Deliverable 2.3
Integration of MonetDB Technology in Virtuoso

Abstract: This document describes the state and roadmap of the close collaboration of CWI and OGL in LOD2 to enhance RDF technology with advanced concepts from analytical relational database systems in Virtuoso. To this end, it first provides a primer in column storage technology. It identifies elements from the MonetDB system developed by CWI whose underlying ideas could be of use in RDF, which encompass columnar storage, column compression, vectorized execution, recycling intermediates, and run-time query optimization. The current state of this integration of ideas is summarized in a description of the new compressed column store features of Virtuoso7, including vectorized execution. We conclude with a roadmap of possible further enhancements to Virtuoso7 during the rest of the LOD2 project.

This report defines the rules that will be enforced by LOD2 auditors before BSBM results can get certified and published on the official website.

The information in this document reflects only the author’s views and the European Community is not liable for any use that may be made of the information contained therein. The information in this document is provided “as is” without guarantee or warranty of any kind, express or implied, including but not limited to the fitness of the information for a particular purpose. The user thereof uses the information at his/her sole risk and liability.
Project co-funded by the European Commission within the Seventh Framework Programme (2007 – 2013)

History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Reason</th>
<th>Revised by</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>06/08/2011</td>
<td>OpenLink Input</td>
<td>Orri Erling</td>
</tr>
<tr>
<td>1.0</td>
<td>24/08/2011</td>
<td>First Draft</td>
<td>Peter Boncz</td>
</tr>
</tbody>
</table>

Author List

<table>
<thead>
<tr>
<th>Organisation</th>
<th>Name</th>
<th>Contact Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>CWI</td>
<td>Peter Boncz</td>
<td><a href="mailto:boncz@cwi.nl">boncz@cwi.nl</a></td>
</tr>
<tr>
<td>OLG</td>
<td>Orri Erling</td>
<td><a href="mailto:erling@xs4all.nl">erling@xs4all.nl</a></td>
</tr>
</tbody>
</table>
Introduction

One of the prime goals of the LOD2 project is boosting the performance and scalability of RDF storage solutions, such that the performance gap that now exists between choosing a solution based on relational technology and one based on RDF technology (the “RDF Tax”) is reduced. The below graph (taken from\(^1\)) illustrates this problem where Triple stores reached 1-10K QMpH, where SQL systems did 40-175K; a difference of a factor 15-40. One can of course question the value of this observation, by e.g. remarking that

- SQL systems have been around for decades and have been heavily optimized in the meantime
- the schema-last approach RDF offers benefits in terms of flexibility
- the particular benchmark shown (BSBM), is highly structured and plays into the strength of SQL.
- performance is just one of many considerations when designing IT architectures and selecting technologies

Nevertheless, the question remains why in such situations the RDF Tax necessarily needs to be this high. Reducing the RDF Tax is thus one of the core objectives of LOD2 and the concrete reason the CWI was invited to participate in this project.

The database research group of the CWI is well-known for its work on database architecture, especially in the area of analytical database technology. Its MonetDB open source system (monetdb.cwi.nl) is the pioneer in bringing column-store technologies back into popularity but also in revising relational query execution architectures to make them work more efficiently on modern hardware. This led to work on tuning the access patterns of database queries to be CPU cache conscious, and also to make query processing amendable to deeply pipelined CPUs with SIMD instructions by introducing concepts like vector processing. This document starts therefore with a primer on advanced column store techniques.

In LOD2, Openlink and CWI work together to bring some of the unique technologies and architectural principles used in MonetDB into the Virtuoso product of LOD2 partner Openlink (OGL), with the goal of providing to the RDF user community a faster and more scalable RDF processing technology. The insights gained in the process will also be useful for other RDF store providers to enhance their respective technologies as well.

During the entire LOD2 project CWI and OGL work together in close collaboration. This document is hence a progress report of what has been done so far, and a roadmap of what is in store. It starts by identifying some of the architectural ideas of MonetDB that may have potential in the RDF space and describing them in detail. It subsequently discusses the progress made in turning Virtuoso into a compressed column store with vectored execution. Finally, it discusses a roadmap of next steps.

---

## Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOD</td>
<td>Linked Open Data</td>
</tr>
<tr>
<td>BSBM</td>
<td>Berlin SPARQL Benchmark</td>
</tr>
<tr>
<td>QMpH</td>
<td>Query Mixes per Hour (BSBM performance metric)</td>
</tr>
<tr>
<td>TPC-H</td>
<td>Relational Decision Support Benchmark (<a href="http://www.tpc.org">www.tpc.org</a>)</td>
</tr>
<tr>
<td>RDF-H</td>
<td>RDF Equivalent of TPC-H</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data (CPU instruction family)</td>
</tr>
<tr>
<td>NSM</td>
<td>Normalized Storage Model (row-storage)</td>
</tr>
<tr>
<td>DSM</td>
<td>Decomposed Storage Model (column-storage)</td>
</tr>
<tr>
<td>PAX</td>
<td>Partitioning Across (a mix between NSM and DSM)</td>
</tr>
<tr>
<td>DBA</td>
<td>Database Administrator</td>
</tr>
</tbody>
</table>
Table of Contents

1. A Column Store Primer ............................................................................................................ 6
   1.1 Background .......................................................................................................................... 7
      1.1.1 Architecture and Application Trends .............................................................................. 8
      1.1.2 Fundamental Performance Trade-Offs ........................................................................... 10
   1.2 Column-Store Architectures ............................................................................................... 11
      1.2.1 C-Store .......................................................................................................................... 12
      1.2.2 MonetDB ....................................................................................................................... 12
   1.3 Column-Store Internals and Advanced techniques .............................................................. 14
      1.3.1 Vectorized processing ................................................................................................. 14
      1.3.2 Compression ................................................................................................................ 15
      1.3.3 Operating directly on compressed data ....................................................................... 17
      1.3.4 Late Materialization .................................................................................................... 18
      1.3.5 Joins ............................................................................................................................ 20
      1.3.6 Inserts/updates/deletes ............................................................................................... 23

2. Identifying RDF Potentials in MonetDB ............................................................................... 25
   2.1 Columnar Compression ....................................................................................................... 25
      2.1.1 Potential for RDF stores .............................................................................................. 25
   2.2 Vectorized Execution ......................................................................................................... 25
      2.2.1 Potential for RDF stores .............................................................................................. 25
   2.3 Run-Time Sample-driven Query Optimization .................................................................... 26
      2.3.1 ROX: Runtime Optimization of XQueries .................................................................... 26
      2.3.2 Chain Sampling in ROX ............................................................................................. 27
      2.3.3 Pipelined Execution .................................................................................................... 30
      2.3.4 Potential for RDF stores ............................................................................................ 30
   2.4 Recycling Intermediate Results .......................................................................................... 32
      2.4.1 MonetDB Implementation ............................................................................................ 32
      2.4.2 Recycling for pipelined RDBMS .................................................................................. 33
      2.4.3 Ongoing Research: ...................................................................................................... 35
      2.4.4 Potential for RDF stores ............................................................................................ 35

3. Enhancing Virtuoso .................................................................................................................. 36
   3.1 Virtuoso Column Store and Vectored Execution ................................................................. 36
      3.1.1 Comparison With Previous Column Store Work .......................................................... 36
      3.1.2 Bulk Load Concurrency .............................................................................................. 38
      3.1.3 Vectored Execution ..................................................................................................... 38
      3.1.4 Vectored Expressions and Procedures ........................................................................ 39
      3.1.5 Vectored Index Lookup .............................................................................................. 39
      3.1.6 Dynamic Vector Size .................................................................................................. 39
      3.1.7 Vectored Insert ........................................................................................................... 39
   3.2 Performance results ............................................................................................................. 41
      3.2.1 BSBM ............................................................................................................................ 41
      3.2.2 Join Micro-Benchmark ............................................................................................... 42
   3.3 RDF-H .................................................................................................................................. 43
      3.3.1 Example Query ............................................................................................................ 43
      3.3.2 Error! Bookmark not defined. ..................................................................................... 45
   3.4 Innovation Ideas .................................................................................................................. 45
      3.4.1 Query Optimization and Dealing with Plan Space Explosion ....................................... 45
1. A Column Store Primer

Database system performance is directly related to the efficiency of the database system at storing data on primary storage (e.g., disk) and moving it into CPU registers for processing. For this reason, there is a long history in the database community of research exploring physical storage alternatives, including sophisticated indexing, materialized views, and vertical and horizontal partitioning.

In recent years, there has been renewed interest in so-called “column-oriented” systems (sometimes called “column-stores”), such as MonetDB, MonetDB/X100, C-Store\(^2\) and SybaseIQ, which completely vertically partition a database into a collection of individual columns that are stored separately. By storing each column separately on disk, these column-based systems enable queries to read just the attributes they need, rather than having to read entire rows from disk and discard unneeded attributes once they are in memory.

![Figure 1: Physical layout of column-oriented vs. row-oriented databases](image)

Figure 1 illustrates the basic differences in the physical layout column-stores as compared to traditional, row-oriented databases (“row-stores”): here, a sales table, with several attributes, is stored on disk. In the two column-oriented approaches (Figure 1(a) and Figure 1(b)), there is one file per column. In this case, a query that computes, for example, the number of sales of a particular product in July would only need to access the provided and date columns, and only the files correspond to these columns would need to be read from disk (we will explain the differences between Figure 1(a) and Figure 1(b) in a moment.) In the row-oriented approach (Figure 1(c)), there is just a single file containing all of the data. Although in principle the disk could access just the provided and data columns in each record, in practice doing so would be extremely inefficient as it would require two seeks to access each record. Instead, the disk and operating system will read a block of data, which likely contains several records, and filter out the needed fields once the data is in memory. For this query, this will result in significantly more data being read in the traditional approach. Since disk I/O and memory bandwidth are often a performance bottleneck in databases, a column-store is likely to be much more efficient at executing this query. In general,

\(^2\) MonetDB/X100 and C-Store evolved into the commercial systems VectorWise and Vertica, respectively.
the benefit of a column-store is most pronounced when queries access many records from a small number of columns. If queries only access a few records, then the column-store will have to seek several times (to different columns) to read those records. If queries access many records, then large swaths of entire columns can be read, amortizing the seeks to the different columns. In a conventional row store, in contrast, if queries only access a few records, then the additional transfer time required to read all of the columns from those records will be dominated by the seek time to the beginning of the record, and there will be fewer seeks than in a column-oriented system. However, as more and more records are accessed, the transfer time begins to dominate, and a column-oriented approach begins to perform better than a row-oriented approach. For this reason, column-stores are typically used in analytic applications, with queries that scan a large fraction of individual tables and compute aggregates or other statistics over them.

