Business Analytics in (a) Blink

Ronald Barber[†] Peter Bendel[‡] Marco Czech[•*] Oliver Draese[‡] Frederick Ho[#] Namik Hrle[‡]
Stratos Idreos[§*] Min-Soo Kim[◁*] Oliver Koeth[‡] Jae-Gil Lee[◇*] Tianchao Tim Li[‡] Guy Lohman[†]
Konstantinos Morfonios[△*] Rene Mueller[†] Keshava Murthy[#] Ippokratis Pandis[†] Lin Qiao[○*]
Vijayshankar Raman[†] Richard Sidle[†] Knut Stolze[‡] Sandor Szabo[‡]

[†]IBM Almaden Research Center [‡]IBM Germany [•]SIX Group, Ltd. [#]IBM
[§]CWI Amsterdam [◁]DGIST, Korea [◇]KAIST, Korea [△]Oracle [○]LinkedIn

Abstract

The Blink project's ambitious goal is to answer all Business Intelligence (BI) queries in mere seconds, regardless of the database size, with an extremely low total cost of ownership. Blink is a new DBMS aimed primarily at read-mostly BI query processing that exploits scale-out of commodity multi-core processors and cheap DRAM to retain a (copy of a) data mart completely in main memory. Additionally, it exploits proprietary compression technology and cache-conscious algorithms that reduce memory bandwidth consumption and allow most SQL query processing to be performed on the compressed data. Blink always scans (portions of) the data mart in parallel on all nodes, without using any indexes or materialized views, and without any query optimizer to choose among them. The Blink technology has thus far been incorporated into two IBM accelerator products generally available since March 2011. We are now working on the next generation of Blink, which will significantly expand the "sweet spot" of the Blink technology to much larger, disk-based warehouses and allow Blink to "own" the data, rather than copies of it.

1 Introduction

Business Intelligence (BI) queries typically reference *data marts* that have a "star"- or "snowflake"-shaped schema, i.e., with a huge *fact table* having billions of rows, and a number of smaller *dimension tables*, each representing some aspect of the fact rows (e.g., geography, product, or time). Traditional DBMSs, and even some column stores, rely upon a *performance layer* of indexes [4] and/or materialized views (or "projections" [1]) to speed up complex BI queries. However, determining this layer requires knowing the query workload in advance, anathema to the *ad hoc* nature of BI queries, and increases the variance in response time between those queries that the performance layer anticipates and those it does not.

The Blink project's ambitious goal is to answer *all* BI queries in mere seconds, regardless of the database size, without having to define any performance layer. Blink is a database system optimized for read-mostly BI queries. Blink was built from the ground up to exploit the scale-out made possible by modern commodity

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

[*]Work done while the author was at IBM

multi-core processors and inexpensive DRAM main memories, together with meticulous, hardware-conscious engineering of query processing. What differentiates Blink is that its proprietary dictionary encoding of data and its cache-conscious algorithms combine to minimize consumption of memory bandwidth, to perform most SQL query processing on the encoded data, and to enable single-instruction multiple-data (SIMD) operations on vectors of those compressed values.

This paper provides an overview of the Blink technology, describes the IBM products that have incorporated the first generation of that technology, gives one example of the performance gains on customer data of those products, briefly introduces the research behind the second generation of Blink, and finally compares Blink to related systems before concluding.

2 Blink technology

This section summarizes the essential aspects of Blink technology – how it compresses and stores data at load time, and how it processes that data at query time.

Compression and Storage. In Blink, every column is compressed by encoding its values with a fixed-length, order-preserving dictionary code. Blink uses a proprietary compression method called *frequency partitioning* [12] to horizontally partition the domain underlying each column, based upon the frequency with which values occur in that column at load time. A separate dictionary is created for each partition. Since each dictionary need only represent the values of its partition, each dictionary can use shorter column codes. More precisely, partition P of column C can encode all its values in a dictionary $D_{C,P}$ using only $\lceil \log_2 |D_{C,P}| \rceil$ bits. In [12] we show that frequency partitioning approaches the efficiency of Huffman coding as the number of partitions grow, but has the advantage of generating fixed-length codes. Furthermore, encoded values are assigned in each dictionary in an order-preserving way, so that range and equality predicates can be applied directly to the encoded values.

