DB2 with BLU Acceleration:
So Much More than Just a Column Store

Vijayshankar Raman   Gopi Attaluri⋆   Ronald Barber   Naresh Chainani⋆
David Kalmuk⋆   Vincent KulandaiSamy⋆   Jens Leenstra◇   Sam Lightstone⋆
Shaorong Liu⋆   Guy M. Lohman   Tim Malkemus   Rene Mueller
Ippokratis Pandis   Berni Schiefer⋆   David Sharpe⋆   Richard Sidle
Adam Storm⋆   Liping Zhang⋆

IBM Research   ⋆IBM Software Group   ◇IBM Systems & Technology Group

ABSTRACT

DB2 with BLU Acceleration deeply integrates innovative new techniques for defining and processing column-organized tables that speed read-mostly Business Intelligence queries by 10 to 50 times and improve compression by 3 to 10 times, compared to traditional row-organized tables, without the complexity of defining indexes or materialized views on those tables. But DB2 BLU is much more than just a column store. Exploiting frequency-based dictionary compression and main-memory query processing technology from the Blink project at IBM Research - Almaden, DB2 BLU performs most SQL operations – predicate application (even range predicates and IN-lists), joins, and grouping – on the compressed values, which can be packed bit-aligned so densely that multiple values fit in a register and can be processed simultaneously via SIMD (single-instruction, multiple-data) instructions. Designed and built from the ground up to exploit modern multi-core processors, DB2 BLU's hardware-conscious algorithms are carefully engineered to maximize parallelism by using novel data structures that need little latching, and to minimize data-cache and instruction-cache misses. Though DB2 BLU is optimized for in-memory processing, database size is not limited by the size of main memory. Fine-grained synopses, late materialization, and a new probabilistic buffer pool protocol for scans minimize disk I/Os, while aggressive prefetching reduces I/O stalls. Full integration with DB2 ensures that DB2 with BLU Acceleration benefits from the full functionality and robust utilities of a mature product, while still enjoying order-of-magnitude performance gains from revolutionary technology without even having to change the SQL, and can mix column-organized and row-organized tables in the same tablespace and even within the same query.

1. INTRODUCTION

As enterprises amass large warehouses of transactional data, they have increasingly sought to analyze that data for trends and exceptions that will provide insights leading to profitable and actionable business decisions. These so-called Business Intelligence (BI) systems are read-mostly, typically inserting "cleansed" and standardized rows in bulk from various transactional systems, and deleting them only when it becomes unprofitable to keep older data on-line. As the data warehouse grows, however, the variation in response times to BI queries grows, depending upon whether the database administrator (DBA) anticipates exactly which queries will be submitted and creates the appropriate indexes and materialized views (or materialized query tables (MQTs)) needed by traditional database management systems (DBMSs) to perform well. Response times can vary from seconds - for those with the optimal indexes and/or MQTs - to hours or even days for those without that "performance layer". But BI queries are inherently *ad hoc*, in which the formulation of each query depends upon the results of the previous queries. So it is almost impossible to anticipate the indexes and MQTs needed for all possible queries, and hence to achieve truly interactive querying, with today's systems.

2. OVERVIEW

This paper describes an innovative new technology for BI queries called BLU Acceleration, which is part of DB2 for Linux, UNIX, and Windows (LUW) 10.5 [13]. DB2 with BLU Acceleration (henceforth called DB2 BLU for brevity) accelerates read-mostly Business Intelligence queries against column-organized tables by 10 to 50 times and improves compression by 3 to 10 times, compared to traditional row-organized tables. DB2 BLU achieves this with no reliance on indexes, materialized views, or other forms of "tuning". Instead, it exploits order-preserving, frequency-based dictionary compression to perform most SQL operations – predicate application (even range and IN-list predicates), joins, and grouping – on the compressed values, which can be packed bit-aligned so densely that multiple values fit in a register and can be processed simultaneously via SIMD (single-instruction, multiple-data) instructions.

BLU incorporates the second generation of Blink technology that originated in IBM Research - Almaden in 2006. [12, 18]. The first generation of Blink [4, 5, 18], was a strictly main-memory DBMS that was highly tuned for mod-

ern multi-core hardware. This first generation has already been incorporated into two IBM main-memory DBMS accelerator products [4]: the IBM Smart Analytics Optimizer for DB2 for z/OS (the mainframe DB2), which was released in November 2010, and the Informix Warehouse Accelerator, released in March 2011.

The second generation builds upon the main-memory efficiencies of the first generation, but isn't limited to databases that fit in memory. As in the first generation, DB2 BLU's hardware-conscious algorithms are carefully engineered to maximize parallelism, with novel data structures that require little latching, and to minimize data-cache and instruction-cache misses. Join and group-by operate directly on encoded data, and exploit DB2 BLU's new columnar data layout to reduce data movement for tasks such as partitioning. However, DB2 BLU is not limited to databases that fit in main memory – tables are stored on disk and intermediate results may spill to disk. Automatically-created fine-grained synopses, late materialization, and a new probabilistic buffer pool protocol for scans minimize disk I/Os, while aggressive pre-fetching reduces I/O stalls. The first generation of Blink stored data in PAX format [2], whereas DB2 BLU is a *pure* column store by default that stores each column separately (though it allows storing highly-related columns together in column groups; for example, NULLable columns can form a column group in which one column indicates whether each value is NULL or not). DB2 BLU uses a novel columnar data layout in which all columns are logically in the same row order, but within a page the ordering physically clusters values by their compression format, and so may differ for each column.

DB2 BLU supports UPDATE, INSERT, and DELETE operations, and adapts to new values not in the initial dictionary for a column by adaptively augmenting it with page-level dictionaries. Both IBM products that were based upon the first generation of Blink - the IBM Smart Analytics Optimizer for DB2 for z/OS and the Informix Warehouse Accelerator - were main-memory accelerator products that contained *copies* of the base data that was stored in a conventional row store. Column-organized tables in DB2 BLU contain the *only instance* of the data, eliminating the need to synchronize the copy in the accelerator with the base copy.

This second generation of Blink has been enhanced and deeply integrated with DB2 for LUW by a team of developers from DB2 Development and the IBM Systems Optimization Competency Center, and researchers from IBM Almaden. Full integration with DB2 ensures that column-organized tables benefit from the full functionality (including sophisticated query rewriting and optimization, prefetching, buffer pool, etc.) and robust utilities (including Load, automatic Workload Management, Backup, Recovery, etc.) of a mature product, while still enjoying order-of-magnitude performance gains from DB2 BLU's state-of-the-art run time. It also facilitates seamless evolution: column-organized and row-organized tables can co-exist in the same tablespace and even within the same query, and a new utility eases migration between the two.

The rest of this paper is organized as follows. Section 3 describes how data is compressed and stored in DB2 BLU. Then we cover query processing, beginning with an overview and important infrastructure in Section 4, followed by details on scans in Section 5, joins in Section 6, and grouping in Section 7. Section 8 covers how data is modified

in DB2 BLU, while Section 9 focuses on workload management. Section 10 contains a brief evaluation of DB2 BLU's performance and compression. We compare DB2 BLU to related work in Section 11, and conclude with Section 12.

3. DATA LAYOUT AND COMPRESSION

In this section we describe the highly compressed column-major storage that is implemented in DB2 BLU. All columnar data is stored in traditional DB2 storage spaces, and cached in DB2 bufferpools, allowing row and columnar tables to co-exist and share resources.

The compression scheme used in DB2 BLU represents a careful balance between storage efficiency and query performance. The compression encoding enables efficient query evaluation directly on the compressed data while simultaneously exploiting skew in the value distribution and local clustering in the data to achieve excellent compression.