1.1 Background

Although recent systems employ concepts that are at a high level similar to those in previous research proposals for vertical partitioning [Batory 1979; Copeland and Khoshafian 1985; Karasalo and Svensson 1983; Lorie and Symonds 1971], they include many architectural features beyond those in early work on vertical partitioning that are designed to maximize the performance on analytic workloads. The goal of this primer is to survey these recent research results, architectural trends, and optimizations. Specifically ideas we focus on include: — Block-oriented and vectorized processing [Boncz et al. 2005; Abadi et al. 2006] By passing cache-line sized blocks of tuples between operators, and operating on multiple values at a time, rather than using a conventional, tuple-at-a-time iterator, column-stores can achieve substantially better cache utilization and CPU efficiency. The use of vectorized CPU instructions for selections, expressions, and other types of arithmetic on these blocks of values can further improve throughput.

- **Column-specific compression** [Zukowski et al. 2006; Abadi et al. 2006] by compressing each column using a compression method that is most effective for it, substantial reductions in the total size of data on disk can be achieved. By storing data from the same attribute (column) together, column-stores can obtain good compression ratios using simple compression schemes.

- **Late materialization** and direct operation on compressed data [Abadi et al. 2007] Late materialization refers to deferring the joining of columns into wider tuples (which must be done before data is presented to users.) Late materialization allows columns to be kept in a compressed representation in memory, whereas creating wider tuples generally requires decompressing them first. It also makes it possible to apply filters to individual columns, and then extract only the values from output columns that satisfy the filters, rather than building wide tuples that contain all columns involved in a query. Both of these optimizations can dramatically increase memory bandwidth efficiency.

- **Redundant representation** of individual columns in different sort orders [Stonebraker et al. 2005] Columns that are sorted according to a particular attribute can be filtered much more quickly on that attribute. By storing several copies of each column sorted by attributes heavily used in an applications query workload, substantial performance gains can be had (C-Store calls groups of columns sorted on a particular attribute ―projections‖.) Additionally, low-cardinality data that is stored in sorted order can be aggressively compressed.

- **Virtual IDs** The simplest way to represent a column in a column-store involves associating a tuple identifier (e.g., a numeric primary key) with every column. Explicitly representing this key substantially bloats the size of data on disk, and reduces I/O efficiency. Instead, modern column-stores avoid storing this ID column by using the position (offset) of the tuple in the column as a virtual identifier (see Figure 1(a) vs. Figure 1(b)).

- **Efficient join** implementations [Manegold et al. 2004; Abadi et al. 2006]. Because columns are stored separately, join strategies similar to classic semijoins [Bernstein and Chiu 1981] are possible. For specific types of joins, these can be much more efficient than traditional hash or merge joins used in OLAP settings.
Efficient loading architectures [Heman et al. 2010; Stonebraker et al. 2005]. Finally, one concern with column-stores is that they may be slower to load and update than row-stores, because each column must be written separately, and because data is kept compressed. Since load performance can be a significant concern in data warehouse systems, optimized loaders are important. For example, in the C-Store system, data is first written into an uncompressed, write-optimized buffer (the “WOS”), and then flushed periodically in large, compressed batches. This avoids doing one disk seek per-attribute, per-row and having to insert new data into a compressed column; instead writing and compressing many records at a time.

To illustrate the benefit that column-orientation and these optimizations have, we briefly summarize a result from [Abadi et al. 2008]. This paper compared the performance of the academic C-Store prototype to a commercial row-oriented ("row-store") system. It studied the effect of various column-oriented optimizations on overall query performance on SSBM [O'Neil et al.] (A simplified version of the TPC-H data warehousing benchmark.) The results of the average runtime of all queries in the benchmark on a scale 10 database (60 million tuples) are shown in Figure 2. The bar on the left shows the performance of an academic prototype of C-Store, as various optimizations are removed (so the “baseline” system with all optimizations takes about 4 seconds, and the completely unoptimized system takes about 40 seconds.) The bar on the right shows the performance of a commercial row-store system. From these results it is apparent that the optimized column-store is about a factor of 5 faster than the commercial row-store, but that the unoptimized system is somewhat slower than the commercial database. One reason that the unoptimized column-store does not do particularly well is that the SSBM uses relatively narrow columns, so the baseline I/O reduction from column-orientation is reduced. In most real data-warehouses, the ratio of columns-read to table-width would be much smaller, so these advantages would be more pronounced.

![Figure 2: Performance of C-Store versus a commercial database system on the Star Schema Benchmark, with different column-oriented optimizations enabled.](image-url)

Though comparing absolute performance numbers between a full-fledged commercial system and an academic prototype is tricky, these numbers show that unoptimized column-stores with queries that select a large fraction of columns provide comparable performance to row-oriented systems, but that the optimizations proposed in modern systems can provide order-of-magnitude reductions in query times.

1.1.1 Architecture and Application Trends

At its core, the basic design of a relational database management system has remained to date very close to systems developed in the 1980s [DeWitt 2009]. The hardware landscape, however, has changed dramatically. In 1980, a Digital VAX 11/780 had a 1 MIPS CPU with 1KB of cache memory, 8 MB maximum main memory, disk drives with 1.2 MB/second transfer rate and 80MB capacity, and carried a $250K price tag. In 2010, servers typically had 5,000 to 10,000 times faster CPUs, larger cache and RAM sizes, and larger disk capacities. Disk transfer times for hard drives improved about 100 times and average disk-head seek times are 10 times faster (30msec...
Integration Of MonetDB Technology in Virtuoso

The differences in these trends (10,000x vs. 100x vs. 10x) have had a significant impact on the performance of database workloads [DeWitt 2009]. The imbalance between disk capacity growth and the performance improvement of disk transfer and disk seek times can be viewed through two metrics: (a) the transfer bandwidth per available byte (assuming the entire disk is used), which has been reduced over the years by two orders of magnitude, and (b) the ratio of sequential access speed over random access speed, which has increased one order of magnitude.

These two metrics clearly show that DBMSs need to not only avoid random disk I/Os whenever possible, but, most importantly, preserve disk bandwidth. As random access throughout the memory hierarchy became increasingly expensive, query processing techniques started to increasingly rely on sequential access patterns. DBMS architectures continued evolving around the premise that sequential access patterns are always desirable. However, as database sizes increased, scanning through the data became slower and slower. A bandwidth-saving solution was clearly needed, yet most database vendors did not find the column stores, also known as the Decomposed Storage Model (DSM) being a viable replacement to row-storage, i.e. the Normalized Storage Model (NSM). Since most queries relied on sequential scans, in order for a column-based storage scheme to outperform a row-based one it needed to have a fast mechanism for reconstructing tuples (since the rest of the DBMS would still operate on rows) and it also needed to be able to amortize the cost of disk seeks when accessing multiple columns on disk. Faster CPUs would eventually enable the former and larger memories (for buffering purposes) would allow the latter.

Although modern column-stores gained popularity for being efficient on processing disk-based data, it was main-memory column-based techniques that gained first attention. By the late 1990s there was intense interest in investigating in-memory data layouts for addressing the growing speed disparity between CPU and main memory. In 1980, the time required to access main memory and execute an instruction were balanced, however, memory latency quickly grew to hundreds of CPU cycles. Two directions emerged: “Pure” column-stores and “Hybrid” systems.

The MonetDB project [Boncz 2002a] was the main representative column-store project in the academic community. The original motivation behind MonetDB, which was initially developed as a main-memory only system, was to address the memory bandwidth problem and also improve computational efficiency by avoiding an expression interpreter [Boncz and Kersten 1999]. New query execution algebra was developed on a storage format that resembled DSM with virtual IDs. Subsequent research studied cache-conscious query processing algorithms (a comprehensive presentation of MonetDB follows in Section 3.2). In the commercial space, one of the first column-store systems was SybaseIQ, which was met with limited adoption.

![NSM vs. Cache vs. PAX Page Storage Models](image)

**Figure 3: Storage models for storing database records inside disk pages:**

- **NSM** (row-store) and **DSM** (a predecessor to column-stores).
- **PAX** (for Partition Attributes Across) was the first hybrid NSM/DSM approach, where each NSM page is organized as a set of mini columns [Ailamaki et al. 2001]. It retained the NSM disk I/O pattern, but optimized cache-to-RAM communication. Data morphing [Hankins and Patel 2003]
presented a dynamic version of PAX, whereas Clotho [Shao et al. 2004] studied custom page layouts using scatter-gather I/O. Finally, Fractured Mirrors [Ramamurthy et al. 2002] leveraged mirroring for reliability and availability to store one copy in NSM format and one in DSM, thereby achieving the performance benefits of each format.

In the mid-2000s, incredibly inexpensive drives and CPUs had made it possible to collect, store, and analyze vast quantities of data. New, internet-scale user-facing applications led to the collection of unprecedented volumes of data for analysis and the creation of multi-terabyte and even petabyte-scale data warehouses. To cope with these challenging performance requirements, architects of new database systems revisited the benefits of column-oriented storage, this time combining several techniques around column-based data, such as read-only optimizations, fast multi-column access, disk/CPU efficiency, and light-weight compression. The (re)birth of column-stores was marked by the introduction of two modern column-store systems. C-Store, which presented the first comprehensive design description of a column-store, and MonetDB/X100 which focused on optimizations targeted at both modern processors and modern storage media. Through the end of the 2000s there was an explosion of new column-oriented DBMS products that were influenced by these systems, which was later followed by the adoption of several column-based techniques by traditional vendors of row-store systems (such as Oracle, Microsoft, and Teradata).

