Constant-Time Query Processing

Vijayshankar Raman#1, Garret Swart#2, Lin Qiao#3, Frederick Reiss#4,
Vijay Dialani#5, Donald Kossmann*6, Inderpal Narang#7, Richard Sidle#8

#*IBM Almaden Research Center, San Jose, CA, USA*
{ 1ravijay@us, 3lsqiao@us, 4frreiss@us, 7inarang@us, 8rsidle@almaden}.ibm.com
2garret@swart.org

**ETH Zurich, Zurich, Switzerland*
6donald.kossmann@inf.ethz.ch

***Abstract*— Query performance in current systems depends significantly on tuning: how well the query matches the available indexes, materialized views etc. Even in a well tuned system, there are always some queries that take much longer than others. This frustrates users who increasingly want *consistent* response times to *ad hoc* queries.**

We argue that query processors should instead aim for constant response times for all queries, with no assumption about tuning. We present *Blink*, our first attempt at this goal, that runs every query as a table scan over a fully denormalized database, with hash group-by done along the way. To make this scan efficient, Blink uses a novel compression scheme that horizontally partitions tuples by frequency, thereby compressing skewed data almost down to entropy, even while producing long runs of fixed-length, easily-parseable values. We also present a scheme for evaluating a conjunction of range and equality predicates in SIMD fashion over compressed tuples, and different schemes for efficient hash-based aggregation within the L2 cache. A experimental study with a suite of arbitrary single block SQL queries over a TPCH-like schema suggests that constant-time queries can be efficient.

I. Introduction

Traditionally, query processors have been architected to solve the following problem: Given a specific workload, and a specific hardware configuration, choose the right data structures (indexes, materialized views, ...) and the right query plans to run the workload as fast as possible. Since the days of System R and INGRES, numerous sophisticated query processing algorithms and data structures have been developed towards this goal.

However, workloads are becoming less predictable, and more and more decision support is done interactively on the database itself. Hardware costs are also becoming less significant, when compared to the human cost of tuning the system. In this environment, the mettle of the DBMS lies not in how well it does on the queries for which it has been tuned, or even on the "average" query – but rather in how well it copes with the bad queries. The new standard, being driven by user familiarity with search engines, is to run each query in a constant time bound. We refer to this goal as *constant-time query processing*.

Since queries can vary widely, meeting this goal means assigning enough resources to each query so that it can meet its deadline. This puts a premium on query plans that have predictable cost, so we can estimate the needed resources, and dynamically scalable, so that additional resources can be used to reduce the response time.

A simple way to generate predictable query plans is to avoid structures that benefit particular queries: no secondary indexes, no materialized views, and using only workload-independent horizontal partitioning schemes to stripe data across nodes. If we denormalize the data as part of loading it, then many queries can be run as a *table scan*. While table scans generally access more tuples than index based plans, they have more consistent run times and are easier to parallelize.

In this paper we present Blink, our first shot at this goal. Blink is built around around highly efficient scans over denormalized tables. Scans can be expensive both in terms of I/O – more data has to be read into the processor, and in terms of CPU – more tuples have to be processed. Blink uses a novel compression technique that compresses tightly while being efficient to parse, a novel technique for SIMD predicate evaluation, and cache aware hash grouping algorithms. We preview these next.

Frequency Partitioning

Compression is one way of overcoming the I/O and memory bottleneck that scan is prone to. Our previous work on the compression of relations [12] coded values into variable length entropy codes, using shorter codes for more frequent values. Variable length fields force the scan operation to serially parse each field in the tuple, significant for tables with many fields. Blink avoids this cost by *Frequency Partitioning*, a new compression technique that compresses close to entropy while partitioning the data into long runs of fixed format tuples.

SIMD predicate evaluation

After parsing comes predicate evaluation. Blink uses order-preserving codes, so equality and range predicates are applied by coding the query rather than decoding each value. But even this is expensive if conjunctive predicates are evaluated in a typical loop: *extract each field – apply predicates on that field – if tuple passes, go to next predicate*, because the unpredictable branches and data-dependent shift instructions slow down modern processors. These costs are typically negligible in traditional warehouses where indexes provide data reduction, but are very important for a scan.

In Blink, we evaluate conjunctive equality and range predicates in parallel using a constant number of instructions, eliminating the loop and branches.

Efficient Hash Grouping

The final part of tuple processing is updating the running aggregates for the tuples that meet the predicates. This can become the dominant cost if the hash table does not fit in L2 cache or if the hash table operations involve unpredicted branches, as with traditional linear probing. In Blink we use two kinds of collision-free hash tables: one using the compression code itself as hash, and the other using a minimal perfect hash function. We adaptively choose the kind of hash table within each frequency partition depending on its number of distinct groups and the expected size of the hash table, so as to minimize cache misses while grouping and aggregating.

Overview: The rest of this paper is structured as follows: Section II gives some background on our compression scheme and an overview of related work. Section III describes frequency partitioning and compression in Blink. Section IV describes the run time, including predicate evaluation and hash aggregation. We present performance results in Section V, and conclude in Section VI.

II. Background and Related Work

Blink is not the first system that relies heavily on scans as opposed to indexes. The Netezza appliance, as well as many "column stores" (*e.g.*, Monet DB [17], C-Store [14], and SAP's T-Rex engine [8]) mainly do scans. Column stores lay out tuples in column-major order, and run queries by scanning through each column to find lists of row-ids that match predicates on that column, intersecting the lists to evaluate conjunctions, and finally fetching other columns needed for aggregation or grouping.

Blink, on the other hand, is a *row store*. Column stores and row stores each have their merits: while the former scan less data on a per-query basis, the latter are simpler to update and save the cost to fetch columns for aggregation. Studying these tradeoffs is beyond the scope of this paper: our goal is to improve the performance of row stores, which are used in almost all widely deployed DBMSs today.

The use of table scans in a row store introduces particular challenges, such as SIMD predicate evaluation on non-byte aligned fields. We describe techniques for applying conjunctive equality and range predicates in SIMD fashion, going beyond previous work on SIMD evaluation of equality predicates (*e.g.*, as part of Monet DB).

Denormalizing the database schema and answering queries as table scans has been studied extensively before, particularly in the context of universal relations [7].

The idea to compress databases to improve query speeds has been explored previously in the context of both column stores and row stores. Most of that previous work, however, relies heavily on fixed length codes; e.g., Sybase IQ [9] and C-Store [14]. Fixed length codes are easier to implement, and leverage the homogeneity of the data layout in order to do fast array-based operations. But, fixed-length codes achieve lower compression rates than variable-length codes. In order to provide more flexibility for compression, Monet DB [17] introduced the idea of using a fixed length code for almost all values and variable length codes for outliers. This idea can be seen as a first step towards frequency partitioning, which generalizes this concept, using multiple partitions for each column according to the value distribution of each column. Westmann et al. support variable-length codes in the granularity of *bytes* [15]. That work advocates the use of embedded pointers to length indicators to speed up the decoding of variable-length encodings. [4] devises a length-lookup table to find lengths in a single lookup. In both of these techniques, the cost of parsing and decoding is linear in the number of variable length fields; in contrast, Blink provides constant time decoding.