3.1 Frequency Compression

DB2 BLU employs column-level dictionary compression that exploits skew in the data distribution by using up to a few different compressed code sizes per column. The most frequent values are assigned the smallest codes while less frequent values are assigned larger codes. The frequency compression used in DB2 BLU is similar to the Frequency Partitioning of [18] but adapted to column-major storage. Example 1 describes a column for which there are three different code sizes.

Example 1: In a DB2 BLU table that contains records of world economic activity, the compression dictionary for the CountryCode column, a 16-bit integer, may have the following partitions:

1. A dictionary containing the values for the two countries that occur most frequently in the sample data. The compressed code size for this partition is 1 bit.
2. A dictionary containing the next 8 most frequently occurring countries, with a 3-bit compressed code size.
3. An offset-coded partition for the remaining countries of the sample, with an 8-bit code.

The approximate compression ratio for the 3 partitions is 16X, 5.25X (details in Section 3.3), and 2X respectively. □

A sample of the data in a table is analyzed to determine the compression scheme that is applied to each of its columns. The sample is either (a) drawn from the data provided to the first bulk load into the table or (b) in the case that the table is populated by SQL INSERT statements, the sample is the data in the table when the table reaches a given size threshold. Histograms of the values in each column are collected from the sample, and a compression optimizer determines how to best partition the column values by frequency to maximize compression. A dictionary is created for each resulting column partition.

Entries of a given dictionary partition are one of (1) full data values, (2) common bit prefixes of data values, or (3) base values for offset coding, in which a code for a value is the value minus the base value. For (1), the code size for the partition is the number of bits required to record the index of the dictionary value. For (2) and (3), the code size is the number of bits required for the dictionary indexes plus the bits needed to store the suffix in the case of (2) or to store the offset from the base values in the case of (3). Within each
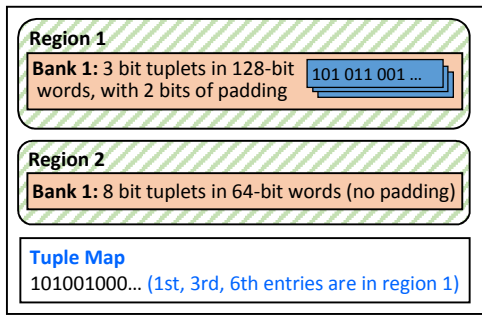
Figure 1: Page of CountryCode with two Regions.

Figure 2: DB2 BLU's page format.

partition, the entries are ordered by value so that the compression encoding is fully order-preserving, which enables arbitrary range predicates to be evaluated directly on the compressed data. Furthermore, the dictionary partitions of a column are ordered by increasing code size. Value encoding is prioritized such that a value will be encoded using the first partition, and with the shortest possible code size, in which the value may be represented.

3.2 Column Group Storage

Columns of a DB2 BLU table are partitioned into column groups, wherein each column belongs to exactly one column group. The DB2 BLU storage layer supports arbitrary numbers of columns per column group. Internally, nullability is in a separate null indicator column, so nullable columns define a column group containing at least two columns: the nullable column and its internal null indicator.

Column group data are stored in fixed-size pages. A larger unit of contiguous storage, called an extent, which contains a fixed number of pages for a given table, is the unit of allocation for storage. Each extent contains the data of one column group only. We call the projection of a row/tuple onto a column group a *Tuplet* (a "mini-tuple"). A tuplet for the column group of a nullable column thus contains at least two column values. Tuples are stored in the same order across all column groups. Tuples are identified by a virtual identifier called a *TSN* (Tuple Sequence Number), an integer that may be used to locate a given tuple in all of its column groups. A page contains a single range of TSNs, as recorded by a (StartTSN, TupletCount) pair in the page header. A *Page Map* records, for each page, the column group to which it belongs and the StartTSN of that page. The page map is implemented as a B+tree and is used at query time to find pages that contain a range of TSNs for a column group.

3.3 Page Format: Regions, Banks, Tuple Map

The cross product of the dictionary partitions of all columns of a column group determine the possible formats (combinations of column sizes) of the tuplets of a column group. We call the resultant combinations *Cells*. In the common case of a single-column column group, a cell is equivalent to a dictionary partition. All cells may occur in a given page of a column group, because tuple sequence is determined by the order in which data are loaded or inserted, which is independent of how the tuple is encoded for compression. Within any page, tuplets that belong to the same cell/partition and that have the same format are stored together in what is called a *Region*. This is a partitioning of the tuplets within
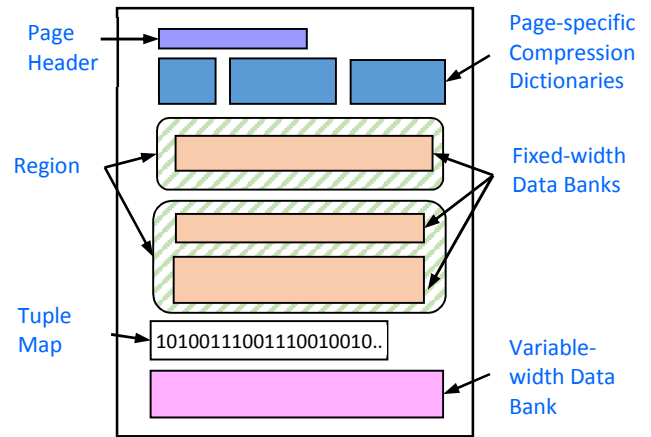
the page according to the cells to which they belong. For our CountryCode column example, if tuplets for its partitions 1 and 2 exist in a given page, then that page will contain two regions, one containing 1-bit tuplets and the other containing 3-bit tuplets. Pages containing more than one region will have a *Tuple Map* that records to which region each tuple was assigned by the encoding. The index of the TupleMap is a page-relative TSN (the row's TSN - StartTSN for that page) and the entry is the index of the region to which the TSN belongs. In a two region page, the TupleMap entries are 1 bit each. The intra-page partitioning into regions introduces a query processing challenge to return back TSN order. Section 5 describes this in detail.

Regions are further sub-divided into *Banks*, which are contiguous areas of the page that contain tuplet values. Each column of a tuplet may be stored in a separate bank. In the case of a column group for a nullable column, the column and its null indicator may be stored in two separate banks of its regions. Most banks contain fixed-size tuplets. For compressed values, the encoded tuplets are stored packed together in 128-bit or 256-bit words with padding to avoid straddling of tuplets across word boundaries. We call the 128/256-bit word size the *width* of the bank. For example, the 3-bit column codes of partition 2 of the CountryCode column will be stored in a 128-bit wide bank in which each word contains 42 encoded values, with the remaining 2 bits left unused as padding bits to prevent straddling word boundaries. This is shown in Figure 1, which depicts the content of a page of the CountryCode column that has regions for partitions 2 and 3 of its dictionary.

We may need to store uncompressed values that are not covered by the compression scheme. Fixed-width uncoded values are also stored in banks of regions where the bank width is determined by the data type of the columns. Uncoded variable-width values, for example of VARCHAR columns, are stored one after another in a separate variable-width data bank of the page that is outside of the regions. A descriptor column for each variable-length column, comprised of an (offset, length) pair, is stored in a regular region bank to record the location of each variable-length data value.

By grouping like-formatted tuplets together into regions, we end up with long runs of tuplets that have the same format, which is important to query performance.

3.4 Page Compression

Additional compression is performed at the page level to exploit local clustering of data values within the table and to compress values not covered by the column-level compression scheme. For example, if only 3 of the countries of CountryCode partition 2 occur in a page, we can reduce the code size for tuplets of that region to 2 bits from 3 by specializing the compression scheme to the page. Page compression is tried for all pages of DB2 BLU tables, and when it is beneficial, mini-dictionaries are stored within the page to record the page-level compression scheme.