1.1.2 Fundamental Performance Trade-Offs

While DSM made it possible to quickly scan a single column of a table, scanning multiple columns or, even worse, scanning an entire table stored in columns was significantly slower than NSM. The reason was the various overheads in reconstructing a tuple from multiple columns, accessing multiple locations on disk, and processing extraneous, per-column information. If column-stores were to ever be competitive to row stores they needed to provide good performance across a range of workloads, and that included queries that needed to access large fractions of a record (large projectivity) or entire records. As CPU speed increases kept outpacing disk bandwidth increases, and software optimization efforts focused on read-optimized data representations, accessing multiple columns during column-store scans became more and more competitive to row-store scans. This was demonstrated by several studies over the past ten years.

- In Fractured Mirrors [Ramamurthy et al. 2002], the authors proposed several optimizations for DSM. Each DSM column was stored inside a B-tree, where the leaf nodes contained all the column attribute values. They eliminated the IDs per column, they amortized the header overhead across multiple column values (Graefe also describes an efficient method for storing columns inside a B-tree [Graefe 2007]), and used chunk-based tuple reconstruction.
- In a from-scratch comparison of column and row scanners [Harizopoulos et al. 2006], the authors implemented a stand-alone storage manager and used both read optimized storage representations and large prefetching units to hide disk seeks across columns. This time, with technology available at 2006 and in worse-case scenarios where every column in a table must be accessed, column-store scans were 20-30% slower than row-store scans Finally, in [Tsirigiani et al. 2009] the authors considered Flash solid state drives as the primary storage media for a database system, and demonstrated the effectiveness of column-based storage models for on-disk storage. Since Flash SSDs offer significantly faster random reads than HDDs, column schemes have a comparable I/O cost to a row-store scan when reading full-row tuples (since there are no expensive disk-head seeks).

Figure 4 consolidates some of the results of the above-mentioned studies into a single graph. Column-store (or DSM) scan times are normalized against row-store scans (which always have a constant I/O cost) for different projectivity (percentage of a tuple that is read). The baseline DSM performance is copied from a 2001 paper [Ailamaki et al. 2001] which also used it as baseline to compare I/O performance against PAX and NSM. Over time, worse-case scenarios for column-stores (projectivity close to 100%) came increasingly closer to row-store performance.
Interestingly, when it comes to solid state storage (such as Flash SSDs), column-oriented storage was shown to never be worse than row storage, and in some cases where selective predicates were used, it outperformed row storage for any projectivity (the details can be found in [Tsirigiani et al. 2009] and are beyond the scope of this primer).

Figure 4: Performance of column-oriented scans for various degrees of projectivity (percentage of a full-row tuple that needs to be reconstructed) compared to (normalized) row-store scan, from various studies over the years

1.2 Column-Store Architectures

In this section, we describe the high level architecture of three modern column oriented databases: C-Store, MonetDB and MonetDB/X100.

1.2.1 C-Store

In C-Store, the primary representation of data on disk is a set of column files. Each column-file contains data from one column, compressed using a column-specific compression method, and sorted according to some attribute in the table that the column belongs to. This collection of files is known as the "read optimized store" (ROS). Additionally, newly loaded data is stored in a write-optimized store ("WOS"), where data is uncompressed and not vertically partitioned. As discussed in the Introduction, the WOS enables efficient loading of data, and amortizes the cost of compression and seeking. Periodically, data is moved from the WOS into the ROS via a background "tuple mover" process, which sorts, compresses, and writes re-organized data to disk.

Each column in C-Store may be stored several times in several different sort orders. Groups of columns sorted on the same attribute are referred to as "projections". Typically there is at least one projection containing all columns that can be used to answer any query. Projections with fewer columns and different sort orders are used to optimize the performance of specific frequent queries; for example, a query that accesses the number of sales in a specific region per month over a certain time frame could benefit from a projection containing the product id, sales date, and region attributes sorted by product region and then date. Sorting allows efficient subsetting of just the relevant records, and also makes it possible to aggregate the results one month at a time without maintaining any intermediate aggregation state. In contrast, another query which just needs to count the sales by month, regardless of region, might benefit from a projection that just stores data sorted by date. Figure 5 illustrates these two alternative projections for the sales table (in C-Store, we use the notation (saleid,date,region|date) to indicate a projection of the sales table containing saleid, date and region attributes sorted by date). Note that these projections contain different columns, and neither contains all of the columns in the table.
Figure 5: Two different projections of the Sales table

Each column in C-Store is also compressed. Different compression methods are used depending on whether the column is sorted, and on the data type and number of distinct values in a column. For example, the sorted product class column is likely to have just a few distinct values; since these are represented in order, this column can be encoded very compactly using run-length encoding (RLE). In RLE, a consecutive series of products of the same class is represented as a (count, product class) pair, rather than count distinct tuples. Details of C-Store compression methods are given in Section 1.4.2 below.

Additionally, C-Store uses a “no-overwrite” storage representation, where updates are treated as deletes followed by inserts, and deletes are processed by storing a special “delete column” that records the time every tuple was deleted (if ever).

Query execution in C-Store involves accessing data from both the ROS and WOS and unioning the results together. Queries are run as of a specific time, which is used to filter out deleted tuples from the delete column. This allows queries to be run as of some time in the past. Queries that modify the database are run using traditional two-phase locking. If read-only queries are tolerant of reading slightly stale data they can be run without setting locks by executing them as of some time in the very recent past. Finally, C-Store’s query executor utilizes a number of advanced execution techniques, including late materialization, various column-oriented join techniques, and batch processing. These optimizations are described in more detail in Section 1.4.

1.2.2 MonetDB

MonetDB is designed from scratch focusing on handling analytical workloads efficiently on modern hardware. MonetDB differs from traditional RDBMS architecture in many aspects, such as its:

- Execution engine, which uses a column-at-a-time-algebra [Boncz and Kersten 1999],
- Processing algorithms, that minimize CPU cache misses rather than IOs [Boncz et al. 1999],
- Indexing, which is not a DBA task but happens as a by-product of query execution [Idreos et al. 2009],
- Query optimization, which is done at run-time, during query incremental execution [Abdel Kader et al. 2009] and
- Transaction management, which is implemented using explicit additional tables and algebraic operations, so read-only workloads can omit these and avoid all transaction overhead [Boncz and Kersten 1999].

Traditional query execution uses a tuple-at-a-time, pull-based, iterator approach in which each operator gets the next input tuple by calling the next() method of the operators of its children in the operator tree. In contrast, MonetDB works by performing simple operations column-at-a-time. In this, MonetDB aimed at mimicking the success of scientific computation programs in extracting efficiency from modern CPUs, by expressing its calculations typically in tight loops over arrays, i.e.,
columns. Such code is well-supported by compiler technology to extract maximum performance from CPUs through techniques as strength reduction (replacing an operation with an equivalent less costly operation), array blocking (grouping subsets of an array to increase cache locality), and loop pipelining (mapping loops into optimized pipeline executions). The MonetDB column-at-a-time primitives not only get much more work done in less instructions compared to tuple-at-a-time database systems - primarily thanks to eliminating tuple-at-a-time iterator function calls - but its instructions also run more efficiently in modern CPUs. That is, MonetDB query plans provide the CPU more in-flight instructions, keep the pipelines full and the branch misprediction and CPU cache miss rates low, and also automatically (through the compiler) allow the database system to profit from SIMD instructions.

The column-at-a-time processing is realized through the so-called BAT Algebra, which offers operations that work only on a handful of BATs, and produce new BATs. BAT stands for Binary Association Table, and refers to a two-column <surrogate,value> table as proposed in DSM. The surrogate is just a Virtual ID; it effectively is the array index of the column and is not materialized. Data in execution is always stored in BATs, and even the final result of a query is a collection of BATs. MonetDB hence takes late tuple materialization to the extreme. The absence of tuple reconstruction fits another goal of MonetDB, namely using a single internal data representation (BATs) to manipulate data of widely different data models. MonetDB follows a front-end/back-end architecture, where the front-end is responsible for maintaining the illusion of data stored in some logical data model. Front-ends have been created that allow storage and querying of purely relational data, but also object-oriented, XML RDF and graph data in MonetDB. The front-ends translate end-user queries in (SQL, OQL, XQuery, SPARQL) into BAT Algebra, execute the plan, and use the resulting BATs to present results.

![Figure 6: MonetDB: BAT Algebra Execution](image)

Figure 6 shows query plans in BAT Algebra being generated by various front-ends to be executed in the MonetDB back-end. BATs are essentially in-memory (or memory mapped) arrays. The BAT algebra operators, such as `select(byear,1927)` fully consume BAT inputs and fully produce a BAT output. The reason behind the efficiency of the BAT Algebra is its hard-coded semantics, causing all operators to be predicate-less. For comparison, in the relational algebra, the Join and Select operators take an arbitrary Boolean column expression that determines which tuples must be joined and selected. The fact that this Boolean expression is arbitrary, and specified at query time only, means that the RDBMS must include some expression interpreter in the critical runtime code-path of the Join and Select operators. Such predicates do not occur in BAT Algebra; therefore we also say it has “zero degrees of freedom”. This absence of freedom means the interpreter is removed from inside the operator; all BAT algebra operators perform a fixed hard-coded action on a simple array. As such, complex expressions in a query must be mapped into multiple subsequent BAT Algebra operators. Expression interpretation in MonetDB effectively
occurs on the granularity of whole column-at-a-time BAT Algebra operators, which much better amortizes interpretation overhead. The philosophy behind the BAT Algebra can also be paraphrased as “the RISC approach to database query languages”: by making the algebra simple, the opportunities are created for implementations that execute the common case very fast.

MonetDB stores columns uncompressed on disk, and uses memory mapping to provide the BAT Algebra operations direct access to it, unhindered by any API. The absence of a buffer manager means that MonetDB must rely on providing virtual memory access advice to the OS, which means the system does not have absolute control over I/O scheduling. An additional drawback of the column-at-a-time execution model is its full materialization of intermediate results. Together, these aspects make MonetDB vulnerable to swapping when it’s working set starts exceeding RAM. These problems were addressed by a new system developed in the same research group, (confusingly) called MonetDB/X100 [Zukowski 2009], though it was a from-scratch development. MonetDB/X100 does perform explicit I/O, even in an advanced way, adaptively finding synergy in the I/O needs of concurrent queries through its Active Buffer Manager (ABM) and Cooperative Scans [Zukowski et al. 2007]. The vectorized execution model of MonetDB/X100 that strikes a balance between full materialization and tuple-at-a-time iterators, as well as its novel handling of updates (Positional Delta Trees [Heman et al. 2010]), and new high-speed compression algorithms [Zukowski et al. 2006] will be discussed later in this primer.