Blink builds on our previous work on lossless compression of relational databases down to their entropy [12]. We briefly review this technique because Frequency Partitioning extends this compression scheme.

The simplest form of relational compression is to replace repeated values with *codes* that index into a *dictionary* mapping codes to values. For example, a CHAR(10) field with 10 distinct values can be replaced by 4-bit codes.

Many domains have highly skewed data distributions: even though many values are possible, only a smaller number are likely, and a still smaller subset predominate. Dictionary coding can be extended to handle skewed data by using variable length codes: shorter codes for frequent values. For example, Huffman coding [5] produces variable length codes that are *prefix-free*: i.e., no code is a prefix of another code. So we can pack multiple codes together without any delimiters or length indicators.

The compression scheme of [12] replaces each field in a tuple with a Huffman code, chosen based on the frequency distribution for that column. These *fieldcodes* are then concatenated together to form a *tuplecode* for the entire tuple.

Further compression is possible by stripping out tuple ordering. Relations are sets, so any information about the order of tuples in a relation is redundant information. A system can remove this redundancy by sorting and *delta-coding* compressed tuples. Instead of storing the binary representation of every tuple directly, delta-coding represents each bit string as a difference, or *delta*, from the previous tuple's bit string. Since these deltas are relatively small numbers, they can be encoded in fewer bits than the compressed tuples, and can be further compressed using an entropy code.

[12] employs Huffman codes in conjunction with delta coding. The concatenated tuplecodes are sorted lexicographically and then delta-coded. [12] also shows that this compression is the best possible, in that it compresses relational data to within a constant bits/tuple of its entropy.

While this compression format is tight, it is tough to parse because each variable length code has to be parsed separately. [4] studies several ways to make this parsing efficient; nevertheless, the cost is around 2ns per tuple per variable length code, which is expensive for tables with many columns.

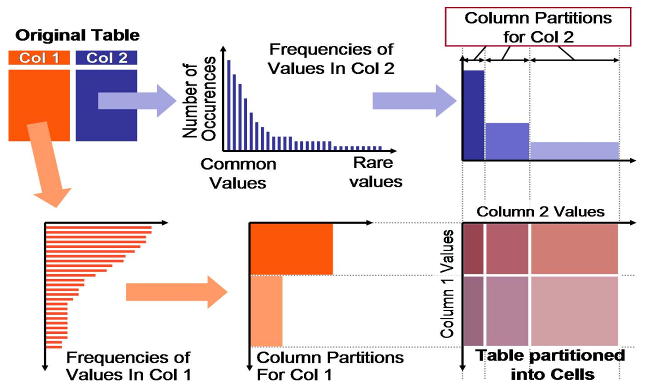

Original Table
Col 1
Col 2
Number of Occurences
Frequencies of Values In Col 2
Common Values
Rare values
Column Partitions for Col 2
Column 2 Values
Column 1 Values
Frequencies of Values In Col 1
Column Partitions For Col 1
Table partitioned into Cells

Fig. 1. Partitioning a two-column table.

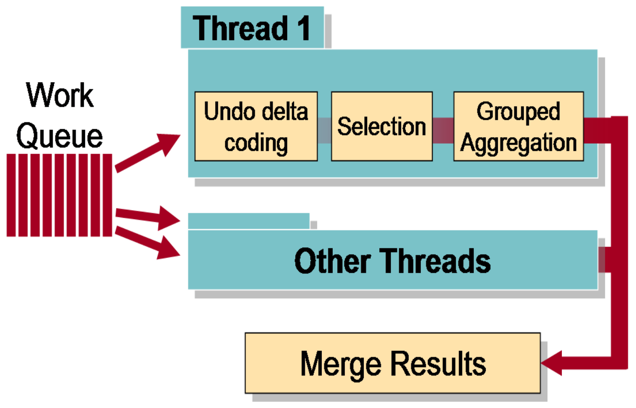


Fig. 3. The Scan operator takes a work queue of cells of compressed tuples, and processes it with a pool of worker threads.

In Blink we use a new Frequency Partitioning scheme that achieves almost the same compression but produces long runs of fixed length codes.

Finally, Blink's query processing algorithms are based on perfect hashing. There is a rich literature on perfect hashing (eg [3]), though databases have traditionally used variants of chained hashing. Recently, several researchers have investigated cuckoo hashing, a dynamic collision-free hash, for query processing [11], [16], [13]. To the best of our knowledge, our work is the first to apply different hash functions on different data partitions, and to use precomputed perfect hash functions to avoid branches in hash lookups.

III. Compression by Frequency Partitioning

Recall that a tuplecode is the concatenation of dictionary codes for each field of a tuple. To access the fieldcode for the i'th field of a tuplecode, the system must parse fieldcodes 1 through $i-1$ to determine their code lengths. This creates control and data dependencies that severely impact performance on modern processors.

This overhead is a well known problem, pointed out in [15], [4]. [4] measures the cost of this decoding as about 2 nanoseconds per variable length field per tuple. Westmann *et al.* [15] present a solution using a lookup table, but this approach sacrifices compressibility and adds a level of indirection.

The basic idea of Frequency Partitioning is to amortize the work of computing code lengths by grouping together tuples that have the same pattern of field code lengths. To achieve this, we partition tuples *coarsely* by the occurrence frequency of their column values, and assign fixed-length codes within each partition.

Figure 1 illustrates this on a two-column table. We start by dividing the distinct values in each column into disjoint *column partitions* – C1a, C1b for column 1 and C2a, C2b, C2c for column 2, according to their occurrence frequencies. Each combination of column partitions (e.g., (C1a,C2b)) forms a partition of the table that we call a *cell.* Blink then creates a separate dictionary of values for each partition of C1 and likewise C2, and assigns them *fixed length* codes. These dictionaries are used to encode the tuples of the table, so every tuple in a given cell is guaranteed to have the same pattern of fieldcode lengths.

In general, say table R has n columns. Let R^i be the domain of possible values for column i, so the domain of R tuples is $\times_i R^i$. Each R^i is partitioned into p_i partitions $R^i_1 \ldots R^i_{p_i}$, with $\cup_{1\leq j\leq p_i} R^i_j = R^i$, such that values with similar frequency are clustered in the same partition.

This partitioning of columns induces a partitioning of an instance of R into cells. Each cell is labeled with a *cell id* $\in \times_i [1, p_i]$. Given a cell with id $(\theta_1, ..., \theta_n)$, column i has only $|R^i_{\theta_i}|$ values, and is given a fixed length code of $\lceil \lg |R^i_{\theta_i}| \rceil$ bits. The tuplecode is $\sum_i \lceil \lg |R^i_{\theta_i}| \rceil$ bits long. The codes are assigned to be *order-preserving*: higher values get higher codes. We use this to apply predicates directly over codes.

