Evolutionary Integration of In-Memory Database Technology into IBM's Enterprise DB2 Database Systems

Knut Stolze*
stolze@de.ibm.com

Guy Lohman†
lohmang@us.ibm.com

Vijayshankar Raman†
ravijay@almaden.ibm.com

Richard Sidle†
rsidle@us.ibm.com

Felix Beier*
fbeier8@de.ibm.com

Abstract:
Recently, IBM announced Blink Ultra (BLU) as an in-memory enhancement for DB2 for Linux, Unix, and Windows. The technology implemented in BLU was tested in various stages until it founds its current form as DB2 feature. In this paper, we give a brief summary on the origins of BLU and the adoption process from the BLINK prototype over the IBM Smart Analytics Optimizer to BLU itself.

1 Introduction

In-memory database technology is very important for highly efficient execution of complex, analytic SQL statements. DB2 for Linux, Unix, and Windows (DB2 LUW) was enhanced recently with Blink Ultra (BLU) to offer this technology as part of a well-established, enterprise relational database system. Such an integration into existing infrastructure is extremely important because it lowers the bar for adopting innovative ideas. No new product has to be installed, maintained, monitored, and tuned – it is all part of DB2 LUW, with which customers are familiar with.

BLU is based on the Blink [RSQ+08, BBC+11] research prototype that was developed originally in IBM's Almaden Research Center. Blink was transformed into a product, the IBM Smart Analytics Optimizer [Dra09] as an accelerator for DB2 for z/OS. In fact, the new functionality is deeply embedded in the DB2 kernel itself. It uses a columnar data store with its own page format and specialized run-time implementation. It's run-time handling was developed from ground up in a unique fashion to exploit multi-core architecture. The algorithms are designed architecture-aware and are very cache-conscious. Internally, BLU uses an order-preserving dictionary compression to support equality predicates and also range predicates already on compressed data.

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

The net result of BLU are order of magnitude improvements for performance and compression. At the same time, putting BLU to the task is very simple by specifying a new keyword in the `CREATE TABLE` SQL statement. Thus, customers can easily adopt it and very quickly benefit from it. In short, time to value is extremely positive.

The first ideas for BLU can be dated back to the year 2007. The development of the first prototype was started in the IBM Almaden Research Center. Soon, initial performance results could be determined. Comparisons with other products shown that this initial prototype could beat competing products by a factor of 2x. Due to this very promising outlook, productization of Blink was started in earnest in 2008 with the IBM Smart Analytics Accelerator (ISAO). A first alpha version of ISAO was evaluated on real customer data already towards the end of 2008. As it turned out, some of the assumptions made on Blink and the design decisions adopted for ISAO had to be revisited (cf. Section 3). ISAO for DB2 z/OS Version 1 saw the light of day in year 2010. While ISAO was further developed and, for example, a ported to Informix IDS, integration work of the core technology of the Blink query engine into DB2 LUW. The first alpha release for BLU was ready to ship to early adopters in 2012. Finally, BLU was officially announced and made generally available just a few weeks ago.

In the following sections, we describe technical details of BLU's roots. The initial Blink prototype is presented in Section 2. The first productization, the IBM Smart Analytics Optimizer, brought many new insights and design changes. Key points of that are highlighted in Section 3. A very short overview on BLU is given in Section 4 before we conclude with a summary in Section 5.

2 BLINK

Blink [RSQ+08, SRSD09] is a query processor that was designed to answer all queries in constant or near-constant time with respect to query complexity. This goal was to be 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 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.

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. While variable length codes provide good compression rates, it makes query processing much harder and slower because no direct access to compressed data is possible [WKHM00, HRSD07]. Therefore, Blink uses frequency partitioning to have fixed-length codes in each partition, which allows the leveraging of 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*.