1.3 Column-Store Internals and Advanced techniques

1.3.1 Vectorized Processing

Database text-books generally contrast two strategies for the query execution layer, namely the “Volcano-style” iterator model [Graefe 1993], which we also refer to as tuple-at-a-time pipelining, versus full materialization. In tuple-at-a-time pipelining, the next () method of a relational operator in a query tree produces one new tuple at-a-time, obtaining input data by calling the next () method on its child operators in the tree. Data is thus pulled upwards through the query tree. Apart from being elegant in the software engineering sense, this approach has the advantage that materialization of intermediate results is minimized.

In full materialization, on the other hand, the query operators work in isolation, consuming an input from storage (disk, or RAM) and writing its output to storage. MonetDB is one of the few database systems using full materialization, product of its BAT Algebra designed to make operators and their interactions simpler and thus more CPU efficient. However, MonetDB therefore may cause excessive resource utilization in queries that generate large intermediate results.

We now turn our attention to a third alternative called “vectorized execution” pioneered in the MonetDB/X100 (now VectorWise) system, which strikes a balance between full materialization and tuple pipelining. This model separates query progress control logic from data processing logic. Regarding the control flow, its relational operators are mostly similar to those in tuple pipelining, with the sole distinction that the next () method of the iterator model is not supposed to return a single next tuple, but a vector of N tuples. Regarding data processing, the so-called primitive functions that operators use to do actual work (e.g., adding or comparing data values) look much like MonetDB’s BAT Algebra, processing data vector-at-a-time. Thus, vectorized execution in VectorWise combines pipelining (avoidance of materialization) that make traditional database systems scale, with the array-loops code patterns that make MonetDB fast.

Vectorized processing has a number of advantages:

- **Reduced interpretation overhead**: the amount of function calls performed by the query interpreter goes down by a factor equal to the vector size (N=1000 is typical). On computationally intensive queries like TPC-H Q1 this can improve performance by two orders of magnitude.
Integration Of MonetDB Technology in Virtuoso

- **Better cache locality:** MonetDB/X100 tunes the vector size such that all vectors needed for evaluating a query together comfortably fit in the CPU cache. If the vector size is chosen too large (like in MonetDB, where vector size is table size), the vectors do not fit and additional memory traffic slows down the query. Regarding instruction cache, the vectorized model also improves the situation as control now stays for 1000 iterations in the same primitive function, thus creating instruction locality.

- **Compiler optimization opportunities:** as mentioned in the description of MonetDB, vectorized primitives which typically perform a tight loop over arrays, are amendable to some of the most productive compiler optimizations, and typically also trigger compilers to generate SIMD instructions.

- **Block algorithms:** the fact that data processing algorithms now process N tuples, often gives rise to logical algorithm optimizations. For instance, when checking for some condition (e.g. output buffer full), a tuple-at-a-time execution model performs the check for every tuple, while a vectorized algorithm can first check if the output buffer has space for N more results, and if so, do all the work on the vector without any checking. Additionally, algorithms that perform memory accesses in a tight vectorized loop on modern CPUs are able to generate multiple outstanding cache misses, for different values in a vector. This is because when a cache miss occurs, modern CPUs can speculate ahead in such tight loops. This is not possible in the tuple-at-a-time architecture, since the late-binding API calls which the CPU encounters between processing different tuples inhibit this. Generating multiple concurrent cache misses is necessary to get good memory bandwidth on modern computers.

Vectorized execution mostly concerns the query operators and their handling of in-flight tuple data flowing through a query execution tree. There can be a distinction made between the data layout used for persistent storage by the storage manager, and the data layout used by the query executor. While vectorized execution was developed in the column storage context of MonetDB and MonetDB/X100, the principle can also be applied to row stores [Padmanabhan et al. 2001] as it is not tied to the storage manager. In [Zukowski et al. 2008] it was shown that a vectorized query execution system can support both vertical (column) and horizontal (record) tuple representations in a single execution framework easily, and even can store in-flight tuples in mixed mode (some columns together as a record, and others vertically). This study also showed that performance of operators can be significantly influenced by the storage format, where operator characteristics, hardware parameters and data distributions determine what works best. Typically, sequential operators (project, selection) work best on vertical vectors (exploiting automatic memory prefetching and SIMD opportunities), whereas random access operator (hash-join or -aggregation) work best using blocks of horizontal records, due to cache locality. Since conversion between horizontal and vertical formats is cheap using vectorized execution, this creates the possibility that a query plan would change the tuple-layout as part of the query plan, possibly multiple times. This open a new ground for query optimizers of query layout planning that should determine the best layout for each stage of the query execution plan using cost-based estimation.

### 1.3.2 Compression

Intuitively, data stored in columns is more compressible than data stored in rows. Compression algorithms perform better on data with low information entropy (high data value locality). Imagine a database table containing information about customers (name, phone number, e-mail address, snail-mail address, etc.). Storing data in columns allows all of the names to be stored together, all of the phone numbers together, etc. Certainly phone numbers will be more similar to each other than surrounding text fields like e-mail addresses or names. Furthermore, if the data is sorted by one of the columns, that column will be super-compressible (for example, runs of the same value can be run-length encoded).

Usually, however, the bottom line goal is performance, not compression ratio. Disk space is cheap, and is getting cheaper rapidly. However, compression improves performance (in addition to
reducing disk space) since if data is compressed, then less time must be spent in I/O as data is read from disk into memory (or from memory to CPU).

Given that performance is what we are trying to optimize, this means that some of the “heavier-weight” compression schemes that optimize for compression ratio (such as Lempel-Ziv, Huffman, or arithmetic encoding), are less suitable than “lighter-weight” schemes that sacrifice compression ratio for decompression performance. Light-weight compression schemes that compress a column into mostly fixed-width (smaller) values (with exceptions handled carefully) are often preferred, since this allows a compressed column to be treated as an array. Iterating through such an array (e.g. for decompression) can leverage the SIMD instruction set on modern CPUs for vectorized parallelism (as described above), significantly improving performance.

There have been several research studies that evaluate the performance of different compression algorithms for use with a column-store [Abadi et al. 2006; Zukowski et al. 2006; Holloway et al. 2007; Hodak 2009; Binnig et al. 2009]. Some of these algorithms are sufficiently generic that they can be used in both row-stores and column-stores; however some are specific to column-stores since they allow compression symbols to span across multiple consecutive values within the same column (this would be problematic in a row-store, since, in a row-store, consecutive values from the same column are not stored consecutively on storage). These schemes include:

- **Run-length Encoding.** Run-length encoding (RLE) compresses runs of the same value in a column to a compact singular representation. Thus, it is well-suited for columns that are sorted or that have reasonable-sized runs of the same value. These runs are replaced with triples: (value, start position, runLength) where each element of the triple is typically given a fixed number of bits. For example, if the first 42 elements of a column contain the value ‘M’, then these 42 elements can be replaced with the triple: (‘M’, 1, 42).
  When used in row-oriented systems, RLE is only used for large string attributes that have many blanks or repeated characters. But RLE can be much more widely used in column-oriented systems where attributes are stored consecutively and runs of the same value are common (especially in columns that have few distinct values).
  As described above, the C-Store architecture results in a high percentage of columns being sorted (or secondarily sorted) and thus provides many opportunities for RLE-type encoding.

- **Bit-Vector Encoding.** Bit-vector encoding is most useful when columns have a limited number of possible data values (such as states in the US, or flag columns). However, it can be used even for columns with a large number of values if the bit-vectors are further compressed. In this type of encoding, a bit-string (whose number of bit is equal to the size of the column) is associated with each possible unique element from a column’s domain, with a ‘1’ in the \(i\)-th position in the bit string if the \(i\)-th value in the column is equal to domain element that the bit string is associated with, and a ‘0’ otherwise.
  For example, the following column data:
  
  | 1 | 1 | 3 | 2 | 2 | 3 | 1 |

  Would be represented as three bit-strings:
  
  - Bit-string for value 1: 1100001
  - Bit-string for value 2: 0001100
  - Bit-string for value 3: 0010010

  Since an extended version of this scheme can be used to index row-stores (so-called bit-map indices [O’Neil and Quass 1997]), there has been much work on further compressing these bit-maps and on the implications of this further compression on query performance [Moffat and Zobel 1994; Antoshenkov 1994; Amer-Yahia and Johnson 2000; Johnson 1999; Wu et al. 2001; Wu et al. 2001; Wu et al. 2002].

- **Dictionary.** Dictionary encoding works well for distributions with a few very frequent values, and can also be applied to strings. The simplest form constructs a dictionary table for an entire table column sorted on frequency, and represents values as the integer position in this table.
  These integers can again be compressed using an integer compression scheme. The global
dictionary may grow large, and the value distribution may vary locally. For such situations, and also to accommodate updates more easily, sometimes a per-block dictionary is used [Poess and Potapov 2003]. Dictionary compression normally lends itself to optimizing queries by rewriting predicates on strings into predicates on integers (which are faster), but this is easiest to accomplish for the situation with a global dictionary.

One practical point of consideration is how to dictionary-compress efficiently, which depends on fast hashing. One particularly fast technique is cuckoo hashing [Zukowski et al. 2006].

- **Frame Of Reference (FOR).** If the column distribution has value locality, one may represent it as some constant base plus a value. The base may hold for an entire disk block, or for smaller segments in a disk block. The value then is a small integer (which takes fewer bits to store than larger integers); hence the physical representation of a block of FOR values is the base followed by one small integer for each tuple [Goldstein et al. 1998]. For example, the sequence of values: 1003, 1001, 1007, 1006, 1004 can be represented as: 1000, 3, 1, 7, 6, 4. Frame Of Reference can also be combined with delta coding, where the current value is represented as a delta with respect to the preceding value. This is especially useful when the next value is strongly correlated with the preceding value. One typical example is inverted list data which consists of ascending integers.