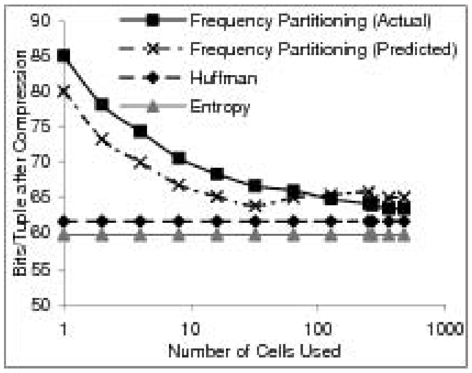
A. Compression Process in Blink

The first step in compressing a table is to analyse the column distributions and accordingly determine the best way to Frequency Partition the table. Next, a separate dictionary is created of values in each partition. Next, these dictionaries are used to assign tuples from the input to the appropriate cells, and to encode and concatenate the fields in each tuple, forming tuplecodes. Finally, the tuplecodes within each cell are sorted and delta-coded as in [12]. We next discuss how the analysis is done, and the compression efficiency of frequency partitioning.

B. Choosing the Partitions

To achieve peak compression, we want column partitionings that minimize the average tuplecode length i.e., we want to assign frequent values to partitions with short code lengths, and rare values to partitions with long codes.

The main constraint in this optimization is the number of resulting cells. As noted earlier, the chief purpose of frequency partitioning is to amortize the work of computing code lengths across all tuples in a cell. If there are too many cells, each cell will have only a few tuples, and this amortization fails. Experimentally we have found it takes about 30,000 tuples/cell



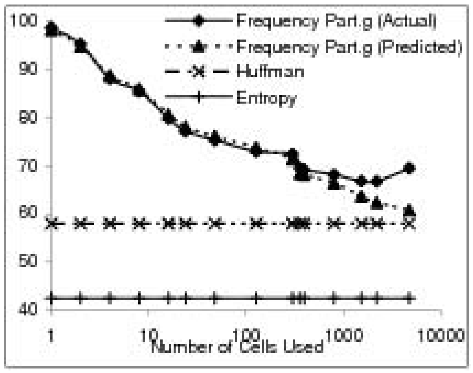


Fig. 2. Compression results for TPCH (left) and Census (right) data.

to amortize per-cell overheads; i.e, we can split a billion-tuple table into ≈ 33,000 cells.

Choosing partitions for One Column

Say that we have decided to allocate p_i partitions to the i'th column. As before, let R^i be the set of distinct values and $R^i_1, \dots R^i_{p_i}$ be the partitioning. The size of codes for R^i_j is $\lceil \lg |R^i_j| \rceil$, so the average code size for column i is

$$\textstyle\sum_{j=1}^{p_i} P(R^i_j)(\lceil \lg |R^i_j| \rceil) \qquad (1)$$

where $P(R^i_j)$ denotes the occurrence frequency of the values in R^i_j (as a fraction of the total).

The average size of a tuple is the sum of average code sizes of each column, minus the savings due to delta coding within a cell. A cell θ with id $(\theta_1, ..., \theta_n)$ has $N \times \Pi_{i=1}^n P(R^i_{\theta_i})$ tuples, where N is the total number of tuples in the table. So, delta coding saves $\lg(N \times \Pi_{i=1}^n P(R^i_{\theta_i})) = \lg N + \sum_{i=1}^n \lg P(R^i_{\theta_i})$ bits from each tuplecode (by Lemma 1 of [12]). The $\lg N$ factor is a constant, but the second factor can be written as a contribution of $\lg P(R^i_{\theta_i})$ bits from column i.

The average value of this contribution (across all cells) is $\sum_{j=1}^{p_i} P(R^i_j) \lg P(R^i_j)$ bits. This must be subtracted from (1) to find the effective size of column i.

Thus, the average size of column i is:

$$\sum_{1 \le j \le p_i} P(R^i_j)(\lceil \lg |R^i_j| \rceil - \lg P(R^i_j))$$

We need to choose a partitioning of R^i that minimizes this objective function.

We start the optimization by forming a frequency histogram of the distinct values in that column, sorted by decreasing frequency. We only consider "interval partitionings" that split these frequency sorted values into contiguous intervals. The reason is that any non-interval partitioning must be sub-optimal, because we can improve compression by swapping two misordered values.

We make two further observations:

• In an optimal partitioning, every interval except the last will have a size that is a power of two. For, in a fixed length code, if an interval has a size that is not power of two, we can improve compression by moving elements into it from the last interval (one with least frequent values).

• Any optimal interval partitioning of values sorted by decreasing frequency will have non-decreasing lengths. For, if we have a partition P of 2^x higher frequency values and a partition Q of 2^y lower frequency values, and $x > y$, we can improve compression by moving the least frequent $2^x - 2^y$ values of P into Q.

These two properties allow Blink to find an optimal partitioning by dynamic programming. We start with 1 partition (the full column). We recursively split the kth partition from the $k-1$ partition at all places in the frequency sorted list of of values that are compatible with the rules above. We then call the optimization step recursively to find the optimal partitioning of the remaining values into $k-1$ partitions.

Choosing partitions across Columns

The next step is to decide how to allocate partitions among the columns, that is, what p_i should we pass into the column partitioning algorithm described above? We do this using a greedy optimization on top of the column level dynamic program. At each step we determine which column could make best use of an extra partition, by computing the optimal partition of each column using one more partition than we are currently using. We then estimate the benefit of this extra partition and the number of additional cells it will cause, using the frequency histogram.

Finally, to produce the optimal partitioning that fits within our cell budget, we apply the greedy optimization step repeatedly until any additional partitioning would cause the cell budget to be exceeded.

C. Compression Efficiency

We now turn to the efficiency of this process: how close the compression comes to entropy. We start with a theoretical analysis of the effect of partitioning.

Suppose the values in a probability distribution $\mathbf{S}$ are partitioned k-way. We denote by $\mathbf{S}|\mathbf{S_1} \dots \mathbf{S}|\mathbf{S_k}$ the resulting (normalized) probability distributions in each partition.

Given a random variable $\mathbf{X}$ (we use bold font for random variables), we denote by $H(\mathbf{X})$ the entropy of $\mathbf{X}$, and by $\{\mathbf{X}\}_m$ the random variable for a multiset of m values chosen i.i.d according to $\mathbf{X}$.

We first give a simple result that partitioning a sequence of values is entropy-neutral (proof by Bayes' rule).