Example types of dictionary specialization that are considered at the page level include:

1. Reducing the number of dictionary entries to only those that appear on the page. The smaller dictionary is stored in the page and contains a mapping from the column-level codes to those on the page. This page dictionary only contains the codes of the column-level column dictionary, not the data values, and as such, the page dictionary benefits from the column-level compression.
2. Reducing the number of bits required to compute offsets for offset-coding, given the range of values that occur in the page.

Figure 2 depicts the DB2 BLU page format. There is a small page header. Next come the page compression dictionaries if applicable. The regions follow, each of which contains some number of fixed-width banks. The TupleMap is next and finally we have the variable-width data bank.

3.5 Synopsis Tables

An internal synopsis table is automatically created and maintained for every column-organized DB2 BLU table. This synopsis table reduces scan times of the user-defined table by enabling page skipping, given predicates on the natural clustering columns. Besides summary information about the values in each column, the synopsis contains a (MinTSN, MaxTSN) pair of columns that specify the range of DB2 BLU table rows represented by each row of the synopsis. The synopsis tables use the same storage format as regular DB2 BLU tables and inherit the compression scheme of the corresponding user-defined table.

3.6 Global Coding

In addition to providing a partition-relative encoding for each value of a DB2 BLU table, the compression dictionary of a table also provides a partition-independent encoding that we call global coding. The size of the global codes for a column is determined by the sum of the code counts over all partitions of the column dictionary. A global code G is computed from a partition-relative code C simply by adding to C the sum of code counts over all partitions that precede the partition to which code C belongs. Global codes are used in join and group by processing on compressed data.

4. QUERY EXECUTION

This section gives an overview of query execution, and some of the infrastructure that supports it.

4.1 Overview of Query Processing

SQL queries that are submitted to DB2 go through the usual DB2 query compilation, including optimization that
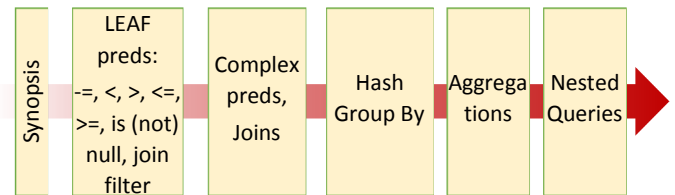

Figure 3: Typical operation sequence. Many operations in DB2 BLU operate on compressed data, achieving better cache utilization and higher memory-to-CPU effective bandwidth.

produces a plan tree of operators. As a result DB2 BLU benefits fully from DB2's industry-leading, cost-based optimizer and query rewrite engine. The DB2 query optimizer identifies portions of the query plan that reference column-organized tables and that can leverage columnar processing, and prepares a special execution plan for each that is composed of pieces called *evaluator chains*. Evaluator chains are chains of DB2 BLU operators, called *evaluators*, each of which performs an action such as loading a column's values, performing an arithmetic operation, a join, etc.. Roughly, there is one evaluator chain per *single-table query (STQ)*, i.e., access to a table, as runtime utilizes a decomposition of a multi-table query into a sequence of single table queries. These evaluator chains are prepared by DB2 BLU as follows:

1. Join queries are restructured into a list of STQs.
2. Predicates are restructured to optimize run-time behavior of evaluators. This includes ordering predicates by cost of evaluation, folding predicates into fewer when possible without a semantic change, and adding inexpensive but possibly redundant predicates before more expensive predicates to attempt a coarse pre-filtration (e.g. adding min-max predicates before an in-list predicate).
3. Predicates on a column are used to construct predicates on synopsis columns. The synopsis, discussed below, contains aggregated metadata such as minimum and maximum values for a range of rows. Transforming predicates on the original table into additional predicates on the synopsis table, which is orders of magnitude smaller and often kept in memory, can be very effective at reducing data-page accesses during query processing.
4. Scalar sub-queries and common sub-expressions are restructured into separate queries whose results are fed to consuming queries. These sub-queries are then rewritten to an executable form, as given above.

Figure 3 shows the typical operation sequence of an STQ in DB2 BLU. A scheduler starts a set of threads to execute each STQ, each thread running this same evaluator chain. An STQ typically starts by scanning a "leaf" column in the plan tree and applying predicates to it. Additional columns may be fetched, applying any predicates, and the results are typically piped into an evaluator that builds a hash table. A subsequent STQ over the join's outer (fact) table will probe this hash table (and usually all other dimensions), and the surviving rows will typically be piped into an evaluator that builds another hash table to accomplish grouping and aggregation. This entire sequence can be short-circuited if joins or grouping are not specified, of course, and it can be nested for more complex queries.
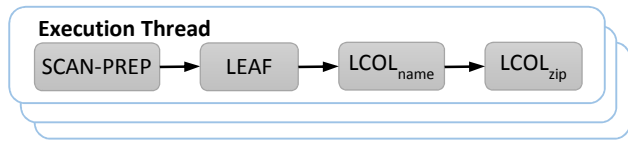
Figure 4: Per-thread evaluation chain that processes strides of rows introduced by SCAN-PREP.
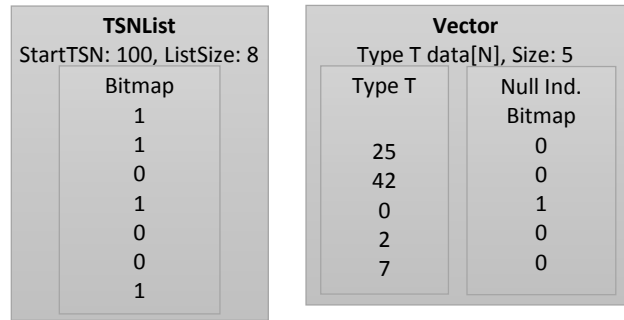

Figure 5: A TSNList for 8 rows (TSNs 100 .. 107) where tuples 102, 104, and 105 have been filtered out due to predicates; and a Vector, whose size (5) matches the number of valid TSNs in the TSNList.

The evaluators in the chain do not work on a single value or a single row; they instead operate on a set of rows called a *stride*. By processing a stride at a time, DB2 BLU amortizes the overhead in function calls, avoids cache-line misses, maximizes opportunities for loop unrolling, and creates units of work that can be load balanced among threads by "work stealing". A stride typically contains on the order of thousands of rows, but its exact size is chosen so that the evaluator's working set fits in the processor's cache, so is dependent on the query, machine, and available resources.

To illustrate this more concretely, consider a very simple example, a single-table query asking for the name and zip code of employees from California:

`SELECT name, zip FROM employees WHERE state=''CA''`

The execution of this query requires an evaluator chain with several steps, illustrated in Figure 4:

1. A SCAN-PREP evaluator, which segments the input table, employees, into strides to be processed, applies synopsis predicates to skip ranges of TSNs, and initiates the processing of each stride into the evaluator chain.
2. A LEAF evaluator, which accesses the page for that stride of the state column, possibly causing it to be retrieved from disk, and applies the local predicate on it.
3. Two LCOL (load column) evaluators, each of which retrieve the values for the name and zip columns, for only those rows that qualified the local predicate.

This simple query was chosen to give a flavor of DB2 BLU and ignores many details, such as the impact of rows introduced by updates (depending upon the isolation level specified by the query), how page prefetching is performed to hide I/O latency, the exploitation of in-memory synopses to skip pages all of whose values are disqualified by a predicate, etc. The next sections discuss DB2 BLU in more detail, starting with the common infrastructure.

4.2 Infrastructure for Columnar Processing

To understand query processing in DB2 BLU, it is important to define some infrastructure used throughout columnar processing in DB2 BLU. These are used for holding row selection state as well as holding results.