- **The Patching Technique.** Dictionary and FOR compression rates suffer if the domain of values becomes too large, or has outliers, respectively. However, if the frequency of the distribution is skewed, then we can still compress the data if the compression is done only for the most frequent values.

A simple extension to both FOR and Dictionary encoding is to allow so-called exception values which are not compressed. The exception technique is typically implemented by splitting a disk block into two parts that grow towards each other: the compressed codes at the start of the block growing forward, and an error array at the end growing backwards. For tuples encoded as exception values, the compressed code would be a special escape. Checking for this escape with an if-then-else, however, constitutes a difficult to predict branch in the very kernel of the algorithm, which does not run well on modern CPUs (branch mispredictions). The patching technique [Zukowski et al. 2006], rather than storing escape values in the codes, uses these to maintain a linked list. Decompression first compresses all codes regardless of the exceptions. In a second step, the linked list is traversed and the exception values are “patched into” the decompressed output. While doing more work than naive testing for escapes, patched decompression performs better, by separating the problematic branch from the main work. The patch technique can also be considered an example of the algorithmic optimization opportunities provided by block-wise processing.

### 1.3.3 Operating directly on compressed data.

In many cases, the column-oriented compression algorithms discussed above (in addition to some of the row-oriented algorithms) can be operated on directly without decompression. This yields the ultimate performance boost, since the system saves I/O by reading in less data but does not have to pay the decompression cost. This benefit is magnified for compression schemes like run length encoding that combine multiple values within a column inside a single compression symbol. For example, if a run-length encoded column says the value “42” appears 1000 times consecutively in a particular column for which we are computing a SUM aggregate, the operator can simply take the product of the value and run-length as the SUM, without having to decompress.

However, operating directly on compressed data requires modifications to the query execution engine. Query operators must be aware of how data is compressed and adjust the way they process data accordingly. This can lead to highly non-extensible code (a typical operator might consist of a set of ‘if statements’ for each possible compression type). One solution to this problem is to abstract the general properties of compression algorithms in order to facilitate their direct operation so that operators only have to be concerned with these properties. This allows new
compression algorithms to be added to the system without adjustments to the query execution engine code. This is done by adding a component to the query executor that encapsulates an intermediate representation for compressed data called a compression block. A compression block contains a buffer of column data in compressed format and provides an API that allows the buffer to be accessed by query operators in several ways. Compression blocks do not necessarily have a mapping to storage blocks. In fact, a compression block can be quite small in its representation footprint (e.g., a single RLE triple) in general; a storage block can be broken up into multiple compression blocks. These compression blocks expose key properties to the query operators. For example, RLE and bit-vector blocks tend to describe a list of locations for a single column value. A query operator such as a count aggregation operator simply needs to call the getSize() method from the API of the compression block, without having to iterate through block. Properties that are highly relevant to many query operators are isSorted(), isPositionContiguous(), and isOneValue(). Based on these properties, query operators can elect to extract high level information about the block (such as getSize(), getFirstValue(), and getEndPosition()) instead of iterating through the compression block, one value at a time.

By abstracting away the key properties of compression schemes that enable direct operation on compressed data, the query operators do not need to be changed when an additional compression scheme is added to the database system. If an engineer desires to add a new compression scheme, the engineer must implement an interface that includes the following code:

(a) code that converts raw data into a compressed representation

(b) code that breaks up compressed data into compression blocks during a scan of compressed data from storage

(c) code that iterates through compression blocks and optionally decompresses the data values during this scan

(d) values for all relevant properties of the compression algorithm that is exposed by the compression block, and

(e) code that derives the high level information described above (such as getSize()) from a compression block.

Results from experiments in the literature show that compression not only saves space, but significantly improves performance. However, without operation on compressed data, it is rare to get more than a factor of three improvement in performance [Abadi et al. 2006]. Once the query execution engine is extended with extensible compression-aware techniques, it is possible to obtain more than an order of magnitude improvement in performance, especially on columns that are sorted or have some order to them.

1.3.4 Late Materialization

In a column-store, information about a logical entity (e.g., a person) is stored in multiple locations on disk (e.g. name, e-mail address, phone number, etc. are all stored in separate columns), whereas in a row store such information is usually co-located in a single row of a table. However, most queries access more than one attribute from a particular entity. Furthermore, most database output standards (e.g., ODBC and JDBC) access database results entity-at-a-time (not column-at-a-time). Thus, at some point in most query plans, data from multiple columns must be combined together into ‘rows’ of information about an entity. Consequently, this join-like materialization of tuples (also called “tuple construction”) is an extremely common operation in a column store.

Naïve column-stores [Halverson et al. 2006; Harizopoulos et al. 2006] store data on disk (or in memory) column-by-column, read in (to CPU from disk or memory) only those columns relevant for a particular query, construct tuples from their component attributes, and execute normal row-store operators on these rows to process (e.g., select, aggregate, and join) data. Although likely to still
outperform the row-stores on analytical workloads like those found in data warehousing, this method of constructing tuples early in a query plan (“early materialization”) leaves much of the performance potential of column-oriented databases unrealized.

More recent column-stores such as MonetDB/X100, C-Store, Vertica, Ingres VectorWise, and to a lesser extent, Sybase IQ, choose to keep data in columns until much later into the query plan, operating directly on these columns. In order to do so, intermediate “position” lists often need to be constructed in order to match up operations that have been performed on different columns. Take, for example, a query that applies a predicate on two columns and projects a third column in the same table after the predicates have been applied. In a column-store that uses late materialization, the predicates are applied to the column for each attribute separately, and a list of positions (ordinal off sets within a column) of values that passed the predicates is produced. Depending on the predicate selectivity, this list of positions can be represented as a simple array, a bit string (where a 1 in the $i$-th bit indicates that the $i$-th value passed the predicate) or as a set of ranges of positions. These position representations are then intersected (if they are bit-strings, bit-wise AND operations can be used) to create a single position list. This list is then sent to the third column to extract values at the desired positions. The advantages of late materialization are four-fold [Abadi et al. 2007].

- First, selection and aggregation operators tend to render the construction of some tuples unnecessary. Therefore, if the executor waits long enough before constructing a tuple, it might be able to avoid the overhead of constructing it altogether.

- Second, if data is compressed using a column-oriented compression method (that potentially allow compression symbols to span more than one value within a column, such as RLE), it must be decompressed during tuple reconstruction, to enable individual values from one column to be combined with values from other columns within the newly constructed rows. This removes the advantages of operating directly on compressed data, described above.

- Third, cache performance is improved when operating directly on column data, since a given cache line is not polluted with surrounding irrelevant attributes for a given operation [Ailamaki et al. 2001]. This is particularly important as the bandwidth between memory and CPU increasingly becomes a bottleneck in modern computing systems. For example, when applying a predicate evaluation operation in the where clause (such as WHERE salary > $100,000), memory bandwidth is not wasted shipping other attributes from the same set of tuples to the CPU, since only the salary attribute is relevant for that particular operator.

- Fourth, the vectorized optimizations described above have a higher impact on performance for fixed-length attributes. In a row-store, if any attribute in a tuple is variable width, then the entire tuple is variable width. In a late materialized column-store, fixed-width columns can be operated on separately.

Despite all the reasoning above, late materialization can sometimes be slower than early materialization (especially if a naive implementation is used). For example, if a predicate is used that is not restrictive (e.g. WHERE salary > $100 AND age > 5 AND ...) on many attributes within a tuple, the process of intersecting large amounts of positional intermediate data (one for each predicate applied) and then extracting and materializing a large percentage of tuples in the table that pass all the predicates is more costly than simply constructing tuples and avoiding all the positional calculations inherent in the late materialization model.
It turns out that careful implementation of late materializations using internal data structures such as multi-column blocks [Abadi et al. 2007] or vector blocks [Boncz et al. 2005] can significantly reduce the space of situations where late materialization is outperformed by early materialization. A multi-column block contains a cache-resident, horizontal partition of some subset of attributes from a particular relation, stored in their original compressed representation (see Figure 7). Multi-column blocks allow predicates to be applied to the compressed representation of each column involved in a query predicate separately, and the position list results from the predicate application are pipelined to an intersection operator that intersects them (while they are still in cache) and outputs the result to a position descriptor (shown to the left of the Figure 7) indicating (using a bit vector in this example) which tuples passed all predicates. This data structure can be passed to higher level operators for further processing.

Although multi-column blocks do not eliminate the need for intersecting positions, only small subsets of each column are operated on at once, allowing the pipelining of predicate evaluation output (position lists) directly to the intersection operator, and therefore enabling the construction of tuples to occur while the values that are going to be stitched together are still in cache. This allows late materialization to outperform early materialization for all but the most extreme queries [Abadi et al. 2007] as long as no joins are involved. However, late materialized joins can be problematic without further optimizations, as will be discussed in the next section.

1.3.5 Joins

Join operators present a plethora of opportunities for performance improvements in column-stores, but these opportunities can also lead to bottlenecks and complexities if not dealt with appropriately. If an early materialization strategy is used relative to a join, tuples have already been constructed before reaching the join operator, so the join functions as it would in a standard row-store system and outputs tuples (yielding the same performance profile as a row-store join). However, several alternative algorithms can be used with a late materialization strategy. The most straightforward way to implement a column-oriented join is for (only) the columns that compose the join predicate to be input to the join. The output of the join is a set of pairs of positions in the two input relations for which the predicate succeeded. For example, the figure below shows the results of a join of a column of size 5 with a column of size 4:

\[
\begin{array}{cccc}
42 & 36 & \times & 38 \\
42 & 44 & 38 & 42 & 46 & 36
\end{array}
\begin{array}{c}
1 \\
2 \\
4 \\
3 \\
2 \\
5 \\
1
\end{array}
\]
For many join algorithms, the output positions for the left (outer) input relation will be sorted while the output positions of the right (inner) input relation will not. This is because the positions in the left column are often iterated through in order; while the right relation is probed for join predicate matches. For other join algorithms (for example, algorithms that sort or repartition both sets of input) neither position list will be sorted. Either way, at least one set of output positions will not be sorted. Unsorted positional output is problematic since typically after the join, other columns from the joined tables will be needed (e.g., the query:

```sql
SELECT emp.age, dept.name
FROM emp, dept
WHERE emp.dept_id = dept.id
```

requires the age column to be extracted from the emp table and the name column to be extracted from the dept table after performing the join). Unordered positional lookups are problematic since extracting values from a column in this unordered fashion requires jumping around storage for each position, causing significant slowdown since most storage devices have much slower random access than sequential.