Theorem 1: Given a distribution $\mathbf{S}$ and a partitioning of it into distributions $\mathbf{S}|\mathbf{S_1}$ through $\mathbf{S}|\mathbf{S_k}$,

$H(\mathbf{S}) = \sum_{1 \le i \le k} P(S_i)(H(\mathbf{S}|\mathbf{S_i}) - \lg P(S_i)))$

But the analogous result for multisets is more involved: *Theorem 2:* Suppose a a multiset of m values is chosen i.i.d per $\mathbf{S}$, and partitioned k-way by a partitioning of the domain S into $S_1, \ldots S_k$. Suppose the partitions of the multiset have sizes $\overrightarrow{\mathbf{N}} = (\mathbf{N_1} \ldots \mathbf{N_k})$. We have, $H(\{\mathbf{S}\}_m) =$

$$H(\overrightarrow{\mathbf{N}}) + \sum_{\overrightarrow{n} | \sum n_i = m} Pr(\overrightarrow{\mathbf{N}} = \overrightarrow{n}) \sum_{i=1}^{k} (H(\{\mathbf{S}|\mathbf{S_i}\}_{n_i}))$$

Proof: In the appendix.

This theorem implies that if each cell resulting from a partitioning is compressed to entropy, then the process of partitioning and then compressing cells compresses the overall multiset close to entropy.

Blink does not compress each cell to entropy because it uses fixed-length codes within a cell. If we partition at a fine enough granularity (i.e. used enough cells), we can make skew within a cell negligible, and fixed-length coding would be sufficient. But the number of cells we can use is restricted, as discussed above. We next study the impact of this tradeoff on two data sets.

The first is an uncorrelated but skewed version of the TPCH dataset. Similar to the data generator we used in our previous work [12], we modified the generator to skew certain columns. We chose 99% of dates to be in 1995-2005, with 99% of that on weekdays, 40% of that on two weeks each around Christmas and Mothers Day. We chose nation distributions from WTO statistics on international trade. We also denormalized the data by joining lineitem, order, customer, and nation.

The second is a vertical partition containing the first 38 columns extracted from the 1.6 million tuple 2000 U.S. census dataset for California from census.gov.

Figure 2 plots the number of bits per tuple as we increase the number of cells used. For each dataset we show: (i) The empirical entropy of the data treated as a set. (ii) The result of Huffman coding each column, concatenating the codes to form a tuplecode, and then sorting and delta coding tuplecodes as in [12]. (iii) Average number of bits per tuple after frequency partitioning, as estimated during optimization. Note that when only one cell is used, this is equivalent to the fixed length coding followed by delta coding. As the number of cells increase, better compression is achieved. (iv) The actual compression ratio determined by looking at the size of our compressed files, including headers and cell level overhead.

Observe that in both cases, partitioning into a few 1000s of cells gets nearly the same compression as Huffman codes[1]. Also, notice that using more cells generally improves compression, though at the very end of the census data set, compression stops increasing due to the overhead caused by too many cells in use.

IV. Query Runtime

The core of Blink's query runtime is the Generalized Scan, a module that combines the functionality of scan, selection, grouping and aggregation. Figure 3 gives a high-level overview. The scan maintains a work queue, where each entry is a block of compressed tuples from a single cell of the input. A pool of worker threads consumes these units of work and produces partial query results, which are merged at the end to produce a complete query result.

Each thread starts by finding the code lengths of each field, and the dictionary used for coding each field – this is done once per cell. Then, it loops through the tuples in the cell, doing 3 steps: *Undo Delta Coding*, *Selection*, *Grouped Aggregation*.

Undoing the delta coding is easy. The first tuplecode in a cell is stored as-is, and subsequent tuplecodes are formed by adding deltas to the previous tuplecode, as in [12]. The next two steps are more involved.

A. Selection by Operation Folding

The standard way to apply a conjunction of predicates on a tuple is as follows:

```
for each field f referenced in the where
clause do
   extract tuple.f
   apply the predicate over tuple.f
   if it fails return false
   else continue
```

This is clearly not a constant-time operation: the cost varies with the selectivity and the number of conjuncts. Further, the conditional evaluation is expensive because each mis-predicted branch breaks the instruction pipeline, and costs about 30-40 cycles in our measurement. As the number of conjuncts increases, predicate evaluation becomes the dominant cost of the scan.

Blink avoids this variability and the conditional by leveraging frequency partitioning. Recall that within a cell, each field has a single code length, and hence occurs at a fixed bit-offset within the tuplecode. We use this to evaluate all equality and range predicates *in parallel*, in constant number of operations (independent of the number of predicates), as we see next.

Parallel Equality and Range Predicates

Most modern processors have a bank of 128 bit registers, on which one can do bitwise operations (AND, OR) as well as "Single-Instruction Multiple-Data," (SIMD) versions of arithmetic operations. After compression, tuplecodes generally fit in a small number of these registers, mostly 1 and never more than 2 in the cases we have seen. Hereafter, for ease of presentation, we will assume that the tuplecode is in a single register. For wider tables, the operations we apply have to be simulated by separate operations on individual registers of the tuplecode.

Evaluating a conjunction of equality predicates on a register-resident tuplecode is easy. First, extract the needed fields by applying a suitable mask (a bitwise AND). Then, compare the result against a second mask containing the expected fieldcodes at the appropriate field offsets. Note that these masks are computed exactly once per cell.

[1]Huffman still loses up to 1 bit/field against entropy because we have to round up fractional bits on each field. This is more noticeable for the wide census table.

Range predicates are more difficult to evaluate in parallel, since they cannot be converted to a single comparison.

Consider a conjunction of $col > literal$ predicates. We could extract the needed fields via an AND and then subtract a mask containing the literals at the right offsets. If the result is individually positive on every field (checked via a mask on the high order bits), the tuple passed all the predicates. Unfortunately, this naive approach does not work. When the subtraction causes a field to go negative, the processor *borrows from the higher-order bits of the register, thereby changing the values of other fields*.

Trapping the borrows

Our solution is place a *borrow-stopper* 1-bit to the left of each field involved in the predicate. To make space for these borrow-stoppers, we process the odd-numbered fields separately from the even-numbered fields. When processing the even fields, we use the space of the odd fields to trap the borrows, and vice-versa. We leave the leftmost bit vacant to trap borrows on the leftmost field.