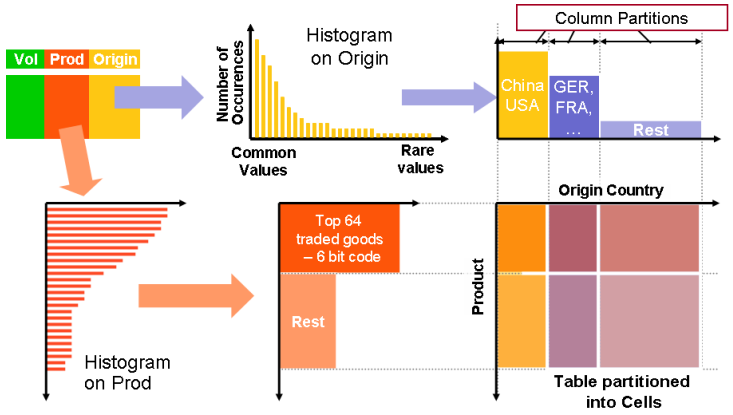

Figure 1: Illustration of Frequency Partitioning

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 histogram 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 $R^i_1 \ldots R^i_{p_i}$,

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 R into cells. Each cell is labeled with an $id \in \times_i [1, p_i]$. Given a cell with id $(\theta_1, \ldots, \theta_n)$, 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 we use 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].

3 IBM Smart Analytics Optimizer

The IBM Smart Analytics Optimizer (ISAO) is an appliance consisting of a compute cluster and a parallel MMDBMS. It was developed based on the Blink query processor for reacting to the trends in modern analytics data warehousing environments. It is aimed to improve warehousing on System z as an extremely well-performing but still cost-efficient solution.

The main idea is offering an hybrid approach for supporting both OLTP and OLAP workloads with quality of service guarantees close to those for System z. The integration of two specialized systems is the primary design idea. As illustrated in Figure 2, OLTP transactions are executed directly by DB2 for z/OS and more expensive OLAP queries are offloaded by DB2 transparently to ISAO. The appliance is attached to the mainframe via TCP/IP network. DB2 recognizes it as an available resource and offloads query blocks only if it is deemed to be beneficial, based on the decision by DB2s cost-based optimizer.

Complex analytical queries are split into query blocks which are executed in parallel by ISAO on several physical nodes in a cluster. Their partial results are merged and transferred back to DB2 completely transparent to user applications which still use DB2s SQL interface without further configuration or source code modifications. ISAOs available main memory and processing power can be scaled by extending the cluster size.

Those parts of the warehouse data (called data marts in the following) which are critical for the analytical workload are replicated from DB2 z/OS to ISAO. While the data is primarily stored on disks in DB2 (and supported by DB2s buffer pools), it is fully held highly compressed in the main memory of the appliance (and only backed with disk storage for