Luckily, there have been several improvements proposed in the research literature to avoid this problem of jumping around in storage to extract values at an unordered set of positions. One idea is to use a “Jive join” [Li and Ross 1999; Tsirogiannis et al. 2009]. For example, when we joined the column of size 5 with a column of size 4 above, we received the following positional output:

```
1 2 3 4 5
```

The list of positions for the right (inner) table is out of order. Let’s assume that we want to extract the customer name attribute from the inner table according to this list of positions, which contains the following four customers:

```
Smith
Johnson
Williams
Jones
```

The basic idea of the Jive join is to add an additional column to the list of positions that we want to extract, that is a densely increasing sequence of integers:

```
1 2 3 4 5
```

This output is then sorted by the list of positions that we want to extract (this sort causes the newly added column to now be out of order):

```
1 1 2 2 4 4
```
The columns from the table are then scanned in order, with values at the (now sorted) list of positions extracted and added to current data structure.

```
1 4  Smith
2 1  Johnson
2 3  Johnson
4 2  Jones
```

Finally, the data structure is sorted again, this time by the column that was added originally to the join output, to revert the current data structure back to the original join order (so as to match up with join output from the other table).

```
2 1  Johnson
4 2  Jones
2 3  Johnson
1 4  Smith
```

This algorithm allows all columns to be iterated through sequentially, at the cost of adding two sorts of the join output data. The cost of these additional sorts increases with the size of the join output (i.e., the number of tuples that join). Since most database systems have a fast external sort algorithm implemented (that accesses the input as sequentially as possible), this algorithm can cause significant performance improvements relative to the random access that would result from the more naive implementation of a late materialized join described above.

Further research has resulted in additional improvements to the above algorithm. It turns out that a complete sort is not necessary to reduce random access performance overhead in value extraction of join output. This is because most storage media are divided into contiguous blocks of storage, and random access within a block is significantly cheaper than random access across blocks. Therefore, the database does not need to completely sort the position list before using it to extract values from columns; rather, it just needs to be partitioned into the blocks on storage (or an approximation thereof) in which those positions can be found. Within each partition, the positions can remain unordered, since random access within a storage block is much cheaper (e.g., the difference between memory and disk I/O, or the difference between cache and memory I/O). The column from which we are extracting values is therefore accessed in block order, but not in exact position order. The Radix Join [Boncz et al. 1999] is an example of a late materialized join along these lines, and provides a fast mechanism for both performing the partitioning of column positions into blocks before the column extraction, and reordering the intermediate data back to the original join order after the extraction has occurred, as long as all data involved are from fixed-width columns.

In practice, due to the additional engineering complexity, many commercial column store implementations do not implement pure late-materialized joins, despite the promising experimental results presented in the literature of the above-described algorithms. Instead, for join algorithms that iterate through the left (outer) input in order and probes the right (inner) input out of order, a hybrid materialization approach is used. For the right (inner) table, instead of sending only the column(s) which compose the join predicate, all relevant columns (i.e., columns to be materialized after the join plus the predicate column) are materialized before the join and input to the join operator, while the left (outer) relation sends only the single join predicate column. The join result is then a set of tuples from the right relation and an ordered set of positions from the left relation; the positions from the left relation are used to retrieve additional columns from that relation and complete the tuple construction process. This approach has the advantage of only materializing values in the left relation corresponding to tuples that pass the join predicate while avoiding the penalty of materializing values from the right relation using unordered positions. For join algorithms that iterate through both input relations out of order, both relations are materialized before the join.
Multi-column blocks (described above) provide an alternative option for the representation of the right (inner) relations. Instead of materializing the tuples of the inner table, the relevant set of columns are input to the join operator in a sequence of multicoloum blocks. As inner table values match the join predicate, the position of the value is used to retrieve the values for other columns (within the same block), and tuples are constructed on the fly. This technique is useful when the join selectivity is low and few tuples need to be constructed, but is otherwise expensive, since it potentially requires a particular tuple from the inner relation to be constructed multiple times.

1.3.6 Inserts/Updates/Deletes

Columnar stores in addition to vertical fragmentation make heavy use of compression, and may also store multiple table replicas or projections in different value orders, all to enhance analytical query performance. The challenge facing updates in column stores is that one I/O per column (replica) is typically needed. For example, a single-row insert can be handled in a row-store with a single disk I/O, but will lead to many disk I/Os in a column store (one for each attribute in the tuple that is being inserted). Even if a user wants to insert many tuples at once, these disk I/Os are scattered (random) I/Os, because of the ordered or clustered table storage. Finally, compression makes updates computationally more expensive and complex since data needs to be decompressed, updated and re-compressed before being written back to disk. Extra complications occur if the updated data no longer fits the original location.

Some analytical columnar database systems, such as C-Store [Stonebraker et al. 2005], handle updates by splitting their architecture into a “read-store” that manages the bulk of all data and a “write-store” that manages updates that have been made recently. Consequently, all queries access both base table information from the read store, as well as all corresponding differences from the write-store and merge these on-the-fly (a MergeUnion against insert, and MergeDiff against deletes). In order to keep the write-store small (it resides typically in RAM), changes in it are periodically propagated into the read-store.

A natural approach to implement the write-store is to store differences (insert, delete, modify) in an in-memory table, maybe even in row-format. The disadvantage of storing deltas in separate tables is that every query must perform a full merge between the read-store and the differential table. Some systems (e.g. C-Store) employ optimizations, improving the performance of value-based delta structures. For example, it is often possible to perform (parts of) a query on the read-store and delta data separately and only combine the results at the end. Also, deletions can be handled by using a boolean column marking the “alive” status of a given tuple, stored in RAM, using some updatable variant of the compressed bitmap index.

The VectorWise system uses a novel data structure, called Positional Delta Trees (PDTs) in which to store differences. The key advantage is that merging is based on knowledge of the position where differences apply, and not on the sort key of the table, which can be composite and complex. When a query commits, it immediately finds out which table positions are affected. As such, it moves the activity of merging from query time to update time, which fits the agenda of read-optimized processing. In contrast, without PDTs, we would resort to CPU-costly MergeUnion/MergeDiff processing that needs to be repeated by all queries. Additionally, it makes each query read the sort key columns, leading to additional I/O if these attributes were otherwise not required for answering the query. Keeping track of positions in an ordered table is tricky, as inserts/deletes halfway change the position of all subsequent tuples. The PDT is a kind of counting B-tree that allows keeping track of positions under logarithmic update cost.

Differential data structures such as PDTs, but also previous approaches like differential files, can be layered: one can create deltas on deltas on deltas, etc. This hierarchical structure can also be exploited in the hierarchical memory architecture of computers, e.g. placing very small deltas in the CPU cache, larger ones in RAM, and huge deltas on disk or on solid state memory. Additionally, layered deltas are a tool for implementing isolation and transaction management. The idea is that a new transaction adds an initially empty top-level PDT to the layer of PDTs already
present. By sharing the immutable lower-level, bigger, PDTs, this provides cheap snapshot isolation. As the transaction makes changes, these get added to this top-level PDT, which effectively captures the write-set of the transaction. The algorithms to keep PDT position tracking consistent under concurrent transactions were shown in [Heman et al. 2010] to be exactly those required to implement optimistic concurrency control. The phenomenon that column-stores when storing updates in differential data structures, thereby also obtain a structure for supporting transaction management, carries over to different approaches than PDTs.
2. Identifying RDF Potentials in MonetDB

In this section we identify five aspects of the MonetDB architecture that are likely beneficial for RDF stores in general and Virtuoso in particular.

The first two of these aspects have been covered in detail in the primer on column stores (Section 1), and have already very early in the LOD2 project been selected for immediate implementation. More details on this implementation can be found in Section 3.

The other three aspects are advanced topics and have not been described in Section 1, and therefore are described here in some detail.

2.1 Columnar Compression

The feature has been described in detail in Section 1.1.1 and Section 1.3.2 and will not further be discussed.

2.1.1 Potential for RDF stores

We note that RDF stores typically keep data organized in multiple indexes, which is essentially equivalent to keeping data in multiple orders (a la C-Store). Hence, it follows that for RDF columnar storage will most likely bring strong reduction in storage size. As column-stores typically compress better and at lower computational encode/decode cost, significantly higher benefits both in space and performance can be obtained thanks to compression, than in comparison with row-stores.

Apart from compression, the question arises if RDF stores would benefit from columnar decomposition from a data access point of view. On first sight, given that triple stores store only three columns, the advantage observed in analytical database systems of handling queries on wide tables that access only a few columns, does not apply. The benefits of columnar storage can hence be sought more in vectorized execution that goes naturally with column stores: systems that consume columns at-a-time will more readily use block-wise processing which benefits raw computational power and also decompression speed.

As such, it seems in place also to consider PAX: from the computational point and compression point of view it brings the advantages of columnar storage, but it makes sure that fine-grained point queries only generate a single I/O per access.

2.2 Vectorized Execution

The feature has been described in detail in Section 1.3.1 and will not further be discussed.

2.2.1 Potential for RDF stores

Vectorized execution can accelerate 10-100x a CPU-bound query processor, for that reason alone it is already interesting for RDF processing.