Take a predicate $c_1 \leq l_1$ and $c_2 \leq l_2 \cdots c_n \leq l_n$ where c_i are *odd-numbered* columns and and l_i are literals. At the start of a cell of tuples, we convert the literals l_i into codes L_i using the dictionary. Now, suppose that the start and end offsets of the fieldcodes for c_i are $[b_1, e_1) \ldots [b_n, e_n)$. We compute 4 masks (once per cell):

// Mask to extract needed fields
M1 $\leftarrow 0^{b_1} \quad 1^{e_1-b_1}\ 0^{b_2-e_1} \quad 1^{e_2-b_2} \ldots$
// Mask with literals (L_i's) and borrow-stoppers (**1**s)
M2 $\leftarrow 0^{b_1-1}\ \underline{\mathbf{1}}L_1\ 0^{b_2-e_1-1}\ \underline{\mathbf{1}}\ L_2 \ldots$
// Mask to extract borrow-stopper bits
M3 $\leftarrow 0^{b_1-1}\ \underline{\mathbf{1}}0^{e_1-b_1}\ 0^{b_2-e_1-1}\ \underline{\mathbf{1}}\ 0^{e_2-b_2} \ldots$
// Mask with expected results if all predicates pass
M4 $\leftarrow 0^{b_1-1}\ \underline{\mathbf{1}}0^{e_1-b_1}\ 0^{b_2-e_1-1}\ \underline{\mathbf{1}}\ 0^{e_2-b_2} \ldots$

Then, on each tuple t, we do this test:

$$((M2 - (t\&M1))\&M3) == M4$$

We do a similar test (with different masks) for predicates on even-numbered fields. [2]

To see why this works, notice that the bit immediately to the left of each field is $\underline{\mathbf{1}}$ in M2 and 0 in $(t\&M1)$. So, $M2 - (t\&M1)$ has $(1 < L_i > - 0 < c_i >)$ in the offsets $[b_i - 1, e_i)$. If $L_i \geq c_i$, this has a 1 at bit $b_i - 1$, and 0 otherwise. This is tested by comparison with $M4$.

To evaluate a mix of $>$ and $\leq$ predicates, we invert the corresponding borrow-stopper bits in $M4$: e.g., for $c_1 \leq L_1$ and $c_2 > L_2$, we set M4 to $0^{b_1-1}\underline{\mathbf{1}}0^{e_1-b_1}\ 0^{b_2-e_1-1}\ \underline{\mathbf{0}}\ 0^{e_2-b_2} \ldots$

Evaluation of Other Predicates

For predicates other than equality and range, the standard solution has been to decode and then apply the predicate. Blink uses an alternative implementation that involves no decoding. Consider for example a LIKE predicate on column C. The idea is to consider the dictionary on C as an implicit dimension table. At the beginning of the query. we evaluate the predicate over the dictionary, identify the matching codes and place them in a hash table. During the scan, the LIKE predicate is evaluated by hashing on the tuplecode after masking out all columns except C.

[2]One optimization that avoids this second test is to place a vacant "sacrificial" bit before every field: this loses 1 bit per field in compression but doubles the predicate evaluation speed.

B. Grouping and Aggregation Stage

The last step is to apply group-by, by updating a suitable running aggregate for each tuple that passes all predicates. The aggregate value(s) in the current tuple are formed by extracting and decoding the fieldcode(s) on aggregation fields. The fieldcodes for all group-by columns are extracted and concatenated (by ANDs and shifts) into a packed *groupcode*.

Informally, the running aggregates are maintained in a hash table, so the update involves:
aggTable$[hash(\text{group})] += $ aggregate value(s) **(1)**
for a suitably defined $+=$ operator.

There are two challenges in doing this efficiently:

- We want to keep the hash table concise so that it fits in the L2 cache
- To do a fast hash table lookup, we want to avoid the random access and conditional branching involved in the usual open-chaining based hash tables.

In Blink we use three hashing techniques, which are applicable in different situations depending on the number of distinct groups. Before discussing these, we need to decide at what granularity to group.

Drawers as Granularity of Grouping

Suppose that a query groups on columns G, H, which have column partitions G1, G2, H1, H2. The table might have numerous cells, from the cross-product of partitions on every column (not just G, H). There are then three granularities at which we can do grouping.

First, we can maintain a separate $aggTable$ per cell. This results in a small aggTable because there will be few distinct groups per cell. But the per cell hash tables need to be combined at the end to get the final result. Second, we can maintain a single $aggTable$ for the whole query. However, this results in a very large hash table (as many distinct groups as there are in the whole table), which may not fit in L2 cache. This approach is also hard to parallelize because all threads would have to synchronize on the hash table.

We choose a third option that lies between these extremes, called a *drawer*, which is defined by the partitioning along the group-by columns:

A *drawer* is a collection of cells that come from a single partition of the group-by columns.

Figure 4 shows a table that has been frequency partitioned on columns A, B, and C. The dashed cuboid shows a drawer for a query that groups on A, B. Note that unlike cells, drawers are defined for a particular query.

Drawers have two important properties which make them the right granularity to compute aggTable's on:

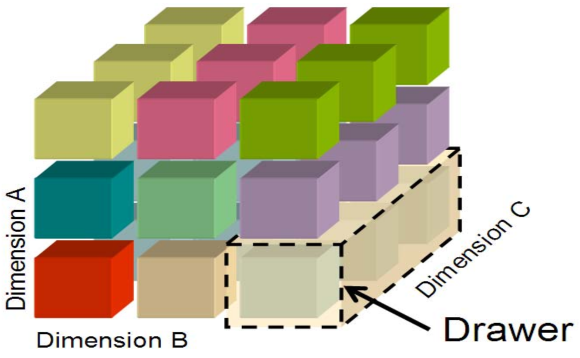


Fig. 4. Relationship between drawers and cells.

1. For all cells within a drawer, the fieldcodes for each grouping column are drawn from a single dictionary. Since dictionaries are 1-to-1 maps of values to fieldcodes, this means we can replace Equation (1) with:
$$\text{aggTable}[hash(\text{groupcode})] += \text{aggregate} \quad \textbf{(2)}$$
so we need not decode the groupcode at this point.
2. Each group can occur only in a single drawer. So the aggTable can be drawer specific, and only needs to hold as many entries as there are groups in a single drawer – this is often much smaller than the total number of distinct groups in the table.

We independently compute aggTable's for each drawer. At the end, we union the lists of $(groupcode, aggregate)$ from each drawer – since each group occurs in only one drawer, this is a trivial operation. We then apply any HAVING clause, and only then decode the groupcodes.

Different drawers can have widely different numbers of groups because we partition columns by frequency, as discussed in Section III. Since we use a separate aggTable for each drawer, we independently choose the hashing technique used for each drawer, according to its number of distinct groups, as estimated from the dictionary.

Implementing Grouping

Blink uses one of three kinds of hash tables to maintain running aggregates, based on the estimated number of distinct groups in the drawer:

GroupCode as Hash (IDX):

The first one is to use the groupcode itself as a hash function to index into a hash table, referred to as **IDX**. This hash function is trivial to compute. Further, since groupcodes are unique for each group, this hash function is guaranteed to be *a perfect hash* – *i.e.*, have no collisions. So the running aggregate can be accessed in a single lookup. For grouping on a single column, compression implies that the groupcode is very close to *a minimal perfect hash* – the smallest possible hash table that will handle all the distinct groups. So Blink uses IDX for all single-column group-bys.

