (via the fixed length quantization introduced in Section 3) (L): they are denoted as delta (D), append (A), delta+len (DL), append+len (AL) in the plots.
- The field encodings are: all fixed length byte aligned, all fixed length bit aligned, and Huffman coded (with the length-lookup table optimization). The schema has 4 fields with skewed distributions: DAYOFWK, YR, SNAT, CNAT. So, the study has encodings with 1, 2, 3, and 4 Huffman coded fields. The resultant six field encodings are marked as zero-B for byte alignment, zero-b for bit alignment), 1V (bit), 2V (bit), 3V (bit), 4V (bit) in the plots to indicate the number of variable length fields the scan operates on.

These formats and the compression sizes they obtain are tabulated in Table 2. Results for each column are presented in Table 3. The uncompressed size is 320 bits/tuple. In contrast, the most compressed format considered, 4V.D, takes up 16.68 bits/tuple, and the entropy (calculated from the distribution) is 11.86 bits per tuple. The 4.82 bits/tuple difference is due to (a) not co-coding the correlated fields, (b) XOR coding differences rather than Huffman coding differences, and (c) using Huffman coding instead of arithmetic coding.

5.2 Tokenization Performance

Query 1 examines the cost of tokenizing a compressed table into tuples and fields. This is measured by running queries Q1a, Q1b, and Q1c: aggregation and group by queries that have no predicates. The three queries differ in the extent of short-circuiting they permit over the variable length fields. Q1a, a grouping on the very last field (CNAT) allows no short-circuiting: every field has to be tokenized. Q1b, a grouping on the first variable length field, allows partial short-circuiting. Q1c, a sum on a fixed length field, allows full short-circuiting.

Q1a, Q1b, and Q1c are run on data compressed in each of the 6 field encodings and 4 tuple encodings. Figure 3 plots these numbers for the Sun. For comparison, Table 4 lists the numbers on Sun and Power-5 for Q1a. Note that tuple encodings that embed lengths for the fixed length (0V or

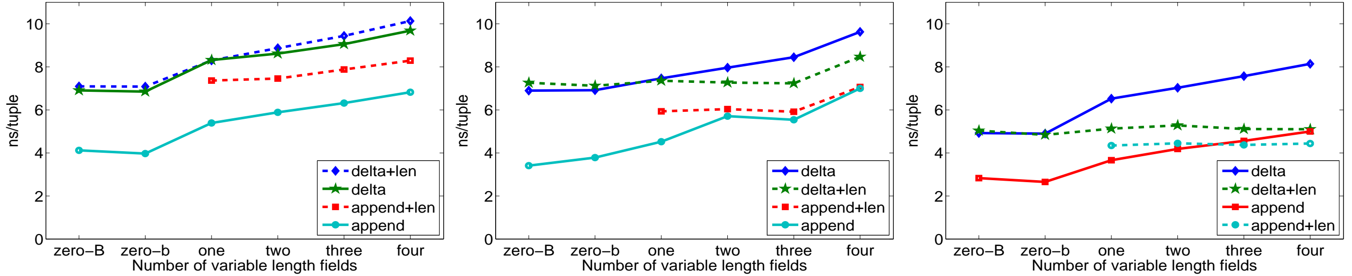


Figure 3: Scan Speed (ns/tuple) vs Compression Format as a function of the number of variable length fields for queries that allow varying levels of short-circuiting (Q1a, Q1b, Q1c). Zero-B is for byte-aligned tuples, and zero-b is for bit-aligned tuples. The results are for a Sun-T1 with 32 threads and one interleave.

Power-5 (2 threads, 2 interleave)						
	0V (byte)	0V (bit)	1V	2V	3V	4V
A	4.00	4.00	5.36	7.08	9.58	13.83
AL			7.33	7.18	7.57	14.53
D	9.54	9.23	11.38	12.13	11.35	16.66
DL	9.96	9.97	11.97	13.50	14.89	22.58
Sun-T1 (32 threads, 1 interleave)						
	0V (byte)	0V (bit)	1V	2V	3V	4V
A	3.41	3.79	4.52	5.71	5.54	7.00
AL			5.93	6.04	5.91	7.06
D	6.90	6.91	7.47	7.96	8.45	9.63
DL	7.26	7.12	7.35	7.27	7.23	8.47

Table 4: Scan Speed (ns/tuple) for Q1a on Power-5 and Sun-T1

zero-b/B) field codings are not considered. For the purpose of this experiment, the parallelism level is set to a natural one for the architecture: 32 threads and 1 interleave for the 1-issue, 8-core-32-thread Sun, 2 threads and 2 interleaves for the 5-issue, 2-core Power.

