Bringing BLINK Closer to the Full Power of SQL

Knut Stolze* Vijayshankar Raman† Richard Sidle† Oliver Draese*

Abstract: BLINK is a prototype of an in-memory based query processor that exploits heavily the underlying CPU infrastructure. It is very sensitive to the processor's caches and instruction set. In this paper, we describe how to close two major functional gaps in BLINK, which arise from real-world workloads. The manipulation of the data maintained by BLINK require specialized data structures. Another aspect being discussed are NULLs and how to avoid the three-valued logic during query processing.

1 Introduction

BLINK [RSQ+08] is a query processor that was designed to answer all queries in constant or near-constant time with respect to query complexity. This goal is accomplished by always running a highly efficient table scan over all the data. The data is stored denormalized in main memory so that no I/O operations can disrupt the query execution time. The data is highly compressed to provide for more raw data being stored in main memory and to reduce the amount of data flowing from memory to CPU during scans. At the same time, the compression scheme results in long runs of fixed-length values, which can be efficiently scanned with the CPU's vector operations without having to decode the data. A BLINK single system in a cluster shall store and process about 100 GB of uncompressed data in main memory.

A prototype demonstrates the feasibility and proves the constant query execution time. This has been achieved without requiring a performance tuning layer on which traditional DBMSs rely heavily. No indexes, materialized views, or a wide variety of tuning and configurations options are necessary. Still, BLINK has shown to improve the query execution time by one or more orders of magnitude compared to highly tuned (or even self-tuning) traditional relational database systems like MonetDB [BMK99].

In this paper, we present our ongoing efforts to extend the functionality of BLINK. First, we recapitulate in section 2 the primary techniques that were used in the query processor and highlight the current gaps in the functionality of the query engine. This serves as a motivation for the subsequent section 3, where we describe how data is organized physically in the main-memory structures. Those data structures need to be tailored to the envisioned usage of BLINK as a full function query engine for data warehousing that can also handle data modifications efficiently. Today's RDBMS all support SQL NULLs to represent unknown or not-applicable values. Section 4 describes our mechanisms how NULLs are represented in BLINK while still maintaining the highly efficient query processing.

*IBM Germany Research & Development, Böblingen, Germany
†IBM Almaden Research Center, San Jose, CA, USA

2 Background on BLINK

BLINK stores all data in main memory to avoid any I/O overhead. Since main memory is not in abundant supply as secondary or ternary storage (i. e. disk or tape), it is necessary to compress the data in memory.

We built BLINK on top of previous work on entropy compression of relations [RS06]. Values are encoded with variable length codes (shorter codes for more frequent values). Query-time scans have to parse each code in each tuple individually because the beginning of the next code depends on the length of the previous codes. The cost can be significant, especially because denormalized tables tend to have many columns. This overhead is a well-known problem, pointed out in [WKHM00, HRSD07]. For the same reason, most of other related work on fast querying over compressed data relies on fixed length codes. For example, Sybase IQ [MF04] and C-Store [SAB+05] use fixed length codes and leverage the homogeneity to do fast array-based operations.

2.1 Frequency Partitioning

BLINK uses a compression technique called *Frequency Partitioning* that still compresses close to entropy but produces long runs of fixed-length codes for efficient scanning. Our compression scheme amortizes the work of computing code lengths by grouping together tuples that have the same pattern of fieldcode lengths. To achieve this, we partition tuples coarsely by the occurrence frequency of their column values, and assign fixed-length codes within each partition. The encoding is dictionary-based. An encoded value is called *fieldcode*. The fieldcodes of a tuple are concatenated and form the *tuplecode*.

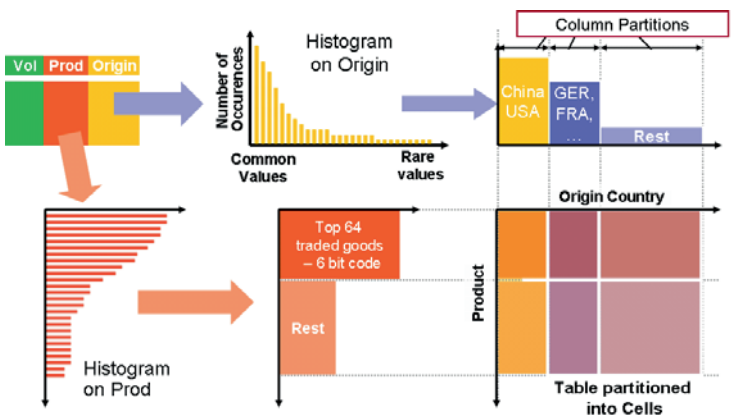

Figure 1: Illustration of Frequency Partitioning

Figure 1 illustrates this on a three-column table where the columns "Prod" and "Origin" are partitioned. We start by building separate histograms for both columns. Each his-

togram collects the occurrence frequencies over all distinct values. We use the histograms to form disjoint *column partitions* – C1a, C1b for column "Prod" and C2a, C2b, C2c for column "Origin", – according to their occurrence frequencies. Each combination of column partitions, e. g. (C1a, C2b), forms a partition of the table, called a *cell*. Separate dictionary of values for each partition of "Prod" and "Origin" are created and fixed length codes assigned. The 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 and, thus, the same tuplecode length.