Explicit Minimal Perfect Hash (MPH):

Unfortunately, the groupcode need not be dense for multi-column group-bys, because of correlation. For example, a sales table with 1000 customers and 1000 stores could encode CustId and StoreId in 10 bits each. But the number of distinct (CustId,StoreId) pairs can be $\ll 10^6$, and it is cache-wasteful to use a 10^6 size hash table. Blink handles such correlated group-bys using a pre-computed minimal perfect hash function (**MPH**). This MPH is constructed during ETL, when all the groupcodes are seen (if new groupcodes arise in incremental updates between loads, they are placed in a separate drawer that is handled by linear probing which is described below).

We do not know the group-by columns in advance, so we automatically construct MPHs on all column-pairs that have sufficient correlation (this is currently specified by hand). We generate one perfect hash function for each drawer for each chosen pair of group-by columns.

Our perfect hash function is based on Jenkins's perfect hash [2]. This hash function has exactly as many buckets as the number of groups. But, the hash function needs an extra auxiliary table, which usually has half the buckets of the hash table. Thus, for correlated columns, an MPH has many fewer buckets than does IDX, but each bucket is about twice as large. A second limitation of an MPH is that it needs two random lookups: one into the auxiliary table and another into the actual aggregate table.

Linear Probing (LPB):

The last hash function that Blink uses is open addressing with linear probing (LPB). We choose open addressing over a chained hash table, as the linked list used in chained hash table has poor cache performance. We use multiplicative hashing as the hash function. Although the hash function can be computed efficiently, LPB has collisions, which results in branches and associated branch mis-prediction penalty. Space-wise, LPB is quite expensive because it needs to store both groupcodes and aggregates. Moreover, the hash table can only be filled up to a load factor which must be around 60%, otherwise we will have too many collisions [6].

We conducted a micro-benchmark to study the applicability of these three hash methods. Figure 5 plots the aggregation time for a simple SUM with GROUP BY query using each method, as a function of the number of groups in the drawer. Groupcodes are assigned to tuples at random, with each groupcode being equally likely. For IDX hash function, we examine two cases. The first (IDX_dense) is when the groupcode happens to be dense, which we define as a groupcode domain that is only twice as large as the actual number of groups. The second (IDX_sparse) is when the groups are sparse, where we set the groupcode domain to be 1024 times of the number of groups. For LPB hash table, we study two load factors: 0.7 (LPB_0.7) and 0.5 (LPB_0.5).

As expected, when the groupcode is dense, IDX is the best solution, because it has the smallest aggregation hash table, and requires only one lookup into it. In the sparse case, the aggregation time of MPH grows slower than of IDX_sparse because it has a smaller hash table. MPH outperforms IDX_sparse when there are more than 32,000 groups. Beyond about 256,000 groups the run time of IDX_sparse

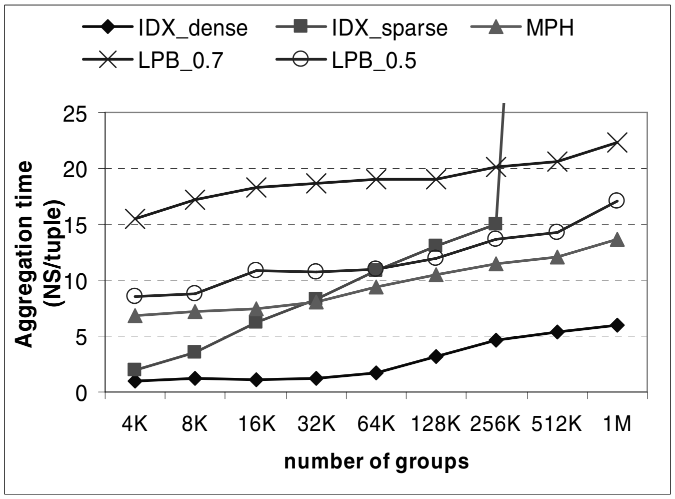


Fig. 5. Hash Method Comparison

explodes because it starts to thrash wildly, as its hash table (whose size is 1024 times the number of groups) starts to page out of memory.

These numbers suggest that one of IDX and MPH always dominates LPB: IDX for small numbers of groups and MPH for large numbers of groups. So we use LPB in Blink only as a fall-back, for correlated multi-column group-bys where we have not constructed an MPH. In such cases, an IDX hash table would spill out of cache, whereas an LPB hash table can fit in cache because it is sized by the number of distinct groups.

V. Experiments

We have built a complete prototype of Blink, implementing all the features described in this paper. Blink compresses data by frequency partitioning and delta-coding within cells. Blink keeps this data in memory for running queries. Blink supports single-block SQL queries with equality, range, and in-list predicates, SUM and COUNT aggregates, and grouping.

We now present an experimental evaluation of Blink. Our goal is two fold: (a) how close is Blink to a constant-time query processor? (b) how far do each of our techniques go to making the scan efficient?

Our dataset is a universal relation formed by denormalizing a variant of the TPC-H schema, as proposed by O'Neil *et al.*[10]. We chose a vertical partition with part key, revenue, order quantity, lineitem price, week of year, month, supplier nation, customer nation, supplier and customer region, discount, category, brand, year, and day of week for our experiments. We populated this table with 200M rows using the same skewed distribution as in Section 3 (the denormalization preserves the cardinality of the fact table). All experiments were conducted on a server with 8 GB of main memory and two 4-core Xeon processors running at 2.66 GHz.

A. Variability in Query Response Time

Our first experiment studies the extent of variation in query response times. We divide response times by table size and report all times in ns/tuple; low variation here translates to constant time by scaling the number of nodes with the data size. We ran a suite of 150 queries with the following template:

select sum(revenue) **from** denormalized **table**
where <conjunction of predicates>
group by <**column** list>

The queries differ in the predicates used and the columns grouped by. The predicates are generated as conjunctions with a randomly chosen number of conjuncts, between 0 and 7. Each conjunct is of the form $(c < D_h)$ or $(c \leq D_h)$ or $(c > D_l)$ or $(c \geq D_l)$ where column c and the comparator are chosen at random, and $[D_l, D_h]$ is the domain of values for column c – *e.g.*, $year \geq 1994$, or $partkey > 1$. The idea is that these queries all have nearly 100% selectivity, and thus force every tuple to be scanned. Otherwise, frequency partitioning has an implicit data reduction effect where entire cells can be eliminated from the scan because no value in its dictionary matched the predicate – we evaluate this effect later in this section.

The group-by columns were chosen at random from among the non-measure columns, with queries grouping by one or two columns.

The first two graphs of Figure 6 plot the query speeds of every query whose group-by produced up to 20000 groups. We use 8 threads, to match the number of cores on our server. Notice that there is little variability in the query response time, when plotted against either number of groups in the result or against number of predicates in the where clause. All the queries run at between 3.1 and 4.5 ns per tuple. This tight spread ($< 50\%$) shows the predictable nature of scans and the ability of our parallel predicate evaluation and hash grouping to mask variations across queries.