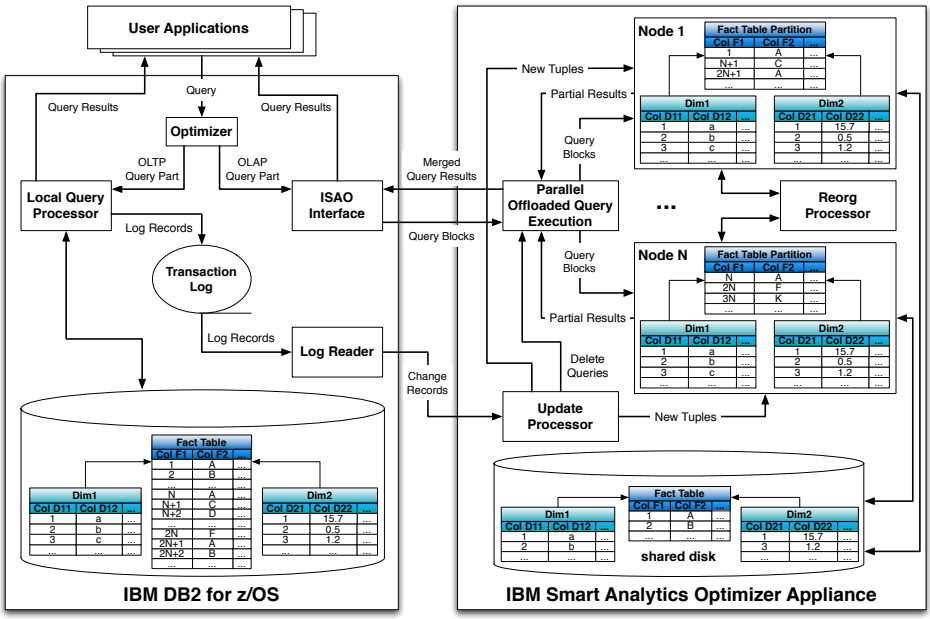

Figure 2: Overview: IBM Smart Analytics Optimizer

recovery purposes). The used compression technique allows a fast evaluation of equality and range predicates using bitmasks directly on the compressed values and hence avoids an expensive decompression in many cases. Furthermore, a cache efficient data structure is used which enables the predicate evaluation of queries with vector operations [JRSS08]. With these techniques, queries can be accelerated by orders of magnitude despite the fact that no workload specific tuning techniques like indexes or materialized views are used.

3.1 Challenges

The integration of two systems – DB2 for z/OS and the Blink query engine – poses unique issues. As [SBKS11] outlined, the query semantics must match. Although SQL is standardized, there are very often (quite subtle) differences to be found in the syntax, supported data types and operators, and so on. With ISAO, we had a huge advantage because the system was built from ground up and, therefore, could easily be tailored towards the behavior of DB2 for z/OS.

The design decisions made for ISAO arose with another set of challenges. As it turned out, the frequency partitioning from Blink over denormalized data is not always applicable to real-world, customer data sets [SBKS11]. When pre-joining multiple tables, the resulting denormalized table becomes very wide, which reduces the effectiveness of the

partitioning. As a consequence, ISAO discarded the notion of denormalizing tables and rather introduced run-time joins. That added to the required development work since no existing infrastructure could be reused quickly.

Having a clean slate for product development may make many things easier during prototypical stages because seemingly uninteresting parts can be ignored. However, if functional gaps are not closed in the final product, it tends to have a wider impact than originally anticipated. For example, ISAOs lack of support for nested sub-queries lead to the situation that DB2 for z/OS could only accelerate those parts of a query, which were the inner-most query blocks. Typically, those were scans over dimension tables in a star or snowflake schema. The more costly scan over the fact table had often to be handled by DB2 itself. Aside from the restriction on the inner-most query blocks, there were some functions available in DB2, which ISAO had not implemented. Queries exploiting those could not be accelerated either. Hence, customers experienced that only a small portion of their workload could be handled by ISAO. While that portion was processed very efficiently, the bult of the overall workload still had to run on DB2.

Finally, having two specialized systems with their own, dedicated data storage, requires the data to be synchronized. Always keeping the data in ISAO in sync with DB2 would have put a heavy toll on non-OLAP workload, which was not acceptable. Therefore, an asynchronous update of the data was realized, and customers can control when which part of the data shall be refreshed on the accelerator. That approach also allowed for a better compression since Blink's frequency partitioning did not support incremental updates efficiently, especially if only very few changes arrived.

3.2 A Fork in the Road

The issues uncovered during the development of ISAO for DB2 for z/OS, lead to two different alternatives how to tackle them. Due to architectural differences between System z (running DB2 for z/OS) and Linux (running the ISAO server), the approach to integrate two systems running on two different hardware platforms was maintained. However, Blink as the query engine for ISAO was replaced with Netezza [Fra11]. Netezza, as a extremely well performing data warehouse solution, is disk-based and it exploits FPGAs for extremely fast scanning and pruning of the data during query processing. At the same time, it offers a much wider spectrum of SQL functionality, including support for subqueries. The IBM DB2 Analytics Accelerator (IDAA) [BBF+12] was the result of this replacement of the query engine.