Rows of tables are then loaded in encoded and packed form, as described below, horizontally partitioned according to the partitioning of their values for each column. Since each column has fixed width within a partition, each row in that partition therefore has the same fixed format.

While relational DBMSs since System R have laid out tables in row-major order for maximal update efficiency, many recent read-optimized systems use a column-major order, so that each query need only scan the columns it references (e.g., Sybase IQ [10], MonetDB [4], C-Store [1]). We argue that both row-major and column-major layouts are suboptimal: the former because queries have to scan unreferenced columns, and the latter because encoded columns must be padded to word boundaries for efficient access. Instead, Blink vertically partitions the bit-aligned, encoded columns of each horizontal partition into a family of fixed-size, byte-aligned *banks* of 8, 16, 32, or 64 bits, to allow efficient ALU operations. Since the width of each column varies from one horizontal partition to the next, so too may this assignment of columns to banks. The bin-packing algorithm that performs this assignment seeks to minimize padding, not bank accesses, and is therefore insensitive to how often columns in each bank are referenced together in queries. We then lay out the table in storage in *bank-major* order – each *tuplet* of a bank contains the encoded values of one row for the columns assigned to that bank. Large, fixed-size blocks are formed with all the banks for a range of RIDs, in a PAX-like format [2].

Banked layouts permit a size trade-off. Wide banks are more compact, because we can pack columns together with less wasted space. When a query references many of the columns in the bank, this compactness results in efficient utilization of memory bandwidth, much as in a row-major layout. On the other hand, narrow banks are beneficial for queries that reference only a few columns within each bank. In the extreme case, when each column is placed in a separate bank, we get a column-major layout, padded to the nearest machine word size. Blink's banked layouts exploit SIMD instruction sets on modern processors to allow a single ALU operation to operate on as many tuplets in a bank as can be packed into a 128-bit register. Narrow banks rely on SIMD processing for greater efficiency.

Overview of Query Processing. Query processing in Blink is both simpler and more complex than in traditional DBMSs. It's simpler because Blink has only one way to access tables – scans (recall Blink has no indexes or materialized views) – and always performs joins and grouping using hash tables in a pre-specified order. So it needs no optimizer to choose among alternative access paths or plans. On the other hand, it's more complex because the query must be compiled for the different column widths of each (horizontal) partition. To limit this overhead, we set an upper limit on the number of partitions at load time.

Query processing first breaks an SQL query into a succession of scans called *single-table queries (STQs)*. Although we originally anticipated denormalizing dimension tables (at load time) to completely avoid joins at run time in Blink [12], customer data proved that the redundancy introduced by denormalizing would more than offset our excellent compression [3]. We therefore abandoned this assumption and implemented (hash) joins. Joins are performed by first performing an STQ that scans each dimension table and builds a hash table of the rows that survive any *local predicates* to that table. Then the fact-table STQ scans and applies predicates local to the fact table, probes the hash table for each of its dimensions to apply its *join predicate(s)*, and finally hashes the grouping columns and performs the aggregates for that group. Queries against snowflake queries repeat this plan recursively, "outside-in", starting with the outer-most dimension tables. Between these outer-most tables and the central fact table, the intermediate "hybrid" dimension tables will act both as a "fact" and as a "dimension" table in the same STQ. That is, each hybrid STQ: (a) probes all tables that are "dimension" tables relative to it; and then (b) builds a hash table for the subsequent join to the table that is "fact" relative to it.

Blink compiles each STQ from value space to code space and then runs the STQ directly on the compressed data without having to access the dictionary. This compilation from value space to code space has to be done separately for each partition, because the dictionaries are different for each partition. However, a benefit of this dictionary-specific compilation is that all rows in an entire partition may be eliminated at compile time if the value(s) for a local predicate cannot be found in that partition's dictionary. For example, a predicate `StoreNo = 47` cannot be true for any row in any partition not having `47` in its dictionary for `StoreNo`, which will be the case for all but one partition.