TSNList. A *Tuple Sequence Number (TSN)* is a logical row identifier that DB2 BLU uses to "stitch together" the values from different columns for a row. The TSN is not explicitly stored with each entry of a column, but maintained only in the page map, as discussed in Section 3.2.

Since DB2 BLU processes strides of TSNs, a *TSNList* is used to track the TSN values expressed in the set. The implementation of a TSNList, shown in Figure 5 (left), is a starting TSN value (64-bit unsigned integer) and a bitmap as long as the stride, with 1 bits for *valid* TSNs and 0 bits for *invalid* TSNs, i.e., TSNs that have been disqualified by a

previous predicate. Keeping this *passing flag* in a bitmap allows for quick population-count operations that scan for the next 1 bit, and improves cache efficiency and avoids memory movement by not having to rearrange a list of TSNs.

Vector. Most processing in DB2 BLU deals with columns, of course, so we often need an efficient way to represent sets of values for a column or result. These sets are stored in *Vectors*, as shown in Figure 5 (right). Vectors leverage C++ templates to provide efficient storage for all supported SQL data types. Vectors also maintain additional metadata: a bitmap indicating NULL values, so no "out-of-band" value is required, and a sparse array to record and return to the client application any warnings or errors that affect any row.

Work Unit. The association of values for particular TSNs is maintained in a unit of work, or *WorkUnit*. This structure is a list of Vectors of values (or codes), along with one TSNList, and is the smallest work package that passes between evaluators in an evaluator chain.

Compaction. Vectors in a WorkUnit are self-compacting. As predicates in the evaluator chain are applied, they disqualify tuples and update the TSNList bitmap with 0-bits. Vectors that were populated before those predicates were applied still contain entries for those non-qualifying tuples. When Vectors are retrieved from a WorkUnit, if the "current" filter state differs from the filter state when the Vector was generated, then the Vector is compacted. Compaction removes all values for tuples that have been disqualified, so that the Vector is tightly packed with only values for "valid" rows.

Evaluator Framework. As stated above, all evaluators process sets of rows, called *strides*. Both TSNLists and WorkUnits correspond to strides. For example, an evaluator that loads a column would take as input just a list of TSNs but would produce a Vector of values from the column for the selected TSNs. An evaluator that performs addition would accept two input Vectors of numeric values and produce a third output Vector.

Multi-threading in DB2 BLU is achieved by cloning the evaluator chain once per thread, with the number of threads being chosen based upon cardinality estimates, system resources, and/or system workload. Each thread requests a unique stride of TSNs for the table being processed as the
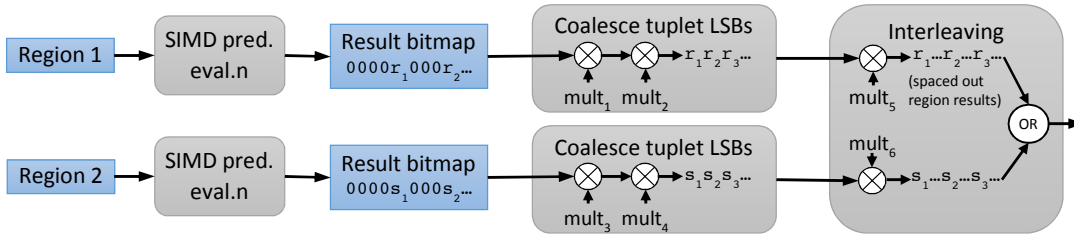
Figure 6: Leaf evaluation on a 2-region page. $mult_i$ **are precomputed multipliers and** r_j, s_k **are result bits.**

first step in its evaluation chain, and continues requesting strides until no more TSNs are available.

5. SCANS AND PREDICATE EVALUATION

Columns are accessed via two access methods:

LeafPredicateEvaluator (henceforth, LEAF): A LEAF applies predicates on one or more columns within a column group: e.g., (COL3 <5 and COL3 > 2) or COL2=4. In each invocation, LEAF accepts a TSNList (usually corresponding to an input stride), and evaluates its predicate over all the TSNs in its range (not just the valid ones). Nullability is handled as part of predicate evaluation.

LoadColumnEvaluator (henceforth, LCOL): An LCOL scans values from a single column, along with its nullability, in either encoded or unencoded form. It accepts a TSNList (a range of TSNs and a bitmap indicating which TSNs from that range need to be loaded), and outputs a Vector holding values or codes for that column, as desired.

5.1 Leaf Predicates

Recall the page structure from Figure 2. LEAFs are applied within each region of each page, and only the predicate output is interleaved across regions. Of course the same predicate can also be applied by loading the column (via LCOL) and then applying a comparison evaluator. This distinction loosely resembles that of SARGable vs non-SARGable predicates in System R [19]. Predicates are applied via LEAF as much as possible, because it is more efficient to apply them on the compressed, tightly packed bank format.

Returning to Example 1, we separately apply the predicate on the 1-bit dictionary coded, 3-bit dictionary coded, and 8-bit offset-coded regions, whichever are present on a page. The result from each region is a bitmap saying which tuples in the region satisfied the predicate, plus a bitmap for nullability [1]. These bitmaps are then interleaved using the TupleMap to form a bitmap in TSN order, as shown in Figure 6.

5.1.1 Applying predicates on each region

Evaluation of predicates of the form *Column <op> Literal* within each region is amenable to SIMD-ized processing for many operators: $<$, $>$, $=$, $<>$, $<=$, $>=$, BETWEEN, small IN-lists, NOT IN-lists, and IS (NOT) NULL. Over unencoded regions, this is straightforward SIMD processing. Over encoded regions, we use the technique described in [14] that takes a bank and produces a result that is also in bank

format, with the least significant bits (*LSBs*) of each tuplet denoting whether the predicate was satisfied.

We then "coalesce" these LSB bits and append them to a bit vector of outputs, using multiplication. For example, to coalesce from 11-bit tuplets, say: $xxxxxxxxxx1xxxxxxxxxx0xxxxxxxxxx1$ (where x stands for "don't care"), into 101.... we multiply by $2^{10} + 2^{20} + 2^{30}$ – this shifts the 1st LSB by 10, the 2nd by 20, etc.

On POWER7^{TM}, we use specialized bit-permutation instructions. For example, on a 128-bit bank, we can evaluate $Literal1 \leq Column \leq Literal2$ predicates in about 12 instructions, *irrespective* of how many tuplets fit into the bank.

Longer IN-lists, Bloom filters, and LIKE predicates are also often evaluated as LEAFs. They are generally not evaluated in an SIMD fashion. Instead, we look up each (encoded) value in precomputed hash tables. We generally apply these complex predicates over the values in the dictionary and identify the qualifying codes, which we enter into a hash table. Predicates over multiple columns within a column-group, as well as complex single-column predicates, e.g. (Column > 5 OR Column IN list), can also be done as LEAFs.

5.1.2 Interleaving results across regions

After applying predicates on each region, we need to interleave the bitmap results using the TupleMap. For example, suppose that a page has two regions, and the result from the regions are *01001001001* and 10101110111, and that TSNs alternate between the regions (i.e., the TupleMap is 01010101010). Then the desired interleaved result is *0110010011010110010111*. The straightforward implementation would be to perform this one TSN at a time, paying for a branch per TSN.

An alternative scheme that we have found to be efficient is to multiply the result from each region by two precomputed multipliers, and then do a bitwise OR. For example, suppose that the TupleMap is 10101010..., and the predicate result from the first region is $b_1 b_2 b_3 b_4$. We perform two multiplications: the first converts $b_1 b_2 b_3 b_4$ into $b_1 xxxxxxx\ xb_2 xxxxxx\ xxb_3 xxxxx\ xxxb_4 xxxx$. We form the multiplier by summing the left-shift (multiplication) needed on each b_i. The second multiplication converts this result into $b_1 0 b_2 0 b_3 0 b_4 0$. Nullability is also interleaved similarly.