Vectorized processing goes hand in hand with efficient decompression, and also provides algorithmic optimization opportunities in many algorithms. One interesting aspect unexplored in MonetDB/X100 which lack B-trees is vectorized B-tree access [Zhou and Ross, 2003], is also interesting from the point of view of RDF stores.
2.3 Run-Time Sample-Driven Query Optimization

Relational database systems have been introduced in the 70’s, and since then the optimization of queries submitted to a database system has been extensively researched resulting in the proposal of a multitude of techniques. While sufficient for some applications, the widely used type of optimizers are not always robust, and in some cases pick execution plans that are far from optimal. The reasons behind the shortcomings of classical optimizers are the following: (i) they depend on statistics and a cost model which are often inaccurate, not up-to-date, and sometimes even absent, (ii) they fail to detect correlations which can unexpectedly make certain plans much cheaper than others, (iii) they cannot efficiently handle the large search space of big queries. The challenges faced by traditional relational optimizers and their impact on the quality of the chosen plans are aggravated in the context of XML and RDF. This is due to the fact that in XML, and even more so in RDF, statistics should capture, in addition to the value of the nodes, the structure of the data. Moreover, the search space of plans for an XML and RDF queries is on average larger than that of relational queries. This is due to the higher number of joins.

2.3.1 ROX: Runtime Optimization of XQueries

To overcome the above challenges, we propose an optimizer that satisfies the following properties: autonomy from statistics and cost model, robustness in always finding a good execution plan, and efficiency in exploring the search space. Our approach is to adopt an optimizer with a fundamentally different internal design which moves the optimization to run-time, and interleaves it with query execution. As such sampling techniques can be used to accurately estimate the cardinality and cost of operators without depending on any statistics and cost model. To detect correlation among the queried data, we introduce the chain sampling technique which we believe to be the first generic and robust method to deal with any type of correlated data. We suggest exploring the search space by interleaving optimization and execution steps, defining the plan incrementally, i.e. the plan is built step by step. The exploration is performed efficiently through the chain sampling technique.

Figure 8: An illustration of the steps of ROX: optimization and execution steps are alternated until all operators in the join graph are executed. Every optimization consists of sampling path segments in the join graph until a "superior" path is found. The execution phase executes the join operators in the chosen path and materializes the results. Therefore, ROX defines the execution plan step-by-step, with each optimization phase shaping one section of the plan, and each execution phase directly evaluating the operators in the newly defined section. When all operators in the graph are executed, ROX returns a relation containing all the tuples satisfying the join operations in the graph.

This idea was first pioneered in the context of XML in the ROX project [Abdel Kader et al, 2010]. ROX is the first run-time optimizer for XQueries. It interleaves optimization and execution steps where each optimization phase initiates a sampling-based search for the superior sequence of operators. The subsequent execution step executes the chosen sequence and materializes the
results, allowing the next optimization phase to benefit from the newly materialized intermediates, and the knowledge which can be extracted from it. ROX has been implemented on top of the MonetDB database system, and experiments have been conducted which have shown that ROX is indeed robust and efficient, and performs better than and relational compile-time optimizers.

In ROX, the execution plan is generated step-by-step: every optimization phase shapes one section of the plan by determining the best sequence of operators to be executed. To produce cheap execution plans, a strategy followed by most database systems is to reduce as much as possible the number of tuples generated and flowing through the operators in the plan. This basically translates into first executing the operators that keep the output small. Using sampling techniques, ROX can estimate the weight (i.e. result size) of each edge in the join graph and then pick out the one with the smallest output cardinality for evaluation. However, we realize that executing the operator corresponding to the spotted edge directly is a greedy decision, and can lead to an execution plan that is far from optimal. Determining the edge with the smallest weight is in fact equivalent to finding a local minimum in the search space of the join graph. However, there might exist a sequence of edges in the graph which forms a global minimum, and thus generates a smaller number of intermediary tuples. To detect the existence of such paths (sequences of joins), the optimization phases of ROX adopt a chain sampling technique which uses the local minimum as starting point and then invests a small amount of time to climb the hill searching for a global minimum. Before we explain the chain sampling technique in more detail, we give the following two definitions:

**Superiority of a path:** Given a set S of paths, a path p is said to be superior to all other paths in S if the execution of p followed by the execution of any path p’ in S is cheaper than executing first p’ and then p.

**Absolute superiority of a path:** Given a set S of paths, a path p is said to be absolutely superior to all other paths in S if the execution of p followed by the execution of any path p’ in S is cheaper than executing p’ alone.

We note that absolute superiority implies superiority, that is a path p that is absolutely superior to all other paths in S is also superior to all paths in S. We also stress that the set S does not include all possible paths in the search space of the join graph. Moreover, the comparisons between pairs of paths can be implemented efficiently such that little time is spent on the checking for the superiority and absolute superiority of a path.

### 2.3.2 Chain Sampling in ROX

Given the edge e with the smallest weight, chain sampling is a process exploring the path segments (sequences of joins) around e, in search for a path segment that is superior to all the explored paths in the join graph including the edge e. When such a path is found, the path is returned for execution. The starting point of the exploration process is the edge e: given a sample chosen from one of the vertices v of e, chain sampling samples iteratively in a breadth-first manner the sequences of unexecuted edges branching from v, until detecting the path p. During every sampling iteration, new path segments are defined by extending previously explored paths with an additional newly sampled edge. The selectivity of an explored path segment is estimated by consecutively sampling the edges along the path, using the output of the sampling operation of one operator as input to the sampling operation of the subsequent operator. By sampling ahead in the branches, ROX may discover that a path, due to correlations, produces a result of much lower cardinality than the initially predicted estimations, and hence proving to be superior to others. We illustrate the chain sampling process with the following example.

**Example.** Consider the join graph in Figure 9a, we suppose that the edge in the join graph with the smallest weight is (v2, v5), and we choose the vertex v2 to be the starting point of the chain sampling exploration. Therefore, chain sampling will explore iteratively, in a breadth-first-manner the path segments branching from vertex v2. The edges in Figure 9b, Figure 9c, and
Figure 9d are labelled with the path id to which they belong, and the arrows denote the direction of sampling (i.e. the left and right operands of the sampling operation). Figure 10 enumerates the edges sampled at each iteration, and illustrates the creation of paths.

Figure 9: Illustration of chain sampling. The starting point of exploration is the edge with the smallest weight which we suppose to be \((v_2, v_5)\). A sample table chosen from the vertex \(v_2\) is used to sample the surrounding path segments of unexecuted edges. The sampling is performed iteratively in a breadth-first manner. The labels on the edges denote the path id(s) to which the edges belong, and the arrows indicate the sampling direction (i.e. the left and right operands of the sampling operation).
• **Iteration 1** (Figure 9b): During the first iteration, all the unexecuted edges branching from v2 are sampled using as input a sample set chosen from v2. Since the number of outgoing unexecuted edges is three, three new paths p1, p2, and p3 are created. Paths p1, p2, and p3 contain respectively the sampled edges (v2, v1), (v2, v3), and (v2, v5).

• **Iteration 2** (Figure 9c): The second iteration samples the next unexecuted edges branching from each of the three defined paths. From each unexecuted edge e branching from the path p, a new path p’ is created. Path p’ contains all the edges in p in addition to the newly sampled edge e. For the previously explored paths p1, p2, p3, we have the following:
  1. Path p1: No unexecuted edge is branching from the end vertex v1 of p1, and therefore no new path is created.
  2. Path p2: Path p2 has one unexecuted edge (v3, v4) branching from its end vertex v3. A new path p4 is created: p4 = {(v2, v3), (v3, v4)}.
  3. Path p3: Two unexecuted edges (v5, v6) and (v5, v7) branch from the end vertex v5 of path p3. Therefore, two paths are created: p5 = {(v2, v5), (v5, v6)} and p6 = {(v2, v5), (v5, v7)}.

• **Iteration 3** (Figure 9d): In the third iteration, the only remaining edge to sample is (v7, v8). The edge is branching from path p6, and therefore a new path p7 containing the edges in p6 and the newly sampled edge (v7, v8) is created. Note that the edge (v6, v9) branching from path p5 is not sampled since it has already been executed in a previous execution phase of ROX.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Sampled Edges</th>
<th>Defined Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration 1</td>
<td>(v2, v1), (v2, v3), (v2, v5)</td>
<td>p1 = {(v2, v1)}, p2 = {(v2, v3)}, p3 = {(v2, v5)}</td>
</tr>
<tr>
<td>Iteration 3</td>
<td>(v7, v8)</td>
<td>p7 = {(v2, v8), (v5, v7), (v7, v8)}</td>
</tr>
</tbody>
</table>

Figure 10: The sampled edges and definition of paths at every iteration of the chain sampling process illustrated in Figure 9. At every iteration, new paths are created by extending previously explored paths with newly sampled edges.

Chain sampling does not necessarily sample all the unexecuted edges in the join graph before deciding about the superior path to return for execution. In fact, at every iteration, the chain sampling process notes down the characteristics of the newly created paths (e.g., cost and selectivity of the operators in the paths), and compares the observed properties of all paths sampled so far to decide if one path is so selective that the exploration can be safely halted. The detected selective path is then returned for execution.

To assist chain sampling in comparing the paths and deciding whether to proceed with or stop the exploration process, a stopping condition has been formulated. The stopping condition identifies the existence of a sampled path segment p that is absolutely superior to every other explored path in the join graph. The stopping condition guarantees that even if any of the explored paths other than p is extended with highly selective edges, p remains absolutely superior to the newly extended paths, making it safe to stop the chain sampling exploration. If the stopping condition succeeds in finding a path p that is absolutely superior to all the other explored paths, then the chain sampling process is halted and p is returned for execution. As we have noted earlier, p is also superior to all the explored paths.
2.3.3 Pipelined Execution

The original variant of ROX, named ROX-full, is proposed in the context of database systems that support full materialization of intermediates. Since most used database systems nowadays adopt a pipelined execution scheme, we designed ROX-sampled, a variant of ROX-full, that is suitable for the aforementioned database architecture. The two ROX variants are similar, but contrary to ROX-full, ROX-sampled uses only sampled data throughout the whole algorithm, hence consuming and generating a small number of tuples at every optimization and execution step.

![Figure 11: An illustration of the steps of ROX-full and ROX-sampled. Both techniques define the execution plan iteratively, but each execution phase in ROX-full processes full tables while execution phases in ROX-sampled process only data samples. As a result, in ROX-full, the plan is defined and executed incrementally, and in ROX-sampled, the execution plan is first entirely defined and then evaluated.](image)

Although only data samples are used throughout the whole ROX-sampled algorithm and the operations performed during its execution phases are cut-off-sampled, its performance is comparable to that of ROX-full especially when using a relatively larger sample size (equal to 1000 tuples). It has been shown that ROX-sampled is a robust optimizer which succeeds in choosing (near-) optimal plans, and is capable of detecting and exploiting different types of correlations.