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 $R^i_1 \dots R^i_{p_i}$, with $\cup_{1 \le j \le p_i} R^i_j = R^i$, such that values with similar frequency are clustered in the same partition. This partitioning of columns induces a partitioning of R into cells. Each cell is labeled with an $id \in \times_i [1, p_i]$. Given a cell with id $(\theta_1, \dots, \theta_n)$, 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.

2.2 SIMD Predicate Evaluation

At query time, it is necessary to scan all the data in BLINK and apply predicates to identify only matching tuples. Since BLINK uses order-preserving codes, we can apply equality and range predicates directly on the encoded data without any decoding. We evaluate all conjunctive equality and range predicates in parallel with a constant number of processor instructions, eliminating loops and branches. Due to the fixed-length tuplecodes in each cell, it is possible to apply bit masks on the tuples to extract only the fieldcodes to which a predicate is applied. The CPU's vector operations (single instruction multiple data (SIMD)) are exploited to operate on multiple tuplecodes in parallel [JRSS08].

2.3 Gaps in BLINK

The implementation of BLINK was designed to work on a single system only. Scaling it up beyond the system limits requires grid or cluster technologies to distribute data and workload across the then-available machines and to coordinate between those. Additionally, query processing has to be distributed as well. Each system produces only a partial result, and this has to be combined before further processing is applied on it. For example, a SQL query using an `ORDER BY` or `HAVING` clause can only be computed on the combined result set. The distribution of queries processing in a cluster is not further discussed.

A second gap in BLINK is that the system was tailored to query processing only. The aspect of maintaining the data has not yet been further investigated. Section 3 dives into the details of the data structures used to efficiently add or remove tuples to or from tables scanned by the query engine.

Thirdly, the data used in the performance measurements presented in [RSQ+08] did not include SQL NULLs. However, NULLs are often found in data warehouses, and many compression techniques actually take advantage of that. *Sparse* data cubes contain many NULLs in the cells of the cube, i. e. the values in the fully denormalized schema. Therefore, we explain in section 4 in more detail how NULLs will be handled in a very efficient manner in BLINK.

3 Data Structures for Data Modifications

A query processor supporting data modifications, i. e. `INSERT`, `UPDATE`, and `DELETE` operations has to ensure the integrity and consistency of the data. BLINK, being positioned for data warehouse applications, faces different requirements than OLTP systems. Data modifications do not trickle in. Data warehouses use batch ETL (extract, transform, load) that collected data changes from source systems, transform it (including aggregation and cleansing), and finally load the data. Thus, non-concurrent mass operations to roll-in/out batches of tuples are the sweet spot that need to be handled efficiently by BLINK as well.

3.1 Snapshot Semantics

A quick design decision was to avoid a lock manager – instead, snapshot semantics are used. Each tuple has an associated validity interval, stating when the tuple was added and when it was finally deleted. A reorganization can be run to eventually clean up deleted tuples at some point. The validity interval could be timestamp based, which is rather expensive storage-wise. A simple counting mechanism (called *epoch*) is sufficient. Each time a data modification is applied to the data managed by BLINK, this counter is incremented. Thus, a (denormalized) table R is extended by two additional columns e_s and e_e, where e_s indicates the start epoch and e_e the end epoch of the validity interval $[e_s, e_e)$.

When a data modification are initiated, the current epoch counter ce for the system is determined and used for all processed tuples. For all new tuples, (e_s, e_e) is set to $(ce+1, \infty)$. A tuple, thus prepared, is simply appended to the table. Deleting tuples is implemented with a specialized query, which does not only scan the data and returns the tuples found. It also modifies matching tuples and sets e_e to ce so that queries starting after the update operation completed exclude those tuples. The global current epoch counter is incremented at the end of the modification processing, at which point new queries will pick up the changed value for ce and the modifications become visible.