The last plot of Figure 6 shows speeds for queries that produce > 20000 groups. The timing degrades rapidly, touching 18 ns/tuple at 51000 groups. This degradation arises because the hash table for storing running aggregates starts to spill outside of the L2-cache. We believe such queries are rare. Further, our dataset had limited skew and so every group was updated almost equally often. We expect in realistic datasets that some groups will be occur much more often than others, and hence benefit from cache-locality. Still, performance of queries that produce more groups than can fit on the cache of a single core is an important challenge for future research.

1) Variability with Query Selectivity: Our next experiment concerns the performance of queries that have selective predicates. These queries follow the same template as before, except that the literals in the predicates are now set so that selectivity varies. As mentioned, this can have a filtering effect at cell level, especially with equality predicates: if a complete cell has no value matching a literal, the dictionary is unable to convert the literal into a code, and so the cell is skipped during the scan.

Figure 7 plots the running times of selective queries. The x-axis is the cell-selectivity: the number of cells that need to

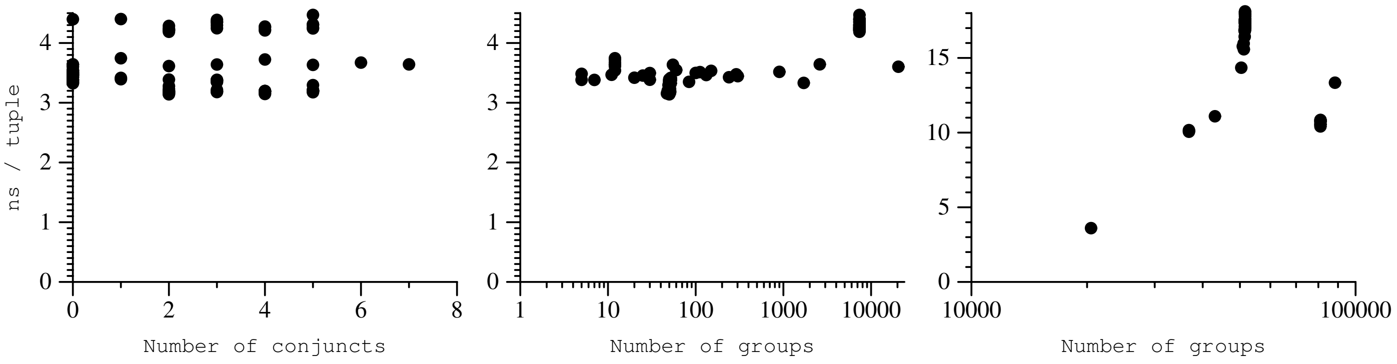


Fig. 6. Scan speed for arbitrary single-block queries. Predicates are crafted to have ≈ 100% selectivity.

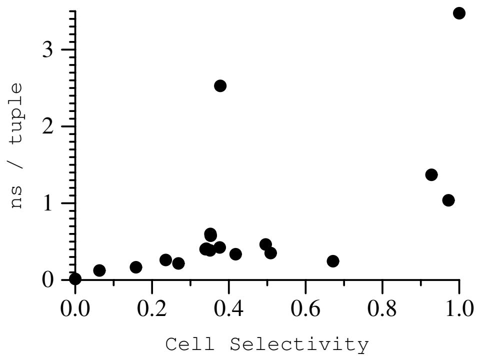


Fig. 7. Scan speed for selective queries

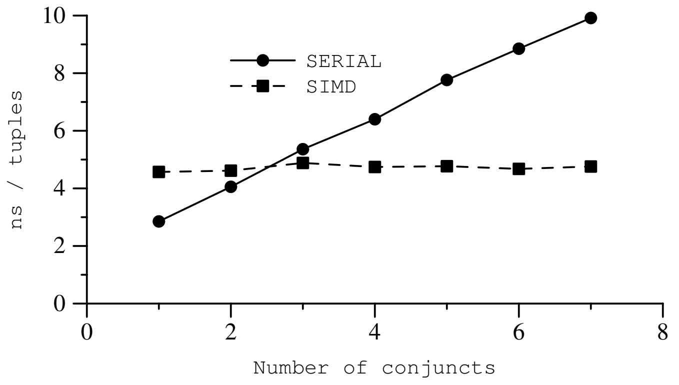


Fig. 8. Value of parallel predicate evaluation

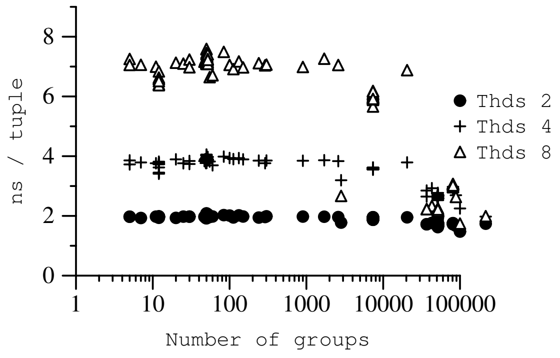


Fig. 9. Multicore speedup

be scanned. Observe that while all queries still run under 4.5 ns/tuple, there is a definite pattern where the speed improves as selectivity decreases.

We believe this behavior is the right behavior for a constant time system. A designer should size the system so that the poorly selective queries will still finish in the required response time, and the faster running of the selective queries only contributes to improved system throughput.

We point out that this cell-filtering is an incidental benefit of frequency partitioning. In contrast to multi-dimensional clustered indexes (eg, [1]), frequency partitions are chosen to optimize for compression, not for filtering. In particular, the choice of partitions is made without care for the workload.

B. Savings from SIMD predicates

Having seen the overall performance of Blink, we turn to its individual components. In Section IV-B we saw an experiment that highlighted the value of IDX and perfect-hash based hash tables, and of picking different hash tables for different drawers. Now we study how much benefit we derive from parallel predicate evaluation.

We run seven versions of a single query, gradually removing its conjuncts one-by-one:

select sum(revenue) **from** denormalized **table**
where QTY $<$ 50 **and month** $\leq$
11 **and** l_price $>$ 1 **and year** $\geq$1995 **and** l_partkey $\geq$
1 **and** revenue $\geq$1.0 **and** week $\geq$1
group by month

Figure 8 plots the query response time vs the number of conjuncts, for two implementations. The first (SIMD) is the standard Blink implementation which evaluates all equality and range predicates in parallel. The second (SERIAL) is a typical implementation which evaluates the predicates in a for-loop. Notice that the response time for the parallel implementation is almost constant, while the serial performance is linear in the number of conjuncts.

C. Scalability of Our System

Our last experiment studies the multi-core scalability of Blink, using our original suite of 150 queries. Figure 9 plots the speedup obtained in going from 1 to 2, 4, and 8 threads. Observe that speedups for 2 and 4 threads are close to ideal, and most queries get 7x speedup with 8 threads. As the number of groups increases beyond around 20000 this scaling drops, because the hash-table spills out of cache. All cores share the memory bus and the cost of random memory accesses to update running aggregates starts to dominate. This suggests that scans can scale with cores, provided the number of groups is moderate.