In each plot, the y axis represents the scan speed in ns/tuple (calculated as (elapsed time)/(number of tuples)). The x axis enumerates the different field encodings, and the four curves represent the different tuple encodings. There are a few major observations to draw from this plot and table:

- Almost all scans are performed in under 10ns/tuple. On Power 5, these same scans ran in 10-20 ns/tuple.
- Delta coding costs approximately an additional 2-4 ns/tuple during scan.
- As expected, variable length fields reduce scan bandwidth. The slope of the curves indicates the cost for each variable length field. It is about 1 to 2 ns per variable length field for query Q1a, which cannot use short circuiting because of the select on the last attribute. But queries Q1b and Q1c, which can have short-circuiting, have much smaller slopes for the length-embedded formats append+len and delta+len (in Q1c, the time taken is almost independent of the number of variable length fields).
- The savings from byte alignment are negligible and not always present.

5.3 Predicate Evaluation Performance

Queries Q2 and Q3 study the cost of predicate evaluation as part of the scan. Q2 is a query with two equality predicates: one on a fixed length field and one on a variable length field. As before, the scan speed for each format is plotted, using 32 threads and 1 interleave (Figure 4). This plot is similar to the previous plots, except that the length embedded formats take the same time regardless of how many variable length fields are used. This is due to short circuiting, but the short circuiting is of a different nature than it was for Q1c in Figure 3. There, Q1c short circuited because it did not use the variable length fields. Here, Q2 uses the last variable length field, but it still short circuits because of the selective predicate.

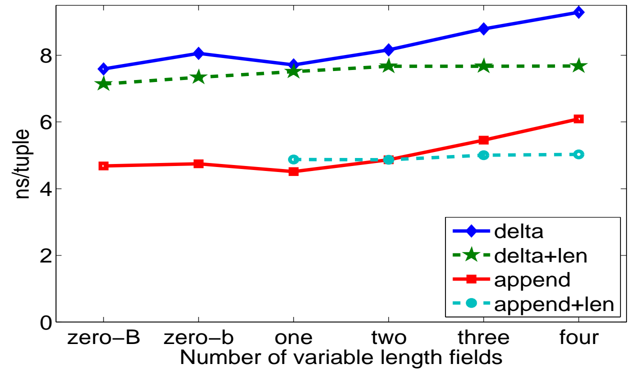


Figure 4: Scan Speed (ns/tuple) for Q2: one equality predicate on a fixed length field and one on a variable length field on the Sun-T1 with 32 threads and one interleave.

Q3 has a range predicate that is a between predicate on a variable length field. As Figure 5 shows, the performance is almost identical to that of the pure scan, except every number is 1-2ns/tuple higher. This represents the cost of evaluating a range predicate on a canonical Huffman coded field.

In our experiments, compilation (with the highest level of optimization) took on the order of a few seconds (1.5s-10s on the 1GHz Ultrasparc T1). This overhead is negligible for typical data warehouse queries that can run for hours. For shorter queries, we could compile multiple queries at the same time to amortize the cost of invoking the compiler, or directly output machine code. In the future we could consider generating the scan program in two compilation units, one with the inner loop (Fig 2) and one with the outer loop. We could then compile the compilation unit containing the inner loop with -O5 and use -O1 for the outer loop.

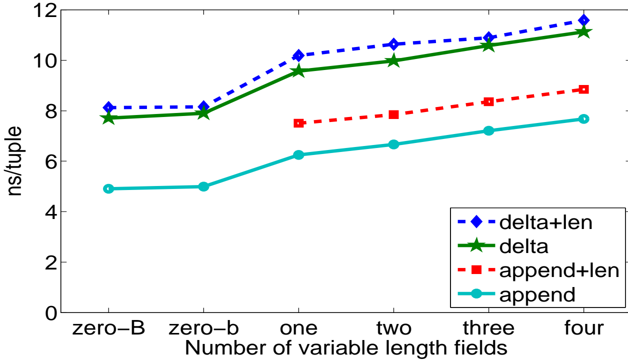


Figure 5: Scan Speed (ns/tuple) for Q3: 2 range predicates on the same variable length field on the Sun-T1 with 32 threads and one interleave.