Blink executes an STQ by assigning blocks of a partition to threads, each running on a core. Since partitions containing more frequent values will, by construction, have more blocks, threads that finish their assigned partition early will help out heavier-loaded threads by "stealing" some unprocessed blocks from the bigger partitions to level the load automatically. In each block, a Blink thread processes all the rows in three stages, with the set of RIDS of qualifying tuples passed from one to the next as a bit vector: (a) fastpath predicate evaluation: applies conjuncts and disjuncts of range and short IN-list selection predicates; (b) residual predicate evaluation: applies remaining predicates, including join predicates, with a general-purpose expression interpreter; (c) grouping and aggregation: each thread maintains its own hash table for grouping and aggregation, to avoid locking or latching, and the resulting aggregates are combined at the end to produce the final answer.

Most modern ALUs support operations on 128-bit registers. By packing the codes for multiple instances of multiple columns into a single 128-bit unit and applying predicates on these multi-tuplets simultaneously, Blink can evaluate the column comparisons on N column values that have been compressed to B bits each using only $N/\lfloor 128/B \rfloor$ operations, as compared to N operations using the standard method.

Wherever possible, operators process compressed codes. Because we have constructed a dictionary of values, we can convert complex predicates on column values, such as LIKE predicates, into IN-lists in code space by evaluating the predicate on each element of the dictionary. Due to order-preserving dictionary coding, all the standard predicates ($=, <, >, \leq, \geq$) map to integer comparisons between codes, irrespective of data type. As a result, even predicates containing arbitrary conjunctions and disjunctions of atomic predicates can be evaluated using register-wide mask and compare instructions provided by processors, as described in [8]. A Decode operator decompresses data only when necessary, e.g., when character or numeric expressions must be calculated.

Blink batch-processes a large gulp of rows, called a *stride*, at each stage of query processing, as in [11, 4], to exploit ILP and SIMD. Short-circuiting for non-qualifying tuples only occurs between stages, to minimize the high cost of mispredicting branches that short-circuiting causes. Run-time operators produce a bit vector

of predicate results, with one bit for each input RID. The final step of the Residual stage uses this bit vector to produce the final RID list of qualifying tuples, which is then passed to grouping and aggregation.

3 Accelerator Products Based on Blink

Although prototyped initially as a stand-alone main-memory DBMS [12], the Blink technology has been incorporated into two IBM products thus far as a parallelized, main-memory accelerator to a standard, disk-based host DBMS. Once the user has defined the mart of interest to be accelerated, a bulk loader extracts a copy of its tables automatically from the data warehouse, pipes the data to the accelerator, analyzes it, and compresses it for storage on the nodes of the accelerator, which can be either blades in a commodity blade center or segments of a single server. The assignment of data to individual nodes is not controllable by the user. Fact tables are arbitrarily partitioned among the nodes, and dimensions are replicated to each. Once the data has been loaded in this way, SQL queries coming into the host DBMS that reference that data will be routed automatically by the host optimizer to Blink, where it will be executed on the accelerator's compressed data, rather than on the host DBMS. The SQL query is first parsed and semantically checked for errors by the host DBMS before being sent to the accelerator in a pre-digested subset of SQL, and results are returned to the host DBMS, and thence to the user. Note that users need not make any changes to their SQL queries to get the router to route the SQL query to the accelerator; the query simply has to reference a subset of the data that has been loaded. Below, we briefly describe each of these IBM products in a bit more detail.

IBM Smart Analytics Optimizer (ISAO). The *IBM Smart Analytics Optimizer for DB2 for z/OS V1.1 (ISAO)* is an appliance running Blink, called the *zEnterprise Blade eXtension (zBX)*, which is network-attached to the *zEnterprise* mainframe containing a standard, disk-based data warehouse managed by DB2 for z/OS. The user doesn't really see this accelerator, as there are no externalized interfaces to it. The zBX is a modified Blade Center H containing up to 14 blades, and ISAO can accommodate multiple such zBXs for scale-out. Blades are (soft-) designated as either *coordinators* or *workers*. Coordinator blades receive queries from DB2 and broadcast them to the workers, then receive partial answers from the workers, merge the results, and return them to DB2. There are always at least 2 active coordinator blades to avoid a single point of failure, plus one held in reserve that can take over for any worker blade that might fail by simply loading its memory image from a disk storage system that only backs up each worker node.