Insertion and deletions can be done concurrently and lock-less to other queries. Again, ce is determined for each query, and the query is augmented with two additional predicates:

$$\ldots \wedge e_s \leq ce \leq e_e$$

The predicate on e_s excludes all tuples that may currently be inserted since e_s is larger

than the query's ce for those tuples. Likewise, the predicate on e_e excludes all tuples that were deleted before the query started while still retaining the tuples that are currently in the process of being deleted.

3.2 Dictionary Based Coding

BLINK determines the optimal partitioning of the denormalized table into cells. The partitioning used a greedy algorithm that calculates the gain achieved in terms of code length if the number of partitions is increased for a column. The frequency histograms are exploited to determine the code length and find optimal splitting points, i. e. the points where a new column partitions should start. The column that contributes the largest gain gets a further column partition. This iterative approach is repeated until the maximum number of cells exceeds a given threshold.

The underlying assumption in that approach is that all data is already available when deriving the frequency histograms. Neither changes in the data distribution nor the addition of completely new values (for which no dictionary did yet exist) could be handled. In order to address those short-comings, we revisit the general partitioning approach and extend it.

Regular Partition. Originally, BLINK had only regular column partitions. All such partitions had an associated dictionary that contained the fieldcodes and their uncoded values. The ordering of fieldcodes in such a dictionary was determined by the ordering of the uncoded values. Those column partitions become *regular partitions*. The regular partitions are determined when the data is initially loaded into BLINK or when a reorganization is done. The cross product of column partitions p_i over all columns i determines all available regular cells, which results in $\prod^i p_i$ cells.

Encoding new values in such dictionaries only works in rare borderline situations if:

1. The dictionary has not yet reached its capacity. Dynamically growing dictionaries is not possible due to the fixed-length codes, which allows only the encoding of 2^n values where n is the length of the field codes in the column partition.

2. The new value to be added is larger than the largest value in the dictionary. Otherwise, the order-preserving property of the dictionary would be violated.

Therefore, we introduce *extension partitions* and *catch-all partitions* for each column to address this issue, which are processed like regular partitions, except for fast SIMD scans.

Extension Partition. All values of a column that cannot be encoded in a regular partition qualify for the single extension partition, i. e. new distinct values added after the dictionaries for regular partitions have been built. An extension partition is still backed up by a dictionary. The dictionary is not necessarily order-preserving.[1] But the values are

[1] If the extension partition uses pure offset coding, its dictionary will be trivially order-preserving, of course.

still encoded, which means compression techniques (cf. section 3.3) can be applied. The dictionary for the extension partition is allocated with such a size that it can accommodate at least as many distinct values as are present in the regular partitions.

The extension partitions of all columns form a single *extension cell*. Extension partitions will never be combined with regular partitions to always guarantee efficient scan processing of range predicates. As soon as one value in a tuple has to be encoded in the extension partition of its column, the whole tuple will go to the extension cell.

The values in an extension partition are either purely offset-coded or pure dictionary-coded, but not a combination of both (cf. section 3.3). The main purpose of an extension partition is to provide a graceful degradation in compression. But once the extension partition is filled, any new values will be placed into the column's catch-all partition.

Catch-All Partition. Each column has a single catch-all partition that stores all its values in an uncoded format. Thus, no advantage of any compression can be achieved. It is used as a last resort to still be able to manage such data at all, even if no efficient evaluation will be possible. All values in such a partition are either identity coded or uncoded (cf. section 3.3), depending on the column length.

Comparable to extension cells, there is only a single *catch-all cell*, which is comprised of the combination of the catch-all partitions of all columns in the table. Once one value in some column c_i cannot be encoded in regular cells and also not in the extension partition of c_i, it falls into the catch-all partition and forces the whole tuple to be placed there.

3.3 Partition Encoding Types

A value in a tuple can be encoded in different ways. The actual encoding depends on the type of the partition (regular, extension, or catch-all) and the dictionary associated with the respective partition. In all cases, it is first attempted to encode the value in a regular partition, preferring partitions with shorter code lengths.

Offset Coding. Per default, a dictionary uses offset coding. The fieldcode of a value is comprised of the combination of the dictionary index and an offset. The length of the fieldcode is the number of bits needed for the dictionary index plus the number of bits encoding the offset. Both of those numbers could be 0, resulting in pure dictionary coding or pure offset coding. A value v encoded to fieldcode fc can be reconstructed by:

$$v = decode(fc) = dictionary[fc.dictIndex] + fc.offset$$

The addition of the offset can take two different forms. In *prefix coding*, uses a bit-wise concatenation. This is used for floating point numbers, for example, that may share a common exponent but have a differing mantissa. Applying a mathematical addition to floating point numbers may give an incorrect result [Gol91]. A concatenation of bits reconstructs the exact same bit pattern of the original value.