5.2 Load Column

LCOL is the main data access operator. It loads either column values or column global codes. The latter is used for key columns of joins and for group-by columns, because both are done over encoded values. The former is used for loading columns used in complex expressions (e.g., $COL2 + 5$), aggregation, etc.. The output of LCOL is always compacted,
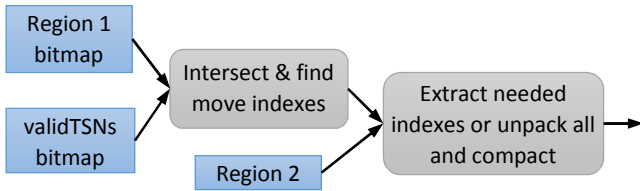
---

[1] Nullability cannot be folded into the predicate result because SQL uses 3-value logic for nulls [20]

Figure 7: Load column evaluation (per region).

i.e., it has only entries corresponding to valid TSNs of the TSNList, those that satisfy previous predicates. As with LEAF, LCOL also loads one region at a time, and then interleaves the results, as outlined in Figure 7.

5.2.1 Loading within each region

Within a region, LCOL involves extracting codes at desired offsets from packed banks, and decoding them if needed. When loading a small subset of the bank, we directly do masks and shifts to extract the codes. Converting from tuplet index in a bank to a bit position involves an integer division (by the number of tuplets per bank), but we precompute a modular inverse and do a multiplication with it instead. When loading most of the bank, we use templatized logic to unpack the entire bank into a word-aligned array, and then extract the needed codes.

5.2.2 Interleaving across regions

Interleaving of LCOL is more challenging than for LEAF because we are interleaving bigger quantities. We combine interleaving with compaction. This involves two bitmaps:
RegionBits: bitmap of tuples in TSN order that fall into this region (e.g., 10101.. if alternate tuples go to a region).
ValidBits: bitmap of tuples that have qualified all previous predicates and need to be loaded.

Using the TupleMap, we form a bitmap of the TSNs (on the page) that belong to each region. Then we use this bitmap along with the valid TSNs bitmap to directly move codes or values from the per-region result into the output Vector. Returning to the previous example, with a two-region page, suppose the TupleMap is 01010*1*01010, and that the validTSNs bitmap is 00100*1*0011*1*, i.e., we need to load the 3rd, 6th,7th, 8th, and 9th tuple within the page (in TSN order). The only values that we need to move out of the result from the second region are the ones in the intersection of these bitmaps, shown in bold italics: we need to move the 3rd entry from the region result to the 2nd position in the output, and the 5th entry from the region result to the 4th position in the output. These move indexes are calculated efficiently by repeatedly accessing all bits up to the right-most 1-bit in the intersection (via n XOR $(n-1)$), and using population count instructions thereafter.

6. JOINS

DB2 BLU supports a wide range of joins: inner joins, outer joins (left, right, full), and early joins (where the first match is returned for each row of the join outer). We also support anti-joins (inner, left, and right), which are joins with a not-equals predicate. There is no restriction that joins should be N:1 and no assumption that referential integrity or even key constraints have been specified on join columns.

Tables being joined need not fit in memory and will be spilled to disk as they exceed available memory heap space.

We employ a novel cache- and multicore-optimized hash join that exploits large memories when available, but gracefully spills to disk if needed. We use partitioning heavily – for latch-free parallelism, for improving cache locality, and for spilling to disk. Unlike traditional row-stores, DB2 BLU partitions only the join columns, leading to significant reduction in data movement, but requiring more sophisticated book-keeping of which payloads correspond to each key.

6.1 Build phase

We use a simple two-table join to illustrate the join algorithm. The query optimizer presents the runtime with a join operator that specifies one side as the build (*inner*) side of the hash join, and the other as the probe (*outer*). This choice is made according to a cost model, and generally favors smaller tables on the inner side, as well as N:1 joins over N:M joins, if known or deduced.

As Figure 8 shows, in the build phase we first scan and partition the join key columns and the join payloads from the inner table (usually a base table, but can be intermediate results) [2]. Each thread scans and partitions a separate subset of the inner rows. After partitioning, each thread takes ownership of a partition to build a hash table mapping each join key to its join payloads. The only cross-thread synchronization during this inner scan occurs between the partitioning and build phases. This inner scan employs late materialization and compaction, as with any regular scan. The number of partitions is chosen carefully so that each partition fits in a suitable level of the memory hierarchy, whichever is feasible.

Joins are performed over encoded data – both join keys and join payloads. This keeps the hash tables compact and fitting into low levels of on-chip cache. Join keys are encoded using the encoding dictionary (global code) of the *outer's* join column(s) (the foreign key columns). This means we convert the join columns from the inner into the encoding space of the outer during the scan of the inner. We choose this over the encoding space of the inner because inners generally have smaller cardinality than outers. At the same time, we also build Bloom filters on these foreign key column values, called *join filters*, for use during the probe phase. Join payloads are encoded using the dictionary of the payload columns when possible, and using a dynamically constructed dictionary for any values that were not encoded at load time.

DB2 BLU uses a novel compact hash table that uses an indirection bitmap to almost completely avoid hash table collisions while having no empty buckets. When possible, we also make the hash table completely collision-free by exploiting the encoded nature of the join keys.

6.2 Probe phase

For the probe phase, shown in Figure 9, the join columns are scanned and first passed through the join filters derived in the build phase. When an outer table is joined with multiple inner tables, join filters from all inners are first applied, compacting as we go, before any joins are performed. So the foreign keys that enter join operators have already been

---

[2] We use the term *join key*, but do not rely on uniqueness constraints; we infer uniqueness at runtime, after local predicates are applied.
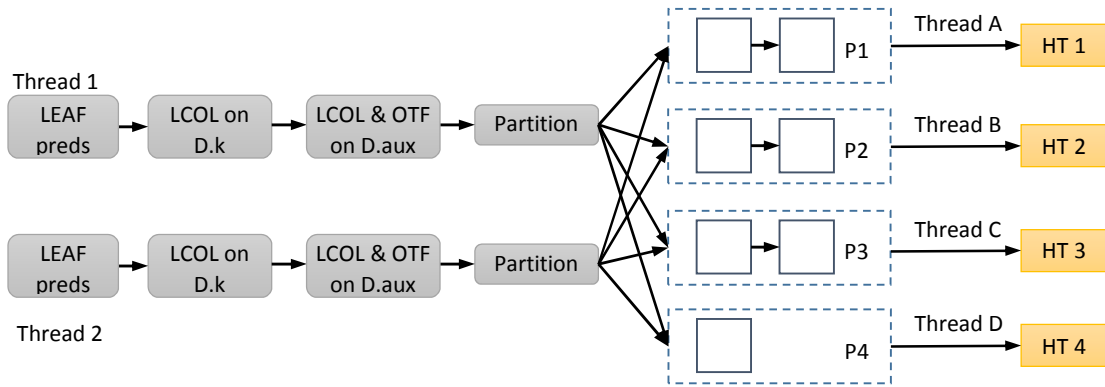
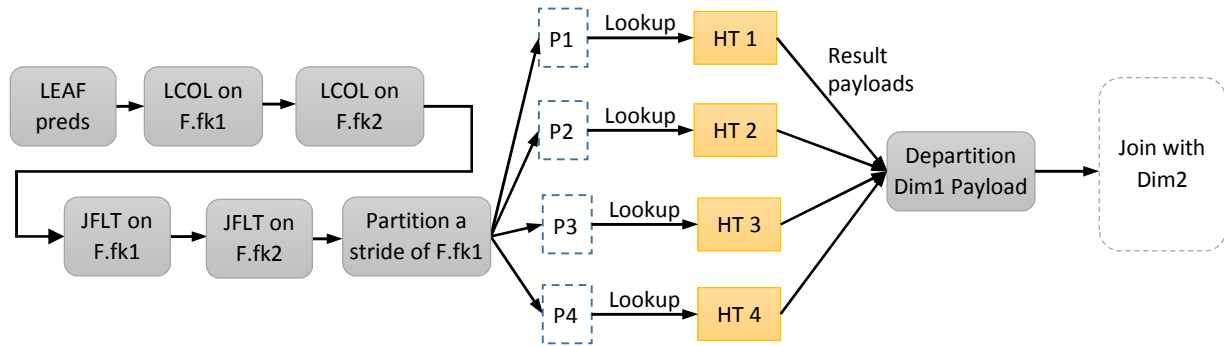**Figure 8: Evaluator sequence for build of join with 4 partitions.**

Figure 9: Evaluator sequence for probe of join. JFLT stands for join filter. Observe that join filters of all dimensions are applied before any joins are performed.

filtered with the combined selectivity of all join (and local) predicates, so we do not need deferred materialization of join payloads, as done in [1].