Each blade contains two quad-core Nehalem chips and 48 GB of real DRAM. Only worker blades dedicate up to 32 GB of DRAM for storing base data; the rest is working memory used for storing the system code and intermediate results. A fully-populated zBX having 11 worker blades would therefore be capable of storing 11 * 32 GB = 352 GB of compressed data, or at least 1 TB of raw (pre-load) data, conservatively estimating Blink compression at 3x (though much higher compression has been measured, depending upon the data).

Informix Warehouse Accelerator (IWA). Blink technology is also available via a software offering, called the *Informix Ultimate Warehouse Edition (IUWE)*. The Blink engine, named *Informix Warehouse Accelerator (IWA)*, is packaged as a main-memory accelerator to the *Informix database server*. The Informix server, when bundled with IWA as IUWE, is available on Linux, IBM AIX© , HP-UX, and Oracle Solaris. When running on Linux, the database server and IWA can be installed on the same or different computers. When both Informix and IWA are running on the same machine, the coordinator and worker nodes simply become processes on the same machine that communicate via loopback. Informix and IWA can be on distinct SMP hardware, with IWA running both the coordinator and worker processes on the same hardware. IWA can also be deployed on a blade server supporting up to 80 cores and 6 TB of DRAM, each blade supporting up to 4 sockets and 640 GB of DRAM. The IUWE software was packaged flexibly enough to run directly on hardware or in a virtualized/cloud environment. Each database server can have zero, one, or more IWAs attached to it.

Query #	1	2	3	4	5	6	7
Informix 11.50	22 mins	3 mins	3 mins 40 secs	>30 mins	2 mins	30 mins	>45 mins
IWA + Informix 11.70	4 secs	2 secs	2 secs	4 secs	2 secs	2 secs	2 secs
Speedup	330x	90x	110x	>450x	60x	900x	>1350x

Table 1: Execution times for Skechers queries with and without on Informix Warehouse Accelerator

4 Performance

Blink is all about query performance. Space constraints limit us to a brief summary of one customer's initial experience; for more, see [3]. Skechers, a U.S. shoe retailer, uses Informix for their inventory and sales data warehouse, which has fact tables containing more than a billion rows. Queries took anywhere from a few minutes to 45 minutes to run on their production server running Informix 11.50. During the IWA Beta program, Skechers tested IWA with the same data and workload. The queries took just 2 to 4 seconds on the Informix Warehouse Accelerator, a 60x to 1400x speed-up, as shown in Table 1. Note IWA's low variance.

5 The Next Generation of Blink

Leveraging our experience with the initial products based upon the first generation of Blink, we are now prototyping the next generation of Blink, to widen the "sweet spot" provided by Blink. First of all, we are relaxing the requirement that all data fit in main memory and allowing Blink tables to be stored on disk, while retaining our main-memory legacy of high-performance cache-conscious algorithms and multi-core exploitation. This re-introduces disk I/O concerns, arguing for Blink to be a more pure column store by favoring "thin" banks, i.e., allocating a single column to each block by default. Since each partition of each column may be represented by a different number of bits, each block may therefore contain a different number of tuplets. Stitching together all the referenced columns for a particular row now becomes quite a challenge, as they're not all in the same block, or even in the same relatively-numbered block for each column. And over-sized intermediate results must be able to spill to disk. Secondly, since disk storage is persistent, Blink tables can now "own" the base data, rather than a copy of the data. This obviates the problem of keeping multiple copies in sync, but raises performance issues for "point" queries to Blink tables. Will we have to backtrack on indexes? And we still need a mechanism for updates, inserts, and deletes, both in batches and as a "trickle-feed", including ways to evolve the dictionaries by which Blink encodes data. Thirdly, we are rethinking our algorithms and data structures for both joins and grouping to minimize cache misses and still avoid any locking or latching between threads, to ensure good scaling as the number of cores increase exponentially. Fourthly, interrupts for disk I/Os once again create opportunities for context switching among multiple concurrent queries, necessitating careful allocation of resources among these queries for maximum efficiency. These and other issues are the focus of our current research.