For integer types, it makes sense to apply *minus-coding* where the offset is combined with the dictionary value using arithmetic addition. Thus, if the dictionary bits in the fieldcode identify a value of 100 in the dictionary and the offset bits are 32, the reconstructed value will be 132 – regardless of the actual bit pattern of that number.

Dictionaries using offset coding are always order preserving. Fast scans over the data in code space can be applied to cells that are associated to such a dictionary.

Identity Coding. Identity coding uses the uncoded value itself as fieldcode. It is very similar to uncoded values. However, there is still a dictionary associated with such a column partition (and it is trivially order-preserving). The query engine can treat such fieldcodes like purely offset coded values and no special-casing is needed in the code.

This encoding scheme can only be applied if the actual values are short enough to fit into processor registers, i. e. do not exceed the maximum length for fieldcodes. Otherwise, SIMD instructions could not be applied to do the fast scanning BLINK relies on heavily. Identity coding is especially useful for epoch columns as described in section 3.1, for which the actual values and their skew are not known in advance.

Uncoded. There are cases when values cannot be encoded at all or when it does not make sense to do so. Uncoded values require a dedicated storage layout.

Use cases are string columns with very high cardinality of distinct values. Such columns would have huge dictionaries with virtually no (or even a negative) compression ratios. Once BLINK determines those properties for a column, the column is marked as being uncoded and the values are stored as-is without being supported by a dictionary. Another use for not coding values are catch-all partitions where identity coding is not an option (due to the processor register limitations).

3.4 Banks and Tuplets

Internally, BLINK uses highly specialized storage structures for each cell. A cell is broken down into multiple *cell blocks*. A cell block defines a work unit for the query engine. Since multiple threads perform the scanning, the further break down of cells into cell blocks helps to balance the overall workload over all threads and facilitates scan sharing.

Cell blocks are partitioned into *banks*. Each bank contains a projection of a tuple, called a *tuplet*. A cell block contains a mixture of row store and column store, i. e. it is a hybrid of row-major and column-major order. Figure 2 illustrates the bank layout of tuples. The banks define a vertical partitioning. The arrows in the figure indicate the order in which the data is stored in the bank and in the cell block.

Columns are assigned to banks in a cell-specific manner. The deciding criteria are the data type of the column and whether it participates in a reference (foreign key) used to define the join conditions during denormalization. A column is assigned either to a *vertical bank* or a *horizontal bank*. A vertical bank is intended for columns on which typically predicates

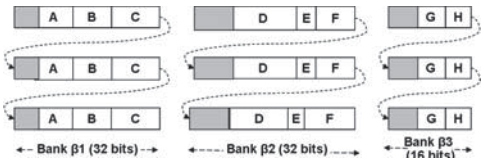

Figure 2: Bank layout

are evaluated while horizontal banks are used for *measure columns* on which aggregation is performed, typically.

Each vertical bank is a separate contiguous unit in the cell block, approximating a column store. Bin-packing is applied to minimize padding as much as possible. Vertical banks can vary in size. One constraint is that the bank width is at most four times the fieldcode length for each column in the bank. This is done to prevent having to scan too much unused data during predicate evaluation. Contrary, horizontal banks use a row store approach.

Uncoded banks, visualized in figure 3, deviate from that because the values may have varying length besides fixed-length values. In order to avoid sequential scans through the banks, offsets pointing directly to the uncoded value in the variable length bank are stored in the encoded portion of the tuplets.

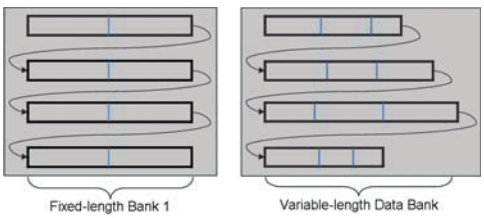

Figure 3: Layout for uncoded banks

4 Null Handling

NULLs were introduced in relational database systems as a method of representing missing data[ISO03]. For example, a table stores customer-related information like the first name, last name, and address of each customer. However, not all customers may have a middle name or middle initial – while others do.

NULLs are not considered to be values of the domain of the data, i. e. NULL is not a string or integer. The introduction of NULLs lead to three-valued logic for predicate evaluation. Only values in the same (or a compatible) domain can be compared with each other and give a well-defined result, namely the comparison is either `true` or `false`. Comparing a value with NULL is defined to be `unknown` [Mul07].