The development team for DB2 LUW was also highly interested in the Blink technology. Since DB2 LUW and Blink run on the same hardware and operating system platform (Linux on Intel and AIX on Power), the natural solution was to integrate Blink functionality into the database kernel. Section 4 goes into more details on that.

4 DB2 BLU

The basic concept for BLU is that a new table type is introduced natively in DB2 LUW. Those BLU tables can co-exist with traditional row-oriented tables – even in the same schema, storage, and in memory. BLU tables use a column-oriented storage layout in main memory. It is possible to convert existing row-oriented tables to BLU tables, either all tables together or just a selected set of tables. Thus, an incremental transition and exploitation of BLU functionality is provided, which is very attractive to customers.

Queries can access BLU tables as well as traditional tables and combine both. The DB2 kernel will exploit BLU wherever possible, i. e. scan the data, apply predicates, join BLU tables, and so on. Once all possible processing in BLU has finished the data representation is transformed to the traditional layout and further processing (e. g. joins with other tables) is done.

4.1 Similarities with ISAO

BLU retains much of the legacy inherited from Blink and ISAO. The primary use case for BLU tables and their data are read-mostly workloads, which is typical for data warehouse systems. Multi-core systems are heavily exploited via data parallelism. SIMD processing to evaluate predicates on compressed data is used wherever in order to avoid decompression of values that are not part of the final or an intermediate result set. The ideas of frequency partitioning (FP) have been adopted. The dictionaries resulting from FP are order-preserving and column-specific as well. The algorithms are cache-conscious to reduce the transfer of data from main memory to the CPUs caches, and they are working on large strides of rows (similar to VectorWise [Cor09]). Joins and grouping operators use fast in-memory hash tables.

4.2 Improvements over ISAO

While ISAO maintained a copy of all the data in DB2, BLU does not do so. A BLU table is the native, primary storage of the table data. Any data modifications on the table's data are directly done inside BLU. No issues regarding data synchronization arise since there is only a single copy.

ISAO was a pure main memory system, BLU deviates from that. The data in BLU tables is stored on pages, and those pages are handled as usual by DB2's buffer pool. The eviction strategies of the buffer pool are adjusted to make them "BLU aware", i. e. to not treat BLU tables in the same way as traditional tables during full-table scan and prefetching. That increases the "sweet spot" for BLU, e. g. supporting more data, spilling of intermediate results, and the co-existence with other tables – even on systems with only a smaller amount of physical memory.

The PAX format used by ISAO where data data was stored in column groups was discarded. The main issue with column groups turned out to be the automated choice of the right groups for most efficient data compression and to align this with the column usage in queries. In other words, the compressed column values that could be stored most efficiently together were not always the same columns used in predicates in queries together. BLU stores individual columns per default, not column groups. Thus, BLU became a pure column store.

ISAO applied histogram-based frequency partitioning on the columns. The cross-product of all column partitions resulted in a set of disjoint cells, and a query was specialized for each cell. Unfortunately, the number of cells grows exponentially with the number of column partitions. BLU uses FP to partition only the values in a single column, independent of the partitioning of other columns. Therefore, the concept of cells is no longer applicable and neither is the pre-compilation of queries for an exponential number of such cells.

The compression dictionaries in ISAO were built when the data was initially loaded. Adding new values to those order-preserving dictionaries was not possible, except under some rare borderline conditions. The resulting dictionary was static and it applied to the whole table. BLU still has the table-level dictionary, but additionally, an adaptive page-level compression was introduced. This evolving compression further reduces the storage requirements of data in BLU tables and bypasses the relative inflexibility of the global, table-level dictionary.