Foreign keys surviving the join filters next enter the join operators, where they are partitioned to probe against the partitioned hash tables from the inner. This partitioning is done in a pipelined fashion, and *only the join columns are partitioned*. A stride of join columns are scanned and partitioned. Then the foreign keys in each partition are probed against the corresponding inner hash table, and finally the result payloads are departitioned to bring them back into the original (TSN) order. By paying for the departitioning, we avoid having to partition (and early materializing) non-join columns such as measures.

6.3 Join Filters and Semi-join

We employ a hierarchy of join filters on the outer scan (on the concatenated foreign key columns) to reduce the number of rows that have to go through join processing. We also use min-max filters on the individual foreign key columns to exploit any clustering there may be between local predicates and key values. Sometimes the inner is very large, making the hash table build expensive and possibly causing it to spill to disk. When we detect this situation, we adopt a semi-join plan in which we do an extra scan on the foreign key column of the outer to determine if that (after applying predicates from other dimensions) can be used to filter rows of the large inner. If so, we build a Bloom filter on these

qualifying foreign keys, and use this to filter rows of the large inner before they are built into its join hash table.

6.4 Spilling

When all partitions of the inner will not fit in the available assigned memory, we employ one of two kinds of spilling to disk. One approach is to spill some partitions of both inner and outer, as in hybrid hash join [9]. An alternative approach picked in many cases (via a cost model) is to spill only the inner. Even though these partitions may have to be spilled to disk and brought back to memory once per stride of the outer, this helps to avoid early materialization of non-join columns from the outer, and allows us to easily adapt to changing memory availability during join execution.

7. GROUPING AND AGGREGATION

In general, DB2 BLU employs a partitioned hash-based implementation of grouping and aggregation. It supports the usual aggregation operations and handles DISTINCT. The design of the grouping and aggregation component is dictated by two main challenges. The first challenge is the very wide spectrum of number of groups for the queries DB2 BLU has to deal with. Even from a single customer we get queries whose grouping output varies from less than ten to hundreds of millions of groups. Thus, the first requirement was robustness with respect to the number of output groups. The second challenge is the hierarchical configuration of modern multicore multisocket database servers that provide multiple levels in the memory hierarchy and expose
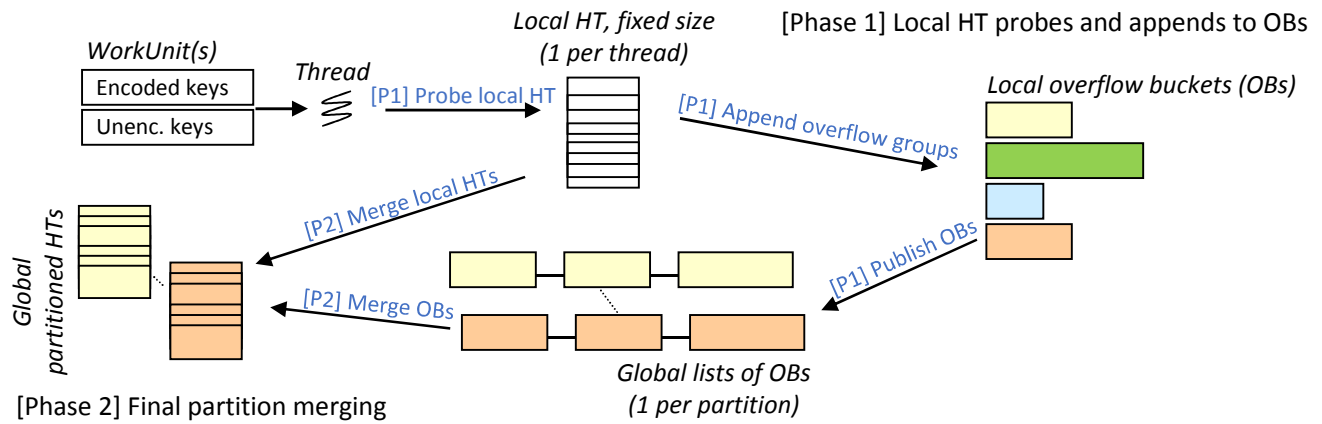
Figure 10: Overview of GROUP BY processing in DB2 BLU with two phases of local and global processing.

non-uniform memory access (NUMA) effects [3, 16]. Thus, the second requirement was robustness to, and exploitation of, the multi-level memories of modern hardware.

For queries that output a small number of groups, we want the majority of the processing to take place in thread-local data structures that reside in on-chip caches. For queries that output a large number of groups, we want to be able to combine the groups generated by all threads in a fashion that incurs minimal, if any, synchronization.

7.1 Overview

At a high level, and as shown in Figure 10, the grouping operation in DB2 BLU takes place in two phases: initially each thread performs a local grouping on local hash tables of fixed size, buffering to overflow buckets all groups that do not fit in the local hash tables; then, in a second phase, each thread picks a partition and merges the corresponding entries of each local hash table and all the overflow buckets for this partition.

The main data structure used is a hash table that uses linear probing. In queries that output a small number of groups, the majority of processing takes place in updating the local hash tables that are of fixed size and fit in on-chip caches. In queries that output a large number of groups, the majority of the time is spent in merging to the global hash tables. Both operations require no synchronization and are thread-local. This design achieves good performance irrespective of the number of groups in the query (first requirement). At the same time, it exhibits good memory locality that exploits modern memory hierarchies (second requirement).

7.2 Normal (local) processing

In the first phase, all threads operate on *thread-local hash tables* of fixed size. Each thread acts independently, updating its local hash table, updating or inserting new groups. The size of the local hash tables is determined by the size of the caches, the number of threads working on the query, as well as the width of both the (compressed) grouping keys and payloads.

When the local hash tables get full, any new groups that do not already exist in the local hash table are appended to a set of buffers called *overflow buckets*. Each thread appends to as many overflow buckets as the number of partitions. The number of partitions depends on the number of threads

working on the query and the available memory. The overflow buckets have fixed size. When they get full, the thread *publishes* them by appending them into a global linked list of overflow buckets for that partition. In order to increase the hit ratio of the local hash tables and reduce the number of overflow buckets, each thread maintains statistics for each local hash table and performs a reorganization by dumping the least frequent groups or periodically dumping all the contents of the local hash table so that new, possibly more frequent, groups enter the local hash tables.

At the end of the first phase, each thread publishes any not full overflow buckets and its local hash table. The entire processing of the first phase requires no synchronization.