5.4 Value of Parallelism

Processors now provide multiple forms of parallelism, and the data is partitioned in order to try to take advantage of as much of it as possible. The Sun-T1 has 8 cores, 32 threads, and is single-issue; thus, interleaved parallelism is not applicable, but thread parallelism is. Figure 6 plots the scan speeds in tuples/ns as a function of the number of threads used, for some of the compression formats. The main observation is that every format, including the ones that do difference coding, scales up very nicely. The speedup is almost linear until 8 threads. This shows that splitting tuples into compression blocks works; even though processing of difference coded tuples within a block is serialized, there is good parallelism across blocks.

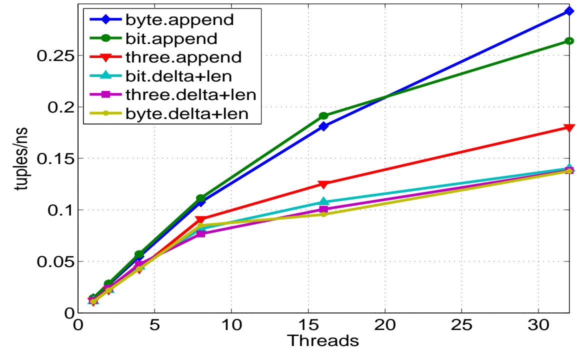
The Power 5 has 2 cores, 4 threads and is 5-issue, so both thread and interleaved parallelism are possible. The results from thread parallelism (not shown) are similar to that on the Sun, except the speedup tapers off after 4 threads. The results from interleaving were not very good. We have observed about a 20% to 50% speedup for 2-way interleaving, but performance deteriorates after that. Our hypothesis is that the scan code saturates all the available registers even at 2 interleaves. This needs to be investigated further.

5.5 Discussion: Which Format is Better?

Having seen the performance of a variety of compression formats, we are tempted to make judgments about which format is better. Our results have shown a clear trade off: the Huffman coded and difference coded formats that compress better generally scan more slowly.

But by keeping all our data sets in memory, we have expressly chosen to measure the CPU speed of the scan. For example, from Figure 3, the scan on the zero-B.Append (byte aligned, non difference coded) format runs at 4ns/tuple. But to actually realize this speed, the I/O and memory subsystem must flow 250M tuples/second into the processor.

We now plot the combined CPU and memory bandwidth requirements of these formats. Figure 7 plots the data from Figures 3, 4 and 5, in a different representation. The plot places six tuple formats on the x-axis (marked by vertical lines), scaled to their compressed sizes. For example, there's a triangle for query Q1c at (19.48, 5.1): a scan speed of 5.1 cycles/tuple with a size of 19.48 bits/tuple (the 3V.DL



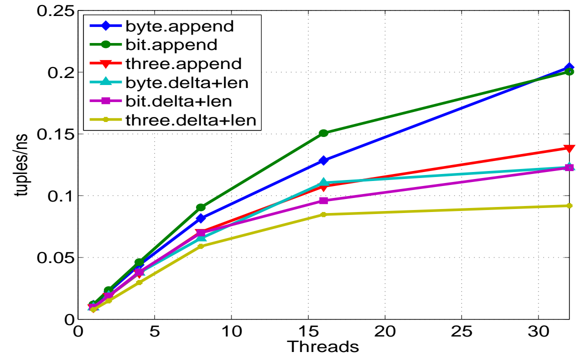


Figure 6: Scan Bandwidth on the Sun-T1 for a scan (Q1b:top) and a predicate on a variable length field (Q3:bottom)

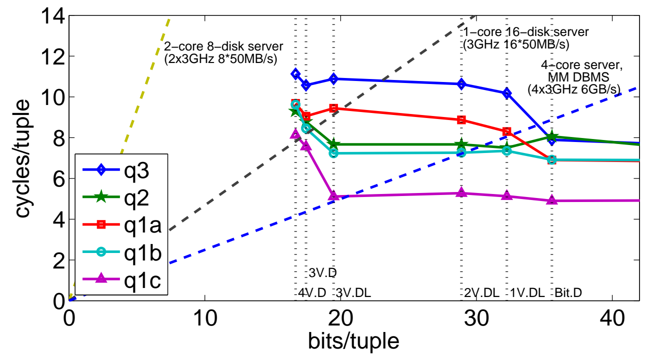


Figure 7: CPU cost vs Memory/ I/O Bandwidth cost for chosen formats on the Sun-T1 with 32 threads and one interleave. Formats larger than 40 bits/tuple or that are dominated in both cycles/tuple and bits/tuple are omitted.

format). The presented compression formats are either bit-aligned or have 1-3 variable length fields. They are all delta coded, and some also have their length embedded. None of the appended tuple formats appear on the graph, since their compressed sizes are larger than 40 bits/tuple and thus require too much memory bandwidth to be interesting for this discussion.