Lastly, ISAO processed single query blocks only. BLU can handle multiple query blocks together. In case of missing functionality (e. g. unsupported functions or sub-selects), the existing DB2 functionality compensates.[1]

4.3 SQL Query Processing

A primary goal for integrating BLU into the DB2 kernel was to exploit its functionality for as many queries as possible. Therefore, BLU became a DB2 component – similar as it was done for XML. The BLU component is used at run-time, and it has its own page formats and large blocks. As a consequence, BLU tables enjoy full DB2 SQL.

Since BLU does not re-implement all the functions that DB2 already has had, the "classic" DB2 run-time is re-used to compensate for this. The BLU run-time only executes the subset of SQL that it can, i. e. the subset where it excels: column-oriented table scans with predicate evaluation, joins, grouping, and aggregation, mostly on compressed data. The re-use of existing functionality avoided the creation of another comprehensive run-time layer and, thus, reduced the development effort and sped up the time to market. Figure 3 illustrates the integration of the BLU run-time into the DB2 kernel.

The integration into the DB2 kernel came with some beneficial side effects. For example, no semantic SQL validation is implemented because BLU relies on DB2's SQL parser and

[1] This compensation is handled very similar to compensating gaps in federated environments where a remote server may not provide all the features that the federated server has, and the federated server compensates for that.

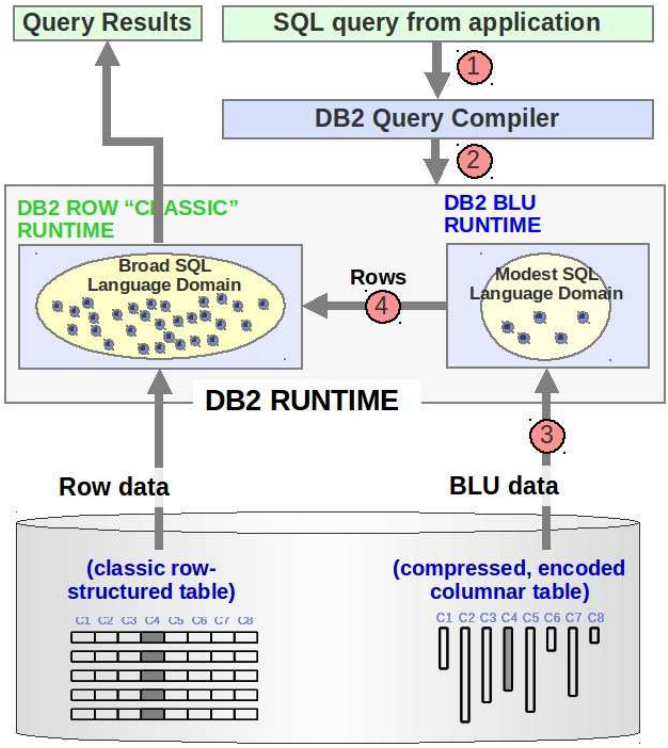

Figure 3: Query Processing using BLU

optimizer for that. The DB2 query rewrite and optimizer is used to simplify nested queries into joins. So even if BLU cannot handle nested sub-queries itself, the combination with the DB2 optimizer (which is aware of BLU tables), provides this capability.

4.4 Evaluation

BLU is a very new feature of DB2 LUW. To demonstrate its value, performance measurements were done with customer data.[2]

A comparison of BLU tables vs. traditional DB2 tables was conducted for a retail customer and also for the workload produced by a Cognos system. Figure 4 shows that in both cases a query speedup of at least 20x could be achieved. Naturally, the results may vary for different customers and workloads.

[2] Artificial benchmarks were not used due to their limited value in specific customer situations.