Todays relational database systems typically use internally a special marker for each value

in columns that may contain NULLs. For example, DB2 for z/OS [IBM07] prefixes the value of columns with a byte where `x'00'` means NOT NULL and `x'FF'` represents NULL. For varying length columns this null marker byte appears after the length of the value and it is included in the length. Using such an approach implies that all predicates specifically have to take care of NULLs by enhancing each predicate by adding a verification to test whether the marker is set or not, which causes branching in the evaluation code.

Avoiding Three-Valued Logic. In BLINK, we break with the notion to treat NULL as being excluded from the value's domain. Instead, the frequency of NULLs is explicitly counted in frequency histograms and a code in the column partition's dictionary (to which NULL belongs) is assigned as well. The dictionary entry for NULL (if present in the dictionary at all) will always be the first code, i. e. fieldcode 0. That way, it is possible to exclude or include NULL by simply adjusting the range predicates as table 1 shows for column c_i and value x. No branching is necessary because regular range predicates can be applied.

Predicate in Query	Predicate on fieldcodes
c_i = x	$fieldcode(c_i)$ BETWEEN $encode(x)$ AND $encode(x)$
c_i > x	$fieldcode(c_i)$ BETWEEN $(encode(x)+1)$ AND ∞
c_i <= x	$fieldcode(c_i)$ BETWEEN 1 AND $encode(x)$
c_i IS NULL	$fieldcode(c_i)$ BETWEEN 0 AND 0

Table 1: Handling of NULL in range predicates over fieldcodes

Null Indicator Column. We described in section 3.3 that not all values may be dictionary encoded but rather uncoded. For that case, it is necessary to still have the NULL marker. Another situation requiring such a marker occurs if the whole domain of column c_i is consumed by values in the column and adding fieldcode 0 for NULL requires an additional bit. But increasing the number of bits is not always possible because the size of the encoded values may exceed the maximum bank width.

The NULL marker is not directly attached to each value. Instead, an internal *null indicator column* is added for each nullable column. Such a column is considered in the partitioning like any other column. Thus, it can be dictionary based and it will have at most two regular partitions. The partitions have 0 or 1 bit – 0 if all values in the column partition are the same, which is exploited if fieldcode 0 is used to represent NULL in the data column.

Predicate evaluation of cells that contain a non-dictionary encoded nullable column c_i is implemented by augmenting the predicate for each such column c_i to exclude tuples where the fieldcode in the corresponding null indicator column c_j represents the NULL marker. It is assumed that NULL is represented by 1 in c_j.

$$fieldcode(c_i) \text{ BETWEEN } x \text{ AND } y \text{ AND } fieldcode(c_j) \text{ BETWEEN } 0 \text{ AND } 0$$

5 Summary

BLINK and its compression and scanning technology already came a long way to show that (near) constant query time processing is possible for data warehouse applications. We have shown that the gaps in BLINK's functionality can be closed. This concerns primarily the support for data modifications and NULLs. Using validity intervals, denoted by epoch columns allows the concurrent execution of DML and queries without requiring a locking mechanisms for all rows. Special attention had to be paid to the handling of new data that does not yet exist in the system. NULLs – despite not being a value of a column's domain – can be encoded like any other value and, thus, we can take advantage of SIMD instruction and avoid three-valued logic, except for some very rare cases for uncoded values.

References

[BMK99] P. A. Boncz, S. Manegold, and M. L. Kersten. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB*, 1999.

[Gol91] D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1), 1991.

[HRSD07] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt. How to Barter Bits for Chronons. In *SIGMOD*, 2007.

[IBM07] IBM. *DB2 Version 9.1 for z/OS – Diagnostics Guide and Reference*, 2007.

[ISO03] ISO/IEC 9075-2:2003. *Information Technology – Database Languages – SQL – Part 2: Foundation (SQL/Foundation)*, 2nd edition, 2003.

[JRSS08] R. Johnson, V. Raman, R. Sidle, and G. Swart. Rowwise Parallel Predicate Evaluation. In *VLDB*, 2008.

[MF04] R. MacNicol and B. French. Sybase IQ Multiplex – Designed for Analytics. In *VLDB*, 2004.

[Mul07] C. Mullins. Using Nulls in DB2, 2007. `http://www.db2portal.com/2006/04/using-nulls-in-db2.html`.

[RS06] V. Raman and G. Swart. Entropy Compression of Relations and Querying of Compressed Relations. In *VLDB*, 2006.

[RSQ+08] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-Time Query Processing. In *ICDE*, 2008.

[SAB+05] M. Stonebraker, D. J. 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.

[WKHM00] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The Implementation and Performance of Compressed Databases. *SIGMOD Record*, 29(3):55 – 67, 2000.