VI. Conclusion

We have made a case for *constant-time query processing*, and argued that table scans are a promising way to achieve this. We have described Blink, a system that partitions data by frequency so as to achieve good compression while maintaining long runs of fixed length codes. Blink also has new techniques for SIMD evaluation of conjunctive equality and range predicates, and for hash-based aggregation. Experimental results suggest that Blink is both efficient and delivers consistent response times. Blink also gets near-linear speedup on multicore architectures.

Many challenges remain as future work. The most important is to make Blink a full-fledged database system; in particular, to efficiently handle updates against this compressed data format. Many aspects of query functionality are also open, such as joins that cannot be addressed via denormalization, sub-queries, and disjunctions. We also plan to compare Blink and its row-wise SIMD processing against column stores.

Acknowledgements: We wish to thank Ryan Johnson for many contributions to the design and implementation of Blink.

References

[1] B.Bhattacharjee, S. Padmanabhan, T. Malkemus, T. Lai, L. Cranston, and M. Huras. Efficient Query Processing for Multi-Dimensionally Clustered Tables in DB2. In *VLDB*, 2003.
[2] Minimal Perfect Hashing . http://www.burtleburtle.net/bob/hash/perfect.html.
[3] Z. Czech, G. Havas, and B. Majewski. An Optimal Algorithm for Generating Minimal Perfect Hash Functions. *IPL*, 43(5), 1992.
[4] A. Holloway, V. Raman, G. Swart, and D. DeWitt. How to Barter Bits for Chronons: Compression and Bandwidth Tradeoffs for Database Scans. In *SIGMOD*, 2007.
[5] D. Huffman. A method for construction of minimum-redundancy codes. In *Proceedings of I.R.E.*, 1952.
[6] D. Knuth. In *The Art of Computer Programming, v3*, 1973.
[7] H. F. Korth, G. M. Kuper, J. Feigenbaum, A. van Gelder, and J. D. Ullman. SYSTEM/U: a database system based on the universal relation assumption. *TODS*, 9(3), 1984.
[8] T. Legler, W. Lehner, and A. Ross. Data Mining with the SAP Netweaver BI Accelerator. In *VLDB*, 2006.
[9] R. MacNicol and B. French. Sybase IQ Multiplex - Designed for analytics. In *VLDB*, 2004.
[10] P. O'Neil, B. O'Neil, and X. Chen. The Star Schema Benchmark (SSB) (Preprint), January 2007. http://www.cs.umb.edu/p̃oneil/StarSchemaB.PDF.
[11] R. Pagh and F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2), 2004.
[12] V. Raman and G. Swart. Entropy Compression of Relations and Querying of Compressed Relations. In *VLDB*, 2006.
[13] K. Ross. Efficient Hash Probes on Modern Processors. In *ICDE*, 2007.
[14] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-Store: a column-oriented DBMS. In *VLDB*, 2005.
[15] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The Implementation and Performance of Compressed Databases. *SIGMOD Record*, 29(3), 2000.
[16] M. Zukowski, S. Heman, and P. Boncz. Architecture-conscious hashing. In *DaMoN*, page 6, 2006.
[17] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE*, 2006.

Appendix

We now show that partitioning is entropy-neutral if we do not consider the order of the tuples to be important. Intuitively, splitting an unordered set of tuples into any number of smaller sets does not change the space needed to represent the tuples.

More formally, a *partitioning* of a probability distribution $\mathbf{S}$ (we use bold font for random variables and probability distributions) with probability function p is a set of distributions induced by a partitioning of the domain of S into sets S_1 through S_k, where the probability function of each S_i is given by $p_i(s) = p(s)/P(S_i)\ \forall s \in S_i$, where $P(S_i) = \sum_{u \in S_i} p(u)$.

We denote these induced distributions as $\mathbf{S|S_1}, \mathbf{S|S_2}, \ldots \mathbf{S|S_k}$, because $\mathbf{S|S_i}$ is the probability distribution of S values conditioned on them being in the partition S_i.

As in [12], we model a table with m tuples as a multiset of m values chosen i.i.d per a probability distribution $\mathbf{S}$. We use the notation $\{\mathbf{S}\}_m$ to denote the random variable corresponding to this multiset. We now use the partitioning on $\mathbf{S}$ to induce a partitioning of the multiset $\{\mathbf{S}\}_m$ into k separate multisets of sizes $\overrightarrow{\mathbf{N}} = (\mathbf{N_1} \ldots \mathbf{N_k})$, one for each partition. We want to show that the entropy of the original multiset $H(\{\mathbf{S}\}_m)$ is the same as the sum of entropies of the k multisets: $H(\{\mathbf{S|S_i}\}_{\mathbf{N_i}})$. A tricky part of this statement is that the multisets are parameterized by the sizes $\mathbf{N_i}$, which are themselves random variables. So we make this a theorem about expectation.

Theorem 2: If a multiset of m values chosen i.i.d from probability distribution $\mathbf{S}$ is partitioned into k multisets of sizes $\overrightarrow{\mathbf{N}} = (N_1 \ldots N_k)$ by a partitioning of the domain S into $S_1 \ldots S_k$, then $H(\{\mathbf{S}\}_m) =$
$$H(\overrightarrow{\mathbf{N}}) + \sum_{\overrightarrow{n} \mid \sum n_i = m} Pr(\overrightarrow{\mathbf{N}} = \overrightarrow{n}) \sum_{i=1}^{k} (H(\{\mathbf{S|S_i}\}_{n_i}))$$

Proof: In the LHS, observe that each $\{\mathbf{S|S_i}\}_{n_i}$ refers to the random variable for a multiset of values chosen from the set S_i. But this multiset was formed by choosing values i.i.d from the set S. So, the random variables: $\{\mathbf{S|S_1}\}_{n_1}, \{\mathbf{S|S_2}\}_{n_2} \ldots \{\mathbf{S|S_k}\}_{n_k}$, are all mutually independent. So,
$\sum_{i=1}^{k} (H(\{\mathbf{S|S_i}\}_{n_i})) = H(\{\mathbf{S|S_1}\}_{n_1}, \{\mathbf{S|S_2}\}_{n_2} \ldots \{\mathbf{S|S_k}\}_{n_k}\})$.
This is the joint entropy of k multisets: one of size n_1 chosen from S_1, one of size n_2 chosen from S_2, and so on. The sum in the L.H.S. of the theorem is over all vectors $\overrightarrow{n} = (n_1, \ldots n_k)$ such that $\sum_{i=1}^{k} n_i = m$. Since the sets S_i form a partition of S, there is a one-to-one map from these k multisets to a multiset of size m from S: *i.e.*, there is a one-to-one map from each value of $(\{\mathbf{S|S_1}\}_{n_1}, \{\mathbf{S|S_2}\}_{n_2} \{\mathbf{S|S_k}\}_{n_k})$ to a value of $\{\mathbf{S}\}_m$. The theorem now follows by applying Bayes' rule. □.