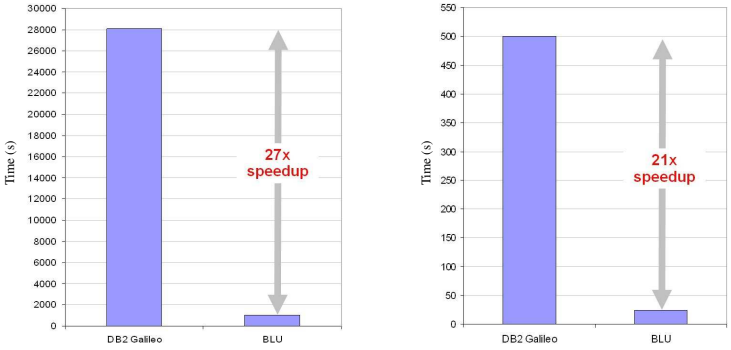

Figure 4: BLU Query Speedup

One aspect in the query performance is the memory consumption, which translates to the compression rates. Several evaluations were conducted in this regard, and a compression ratio of 10x was typical for BLU (cf. Figure 5), sometimes even better.

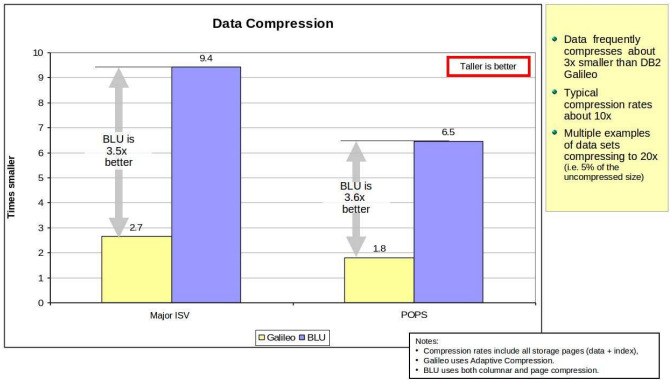

Figure 5: BLU Compression Rates

5 Summary

DB2 is an enterprise-scale database system offered by IBM. It is available on multiple platforms, ranging from mobile devices to Linux, Unix, and Windows systems, to mainframes (System z). Efficiently supporting analytics operations is important for such a product. In this paper, we outlined how main-memory technology and exploitation of the underlying hardware (special processor instructions, cache-conscious programming, ...) found its way from a research prototype "Blink" into DB2 for LUW.

Further improvements on BLU (and also on IDAA) are actively being worked on. The goals are to increase the "sweet spot" for queries, increase compression rates without loosing query performance, just to name the most important ones. In IDAA itself, the topic of main memory query processing is also very much relevant.

6 Trademarks

References

[BBC+11] R. Barber, P. Bendel, M. Czech, O. Draese, F. Ho, N. Hrle, S. Idreos, M.-S. Kim, O. Koeth, J.-G. Lee, T.T. Li, G. Lohman, K. Morfonios, K. Murthy, L. Qiao, V. Raman, S. Szabo, R. Sidle, and K. Stolze. Blink: Not Your Fathers Database! In *BIRTE*, 2011.

[BBF+12] P. Bruni, P. Becker, W. Favero, R. Kalyanasundaram, A. Keenan, S. Knoll, N. Lei, C. Molaro, and PS Prem. *Optimizing DB2 Queries with IBM DB2 Analytics Accelerator for z/OS*. IBM Redbooks, 2012.

[Cor09] Ingres Corporation. Ingres/VectorWise Sneak Preview on the Intel¡AE¿ Xeon¡AE¿ 5500 Platform. Whitepaper, Ingres Corporation, 2009.

[Dra09] O. Draese. IBM Smart Analytics Optimizer Architecture and Overview. Presentation Slides IOD 2009 Session Number TDZ-2711, IBM Corp., October 2009.

[Fra11] P. Francisco. *The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics*. IBM Redbooks, 2011.

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

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

[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.

[SBKS11] K. Stolze, F. Beier, O. Koeth, and K.-U. Sattler. Integrating Cluster-Based Main-Memory Accelerators in Relational Data Warehouse Systems. *Datenbank Spektrum*, 2011.

[SRSD09] K. Stolze, V. Raman, R. Sidle, and O. Draese. Bringing BLINK Closer to the Full Power of SQL. In *BTW*, pages 157–166, 2009.

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