Which of these formats to choose depends on how balanced the system is, in terms of bandwidth into the processor and the speed of the processor. On the same plot, we have also drawn dashed lines of the instructions per tuple available under different system configurations. For example, a main-memory DBMS on a 4-core server might run at 4x3 GHz, but be able to access its memory at 6GB/s. For the main-memory DBMS, any format to the left of its dashed line is a viable configuration – formats to the right will not be able to get tuples into the processor fast enough. A system designer has to choose the right compression format by plotting the dashed line of instructions/bit available on her system, and then picking a format close to this line.

6. CONCLUSION AND FUTURE WORK

This work begins by reviewing myriad compression techniques, and discusses the granularities at which the compression can be performed: the column level, the tuple level, and the block level. The techniques range from the simple, such as domain encoding, which always results in fixed-length codes, to Huffman coding, which results in variable-length codes, to delta coding, which works on sequences of tuples. Each of the different techniques has a different cost and effect on scan time. We also introduce fixed-length quantized codes and a length lookup table to decrease processing time.

Currently, the level of compression in a system is fixed, and cannot be easily modified to reflect the system's abilities. In order to study the effects of compression at the CPU level across different systems, we created a scan generator, which takes compression, tuple representation, query and system parameters and outputs a custom scan C file. This custom scan file provides code for a fast scan, and may or may not include short-circuiting, interleaving, threading, and fast access to Huffman-coded fields. We believe that eventually database administrators will be able to use a physical format optimizer to tailor datasets to their computational environments and workloads.

All the results have shown that the Huffman coded and delta coded formats compress better but normally take more CPU time. However, these results do not include I/O time. When I/O and memory subsystem times are also included in the decision, the format to choose becomes less clear-cut. If a physical format optimizer or system administrator had this information and a fast scan generator, they could make a more informed choice as to the best way to store the data.

7. ACKNOWLEDGMENTS

The authors would like to thank Ken Ross and the anonymous reviewers for helpful feedback on the paper.

8. REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.

[2] G. Antoshenkov, D. Lomet, and J. Murray. Order preserving string compression. In *ICDE*, 1996.

[3] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.

[4] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. In *SIGMOD*, 2001.

[5] G. Cormack. Data compresson on a database system. *CACM*, 28(12), 1985.

[6] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing relations and indexes. In *ICDE*, 1998.

[7] G. Graefe and L. D. Shapiro. Data compression and database performance. In *Proc. ACM/IEEE-CS Symp. on Applied Computing*, 1991.

[8] S. Harizopoulos, V. Liang, D. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. In *VLDB*, 2006.

[9] D. Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the I.R.E.*, pages 1098–1102, 1952.

[10] B. R. Iyer and D. Wilhite. Data compression support in databases. In *VLDB*, 1994.

[11] T. Lehman and M. Carey. Query processing in main memory database management systems. In *SIGMOD*, 1986.

[12] R. MacNicol and B. French. Sybase IQ Multiplex - Designed for analytics. In *VLDB*, 2004.

[13] S. J. O'Connell and N. Winterbottom. Performing joins without decompression in a compressed database system. *SIGMOD Rec.*, 32(1), 2003.

[14] M. Pöss and D. Potapov. Data compression in Oracle. In *VLDB*, 2003.

[15] V. Raman and G. Swart. Entropy compression of relations and querying of compressed relations. In *VLDB*, 2006.

[16] G. Ray, J. Haritsa, and S. Seshadri. Database compression: A performance enhancement tool. In *COMAD*, 1995.

[17] M. Stonebraker et al. C-store: a column-oriented DBMS. In *VLDB*, 2005.

[18] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189–222, 1976.

[19] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *SIGMOD Record*, 29(3):55–67, 2000.

[20] A. Zandi, B. Iyer, and G. Langdon. Sort order preserving data compression for extended alphabets. In *Data Compression Conference*, 1993.

[21] M. Zukowski, S. Heman, N. Nes, and P. A. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE*, April 2006.