6 Related Work

Numerous new systems in industry and academia target fast processing of BI queries, but unfortunately most of the commercial systems have very limited or no documentation in the refereed literature. Compared to existing systems such as Vertica [1], which is based upon the C-store academic prototype [13], and VectorWise [7], which is derived from MonetDB [5] and X100 [4], Blink is closer to the vectorized storage and processing model pioneered by VectorWise. C-store creates *projections*, a redundant copy of the base data sorted on a leading

[1] Vertica, an HP Company: http://www.vertica.com

column and resembling an index that can exploit run-length encoding to reduce storage and processing overhead. Blink introduces a more advanced order-preserving compression scheme, frequency partitioning, which allows it to achieve a good balance between reducing size while still maintaining fixed-width arrays and performing most database operations on the encoded data. HYRISE [6] and HyPer [9] are both recent academic main-memory DBMSs for mixed (OLTP and BI) workloads, but their real-world feasibility remains to be proven. The main feature of Hyper is that it exploits the ability of modern hardware and operating systems to create virtual memory snapshots by duplicating pages on demand when BI queries conflict with OLTP queries. This allows BI queries to see a very recent snapshot of the data, while OLTP queries can continue in parallel. The main contributions of HYRISE seem to be an offline analysis tool for deciding the proper grouping of columns and physical layout to optimize performance for a given mixed workload, which may be valuable for organizing banks in Blink, and a detailed cost model for each operator in terms of cache misses. SAP's HANA[2] and its predecessor Netweaver Business Warehouse Accelerator (BWA) are main-memory DBMSs that resemble Blink by using dictionary encoding to pack multiple values in a register and exploit SIMD operations to do decompression and (local) predicate evaluation. However, unlike Blink, the values of only one column are packed without padding to align to register boundaries, so that values may span register boundaries, creating challenges for processing values on those boundaries and extracting results using a clever series of complicated bit shifts [14].

7 Conclusions

Radical changes in hardware necessitate radical changes in software architecture. Blink is such a radically novel architecture–a main-memory, special-purpose accelerator for SQL querying of BI data marts that exploits these hardware trends. It also exploits proprietary order-preserving compression techniques that permit SQL query processing on the compressed values and simultaneous evaluation of multiple predicates on multiple columns using cache-conscious algorithms. As a result, Blink can process queries in simple scans that achieve near-uniform execution times, thus speeding up the most problematic queries the most, without requiring expensive indexes, materialized views, or tuning. Completely obviating the need for a tunable "performance layer" is the best way to lower administration costs, and hence the total cost of ownership and time to value.

References

[1] D. Abadi et al. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.
[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, 2001.
[3] R. Barber et al. Blink: Not your father's database! In *BIRTE*, 2011.
[4] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.
[5] P. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51, 2008.
[6] M. Grund et al. HYRISE—A main memory hybrid storage engine. *PVLDB*, 4, 2010.
[7] D. Inkster, M. Zukowski, and P. Boncz. Integration of VectorWise with Ingres. *SIGMOD Rec.*, 40, 2011.
[8] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *PVLDB*, 1, 2008.
[9] A. Kemper and T. Neumann. HyPer – a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, 2011.
[10] R. MacNicol and B. French. Sybase IQ Multiplex - Designed for analytics. In *VLDB*, 2004.
[11] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *ICDE*, 2001.
[12] V. Raman et al. Constant-time query processing. In *ICDE*, 2008.
[13] M. Stonebraker et al. C-store: a column-oriented DBMS. In *VLDB*, 2005.
[14] T. Willhalm et al. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2, 2009.

[2] http://www.intel.com/en_US/Assets/PDF/whitepaper/mc_sap_wp.pdf