7.3 Merging to global hash tables

In the second phase, each thread reserves, by atomically increasing a counter, a partition to merge. Then it merges all the entries of *all* local hash tables that belong to that partition plus all the overflow buckets. The thread continues this process until there are no other partitions to be merged. All the groups of a partition are merged to a *global partitioned hash table*, which is allocated to a socket local to the merging thread and, in contrast to the local hash tables, does not have fixed size. In order for a merging thread to identify the entries of each local hash table that belong to a specific partition, we store with each local hash table a packed array of partition identifiers. All the processing in the second phase is again done in a latch-free fashion.

As all the incoming grouping keys are touched once for the needs of the local processing in the first phase, we collect statistics that can facilitate the processing of the second phase. Such statistics include an estimation of the number of distinct groups of each partition, so that we allocate appropriately-sized global hash tables for each partition. If the free space in the global hash table gets low and there are still overflow buckets for this partition that need to be merged, we allocate a new hash table of bigger size and transfer the contents of the previous table. Because of the overhead of this operation, we try to estimate as accurately as possible the final size of a global hash table.

7.4 Early Merging & Spilling

If during the normal (local) processing the system runs low on memory, for example, because it has created many overflow buckets, then some threads stop their normal pro-

cessing and pick a partition to *early merge*. When a thread early merges a partition, it marks the partition as being early merged, and processes all the overflow buckets for that partition that have been published up to that point. Early merging frees up space from the overflow buckets processed thus far. If even with early merging the overflow buckets do not fit in the available memory for the grouping, then we must spill them to disk. We pick chains of published overflow buckets and spill them to disk, marking the corresponding partition as less desirable to be early merged, because early merging that partition would require bringing its overflow buckets from disk, incurring additional I/O. Early merging and spilling occur in queries with a very large number of groups. This design allows DB2 BLU to robustly process even customer queries that output several hundreds of millions of groups with wide grouping keys and payloads.

8. INSERTS/DELETES/UPDATES

In addition to having a high performance bulk data load utility, DB2 BLU fully supports SQL INSERT, DELETE, and UPDATE statements, as well as the multi-threaded continuous INGEST utility. DB2 BLU is a multi-versioned data store in which deletes are logical operations that retain the old version rows and updates create new versions. Multi-versioning enables DB2 BLU to support standard SQL isolation levels with minimal row locking. Furthermore, update-in-place is infeasible in any highly compressed data store (such as DB2 BLU). Write-ahead logging is performed for all modifications to column-organized tables and DB2 BLU leverages the existing robust facilities of DB2 for database resiliency and recovery.

8.1 Inserts: Buffering and Parallelism

Achieving good performance for inserts and updates in a column-store system is challenging because a page for each column must be modified for each row insert. The two major costs are (1) latching in the buffer pool a page for each column and (2) logging the changes to each of these pages. In DB2 BLU, we have focused on providing good performance for large insert transactions, which for example, insert on the order of 1000 rows each. Such transactions are the norm in the data warehouse systems initially targeted by DB2 BLU.

Inserts in DB2 BLU are buffered to amortize the main costs over many rows. A mirror copy of the last, partially-filled page of each column group is maintained in memory outside of the buffer pool. Inserts of a transaction are applied to these buffers, but these changes are not reflected in the buffer pool or log stream until either (a) the buffer page becomes full or (b) the transaction commits. At these events, the buffer is flushed, causing its new entries to be logged and copied back to the buffer pool page. Physical logging is performed for the changes and, for large transactions, the resulting log records will be large, containing many values. This effectively amortizes the per-page log record header overhead that can dominate log space in the presence of small insert transactions.

The insert buffers have the same format as DB2 BLU pages, which enables queries to operate directly on the buffers, as needed such that transactions are able to query their own uncommitted changes.

DB2 BLU supports a high degree of insert and update parallelism by dividing the TSN space of a table into many non-overlapping partitions called Insert Ranges. On per-forming its first insert, a transaction locks an insert range of the target table, obtaining exclusive access to the range. From that point on, the transaction is fully free, without any further synchronization, to consume TSNs of the insert range and insert rows into the tail pages of the insert range.

8.2 Deletes, Updates, Space Reclamation

DB2 BLU tables contain a 2-bit wide internal TupleState column that records the current state of each row. A row is in the Inserted state when first created by a LOAD, INSERT, or UPDATE operation. Deleting a row simply updates its state to PendingDelete. The change to the TupleState column is logged and multiple deletes to the same page from a single transaction are recorded in one log record. PendingDelete state rows are cleaned in a lazy fashion to the Delete state, which indicates that the delete is committed. A REORG utility is provided to free up pages for reuse at the extent level for those extents in which all rows are in the Delete state. The REORG utility can be invoked by the user directly, or it can be set to run automatically using the automatic reorganization feature.

Updates are processed internally by DB2 BLU as a combination of a delete of the qualifying rows and insert of the new versions of these rows that result from their update. DB2 BLU tables contain a second internal column, called PrevRowTSN, that links a new version row resulting from an update back to the original, root version of the row by TSN. Compression eliminates the storage of this column for rows that are created by inserts, so PrevRowTSN is only materialized in the case of updates. The new version rows that are created by an update are inserted into the insert range that is locked by the transaction, so that parallelism is supported for both updates and inserts.

8.3 Locking and Concurrency

DB2 BLU supports both the Uncommitted Read and Cursor Stability isolation levels, in which readers do not block writers and writers do not block readers. A full description of how this is implemented is beyond the scope of this paper, but a few of the main ideas are introduced here. Row locking is performed for those rows modified by INSERT, DELETE, and UPDATE transactions. Multi-versioning enables DB2 BLU to minimize locking for queries. A high-water mark TSN (HWM) is maintained for each insert range that records the TSN up to which all inserts are committed. The HWM enables queries to bypass locking for all Inserted state rows that are below the HWM. Locking is also not required for any rows in the Deleted state, and standard CommitLSN tricks are used to avoid locks in pages known to contain only committed data [17]. Finally, the PrevRowTSN column is used to ensure that queries return a single version of each qualifying row.

9. AUTOMATIC WORKLOAD MANAGEMENT

Another consideration for the DB2 BLU technology is how it deals with the division of machine resources when faced with multiple queries of varying degrees of complexity submitted simultaneously. BLU queries can execute significantly faster if they are processed completely in-memory without the need to spill partial results to disk. The number of queries that DB2 BLU attempts to process in parallel is one of the key determinants of whether query execution can proceed completely in memory.

Table 1: Speedups using DB2 BLU against an optimized row-organized system with optimal indexing.

Workload	Speedup with DB2 BLU
Analytic ISV	37.4x
Large European Bank	21.8x
BI Vendor (reporting)	124x
BI Vendor (aggregate)	6.1x
Food manufacturer	9.2x
Investment Bank	36.9x

In order to exploit the fact that better overall performance can be achieved by subdividing machine resources between fewer queries at a time, DB2 BLU incorporates an admission control mechanism to limit the number of queries that are allowed to execute concurrently on the database server. Any read-only queries submitted to the system are categorized based on the query cost estimate generated by DB2's cost-based optimizer into a managed and unmanaged query class. Lightweight queries that are below a defined cost floor (and non read-only activities) are categorized as unmanaged and are allowed to enter the system with no admission control. This avoids a common problem with queueing approaches in which the response time of small tasks that require modest resources can suffer disproportionately when queued behind larger, more expensive tasks.

For heavyweight queries above the cost floor, we categorize these as managed, and apply an admission control algorithm that operates based on the CPU parallelism of the underlying hardware. Based on the hardware, once a certain number of managed queries are executing on the server, further submitted queries will be forced to queue and wait to begin execution until one of the currently executing queries has completed and exits the system. This is done completely transparently to the user, who only sees the resulting performance benefit from a reduction in spilling and resource contention amongst queries.

10. RESULTS

While the improvements in performance provided by DB2 BLU vary by server, workload, and data characteristics, speed-ups of one or up to two orders of magnitude over optimized row-store systems are common. Table 1 shows the performance speed-up of DB2 BLU's code for a set of customer workloads compared on the same server against traditional row-organized processing with optimal indexing. Speedups between a modest 6.1x and more impressive 124x can be achieved for some workloads. What is really exciting is that **no tuning** was required to achieve these results.

Overall performance is often skewed by outliers, those few queries that seem to run much longer than all the others. DB2 BLU actually provides the biggest speed boost to these queries having the longest execution times. We found that DB2 BLU's performance is commonly at least three times more consistent (less variable) than traditional BI systems, because access plans for column-organized tables are simplified. In addition, although DB2 BLU is in-memory optimized, it is not main memory-limited, and performance does not fall over a cliff as data size grows beyond RAM.

DB2 BLU also dramatically reduces storage requirements, not only because it does not use secondary indexes or MQTs, but also because of its order-preserving compression. Figure 11
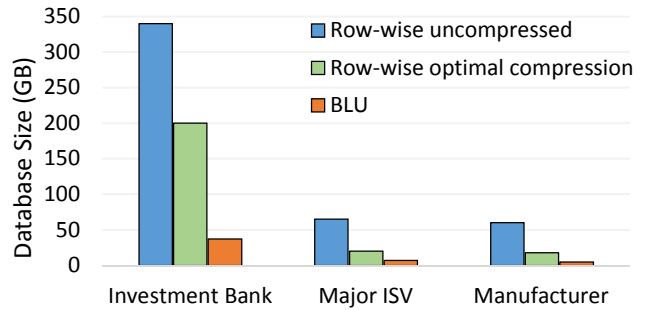

Figure 11: DB2 BLU dramatically reduces storage requirements for analytics databases, typically by 10x compared to uncompressed data in traditional (row-organized) databases.

shows the storage requirements for storing the databases of three different customers when in uncompressed row-organized, compressed row-organized, and DB2 BLU's column-organized formats. While storage savings are data dependent, we have observed savings over uncompressed data averaging around 10x for most customer workloads.

The overall result is a system that simultaneously has the full functionality and robustness of a mature product (DB2 LUW) while being in-memory optimized, CPU-optimized, and I/O-optimized; having less storage requirements; and requiring virtually no tuning to achieve all these.

11. RELATED WORK

Column-organized tables date from 1985 [8], and products that organize data this way are also not new. Sybase IQ has been around since 1994, and since 1997 the EVI feature of DB2 for i5/OS [6] allows some columns to be stored column-major and dictionary encoded. Indexes and index-only access are also long established, although every index is stored in a different order, making them impractical for the scans of BI queries that need to touch many columns. But the MonetDB [7, 23] and later the C-store [21] research prototypes revived research interest in column stores for BI workloads, with an emphasis on columnar query execution, not just storage. They led to a flurry of new columnar start-up products, including the Vertica product, now marketed by HP, that was derived from C-Store. One feature that distinguishes Vertica is the ability to define projections, which, like indexes in row stores, contain copies of the base data that are in domain order, rather than the default order in which rows were initially loaded. While projections can benefit performance for queries requiring ordering, they consume additional storage and complicate both database design and the re-assembly of rows during query execution. Actian's Vectorwise was spawned from the MonetDB research prototype. Like DB2 BLU, both Vectorwise and SAP HANA [10] store each column separately as a vector, with the value for the Nth row stored in the Nth entry in compressed form. HANA packs encoded values so densely that they may span registers, requiring complex shifting operations to isolate individual values [22]. In contrast, DB2 BLU pads bit-aligned values to register boundaries, so that it can operate on multiple values simultaneously in SIMD fashion without needing to do bit shifting [14]. And unlike HANA and Vertica, which temporarily stage INSERTs in row-organized tables,

DB2 BLU has no staging tables for INSERTed rows, simplifying concurrent query processing. Main-memory column stores such as Exasol are limited to what data can be cost-effectively stored in main memory, whereas DB2 BLU can store any amount of data cost-effectively on disk and cache the active portions in memory.

DB2 BLU is fully integrated into a full-function, mature DBMS (DB2) that permits storing data in either row- or column-organized tables in the same database, and referencing both in the same query, as Teradata's column store and EMC's Greenplum do. However, Teradata and Greenplum only store the columns separately, re-assembling them into rows for query processing [3]. In contrast, DB2 BLU keeps the columns separate and compressed during query processing, performing major operations such as predicate evaluation, joins, and grouping on the compressed values. SQL Server evaluates predicates directly on the compressed data for scan operators, but it's unclear whether it can perform other operations such as joins and grouping on compressed data, and no mention is made of SIMD operations [15].

12. CONCLUSIONS

Decades after the invention, prototyping, and initial products of relational DBMSs, the basics of relational DBMS technology are still in flux today. Databases are of course much, much larger than they were in the 1970s, and new data-intensive applications such as large-scale data analysis and program-driven trading are only further stressing DBMSs. But the main reason DBMSs have to keep changing is major changes in hardware. Our problems are so data intensive that they are chronically impacted by evolution in processors and memory hierarchies, and continually playing catch-up; that core ideas have lasted this long is itself a testament to the quality of the original designs.

DB2 BLU proves that one need not build a completely new database system from scratch to adapt to today's hardware or to achieve revolutionary improvements in performance and compression, that the extensibility built into DB2 LUW with the Starburst technology 20 years ago [11] can be leveraged to evolve and exploit the investment in mature products like DB2. This revolution through evolution concept allows for the non-disruptive and seamless introduction of radically new technology like BLU Acceleration into customer environments, while preserving full functionality.

Acknowledgments

---

[3] http://dbmsmusings.blogspot.com/2009/10/greenplum-announces-column-oriented.html

References

[1] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented DBMS. In *ICDE*, 2007.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, 2001.

[3] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10), 2012.

[4] 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. M. Lohman, K. Morfonios, R. Mueller, K. Murthy, I. Pandis, L. Qiao, V. Raman, R. Sidle, K. Stolze, and S. Szabo. Blink: Not your father's database! In *BIRTE*, 2011.

[5] 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. M. Lohman, K. Morfonios, R. Mueller, K. Murthy, I. Pandis, L. Qiao, V. Raman, R. Sidle, K. Stolze, and S. Szabo. Business analytics in (a) blink. *IEEE Data Eng. Bull.*, 2012.

[6] R. Bestgen and T. McKinley. Taming the business-intelligence monster. *IBM Systems Magazine*, 2007.

[7] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51, 2008.

[8] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *SIGMOD*, 1985.

[9] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood. Implementation techniques for main memory database systems. In *SIGMOD*, 1984.

[10] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1), 2012.

[11] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. *IEEE TKDE*, 2(1), 1990.

[12] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *SIGMOD*, 2007.

[13] IBM. DB2 with BLU acceleration. Available at http://www-01.ibm.com/software/data/db2/linux-unix-windows/db2-blu-acceleration/.

[14] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *PVLDB*, 1, 2008.

[15] P.-A. Larson, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, S. Rangarajan, R. Rusanu, and M. Saubhasik. Enhancements to SQL server column stores. In *SIGMOD*, 2013.

[16] Y. Li, I. Pandis, R. Mueller, V. Raman, and G. Lohman. NUMA-aware algorithms: the case of data shuffling. In *CIDR*, 2013.

[17] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17(1), 1992.

[18] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *ICDE*, 2008.

[19] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[20] K. Stolze, V. Raman, R. Sidle, and O. Draese. Bringing BLINK closer to the full power of SQL. In *BTW*, 2009.

[21] 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.

[22] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2, 2009.

[23] M. Zukowski and P. A. Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 35(1), 2012.