Summarizing, ROX-sampled is a generalization of the ROX approach, allowing the large number of existing pipelined database systems to integrate the ROX idea into their optimization paradigm. ROX-sampled has shown to be a robust and efficient optimizer with a performance closely comparable to that of ROX-full.

2.3.4 Potential for RDF stores

In RDF storage that store data in some triple (or quad) table (or index) in some relational backend or storage manager, the original graph structure of the data is lost in the sense that it is not part of the schema. Instead, the graph structure is turned into data correlations.

For instance, in RDF-H, each ORDER subject will join with average 6 LINEITEM subjects, each CUSTOMER subject may join with hundreds of ORDER subjects, but each NATION subject may join with millions of CUSTOMER subjects. In effect, the association of one subject with some property (identifying it as NATION, ORDER, LINEITEM or CUSTOMER) may influence the fan-out ratio of other properties, or even influence the fan-out multiple property steps away.

Tracking these kinds of correlations to make correct optimization decisions is not supported in relational query optimizers, as the huge amount of possible correlations is too large. This is the basic reason why even low- to medium-size SPARQL queries easily run aground: each binding pattern generates a self-join, but for the relational query optimizers all relational self-joins are the same as it does not detect the correlations with e.g. selection clauses. Instead, the optimizer just
estimates selectivities for the eaves in the query graph (selection predicates) and uses heuristics based on the shape of the query graph or global statistics on property occurrences to select a query plan. As it misses data correlations due to the graph structure generally, in comparison with plans generated for relational queries, chances of the optimizer choosing a bad plan are significantly higher. As anecdotal evidence of this problem, the first time the BSBM-BI query mix was tested\(^3\), all RDF stores encountered problems with bad query plans, even though the BSBM-BI query mix contains BI queries of relatively simple structure. Early experience with the RDF-H equivalent of TPC-H, experienced this problem even more strongly (in almost all queries).

The fact that ROX is able to establish the fan-put of joins on-the-fly and using its sampling will detect correlations makes ROX an interesting technique for RDF stores.

ROX is able to detect join fan-out multiple steps away (chain sampling) as well as correlations, and can therefore be used to find better RDF plans. However, in case of long linear path chains (ignoring star-patterns just fetching simple properties of subjects), there are in effect only limited options an optimizer can choose from.

In the case of XML, ROX relied on initial statistics provided by element indices and was able to navigate not only through structural joins following the XML structure, but also had per-document value indices at its disposal. In RDF, the value indices and the element indices find their equivalent in the multiple \((P,O,G,S)\) index orders maintained. The missing piece here is the equivalent to structural joins that will run as efficiently as the staircase join run for XML in MonetDB (essentially minimal time: \(O(|\text{input}|+|\text{output}|)\)). Even in case of multiple materialized indices, massive B-tree lookup cost is going to be significantly higher than experienced in MonetDB and XML.

Additionally, the graph structure of the data may cause explosions of intermediate results if only simple navigation can be used. For instance, if two groups of nodes are only sparsely connected, but both have many outgoing paths into other directions, simple navigation will explore them all before finding the small result set. One way to deal with this is to keep multi-step join path indices, especially for those cases where the result sets are limited and the path appears relevant in a workload. But, finding such interesting paths to cache is too complex to do by hand. The techniques around automatic caching of intermediate subexpressions (Recycling, see Section 2.4) may be a technique that could create such a pool of path indices dynamically, ready for exploration by a ROX-like optimizer.

---

\(^3\) [http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/results/V6/index.html#expBI](http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/results/V6/index.html#expBI)
2.4 Recycling Intermediate Results

[Ivanova et al. 2009] showed that workloads which exhibit common sub-expressions can benefit from **recycling** intermediate and final query results. Recycling is a refinement of operator caching and materialized views. In contrast to materialized views, a resource pool of recycled intermediate results adapts continuously to the workload without DBA intervention and incurs minimal start-up and maintenance costs.

2.4.1 MonetDB Implementation

The idea was first tested using a workload from the SkyServer\(^4\) application, a complex scientific database portal, to illustrate the impact of recycling intermediate results in a real-world scenario. These experiments showed that recycling intermediate and final results can improve the response time and throughput of a workload. However, a workload can only benefit from recycling if the initial materialization cost of each result in a Recycler Cache and the overhead introduced by the recycling system is outweighed by the evaluation cost saved by reusing materialized results while they are in the cache.

![Figure 12: SkyServer workload benefits with either Credit-based or Keepall admission](http://cas.sdss.org)

**And 1GB/unlimited Recycler Cache sizes, for query sequences with update-caused Recycler Cache flushes splitting the 100query sequence in 1, 2 or 4 batches.**

The implementation of the recycler required a modification in the MonetDB query execution engine. Its abstract machine interpreter was hooked up with a recycler optimizer and runtime module. The optimizer marks operations of interest for harvesting. The runtime support uses this advice to manage a pool of partial results. It avoids re-computation of common sub-queries by extracting readily available results from the pool.

The key issue in the design of the recycler is to identify efficient and effective policies to use and maintain the resource pool. It encompasses decisions in three dimensions: instruction matching, investment cost versus savings, and pool administration maintenance. For each instruction to be executed the recycler performs a matching process, i.e., it searches for a possible reusable relational algebra operation in the recycle pool. For each operation executed the recycler decides if it is beneficial to keep the result. Finally, to prevent the pool of intermediates becoming a resource bottleneck itself, operations with low potential for reuse should be cleaned from the pool to reduce the memory usage and the search time.

\(^4\) http://cas.sdss.org
2.4.2 Recycling for Pipelined RDBMS

It is important to remark that the architecture of MonetDB - where recycling was first proposed - significantly differs from commercial state-of-the-art database systems. Instead of relying on a pipelined query evaluation approach like the widely used pipelined operators that implement an iterator, MonetDB uses the operator-at-a-time paradigm which materializes complete intermediate results as a by-product of every step of the query execution. Therefore, intermediate results are already available in main memory and the Recycler only has to decide which ones to submit to the Recycler Cache. In pipelined systems, materializing results is an explicit action that slows down a query and consumes additional RAM. In the sequel, we describe recent work in the context of the VectorWise system that showed that it is also worthwhile to use recycling in pipelined database systems, though careful workload monitoring is needed to enable the system to avoid wasting system resources and slowing down queries through materializing results that will never be reused.

The recycler cache is an in-memory cache of finite size that contains materialized intermediate and final results from previous query evaluations which have been deemed beneficial enough for caching. Each result is stored as a non-persistent in-memory table. However, simpler structures with less overhead than tables are also conceivable (e.g. a list of arrays). The content of the recycler cache is managed by an admission and a replacement policy. The admission policy selects results from currently executing queries for materialization and adds them to the cache once their result has been produced. Once the recycler cache is full, the replacement policy evicts materialized results in favour of new results that are deemed more beneficial.

Both the admission and replacement policies utilize a benefit metric to make sure that the cache always contains the results that are considered most beneficial at that time. The benefit metric consists of the following elements:

- an estimate of the expected amount of times the cached result will be re-used
- (an estimate of) the computational cost to compute the subquery.
- (an estimate of) the size of the subquery result.

Note that if we detect that a subexpression is executed for a second time, we do have exact information on both the size and the computational cost (even though it was not materialized the first time), as all queries in the workload are remembered in a recycler tree, which is the query graph that contains all queries seen in the workload so far, with common subexpressions eliminated. The recycler tree serves as a structural representation of previous query invocations. Its purpose is two-fold:

- it is used to match the query tree of the current query with query trees from previous queries in order to find materialized results that can be used to answer the current query. Materialized results correspond to those nodes in the recycler tree, for which a result is available in the Recycler cache.
- to find subexpressions in the current query that could be beneficial to materialize in order to answer future queries. That is, if we detect that a subexpression is now occurring for the second time, its value is bigger. In general, occurrence in the workload is evidence that increases the estimated number of reuses of a result.

Each node in the recycler tree represents a relational algebra operator and contains parameters that further specify that operator. For instance, one such operator might be a selection and its parameter the selection predicate it evaluates. Each node is annotated with statistics such as number of references, execution time that have been collected during previous query evaluations. These statistics are utilized to compute the benefit of the result produced by the execution plan corresponding to that node and its subtree.

Two subtrees corresponding to partial plans that produce the same results are termed exactly matching. Partial plans are shared in the recycler tree and are stored only once; therefore, nodes
of the tree can have multiple parents. A single virtual node acts as the sole root of the tree. An example recycler tree is shown on the right side of Figure 13.

The identification of exactly matching nodes is central to the structure of the recycler tree as they ensure a single instance of each subtree, thereby reducing the cost of finding an exactly matching subtrees in the recycler tree because there is at most one Shared subtrees additionally reduce the recycler tree's memory footprint. The recycler tree can furthermore be utilized to find materialized results that subsume a result in the query.

The recycler interacts with query evaluation through *query rewrite rules* and *Store* operators. The rewriter rules are used to match incoming query trees with the recycler tree to identify results worth materializing and materialized results that can be used to answer the query. Store operators are pipelined execution operators that can either buffer, materialize or just pass along the tuples produced by their input operators without interrupting the tuple flow in the execution pipeline.

The rewriting rules in question work as follows. First, a bottom-up rewriting rule traverses the query tree and identifies tree operators that have an exact match in the recycler tree. Operators that do not have an exact match are inserted into the recycler tree.

A second, a top-down rewriting rule then checks if the first rewriting rule has identified sharing opportunities. If matches have been identified whose result is already in the recycler cache, the operators in the query tree are substituted with an operator that uses the result from the cache instead of recomputing it from scratch.

The rule furthermore injects Store operators on top of operators that have a match in the recycler tree and are deemed beneficial enough for materialization and on top of all operators that have no match in the recycler tree because the rule does not have the required information to decide on materializing the results of these operators and the decision therefore has to be postponed into the execution stage.
The query builder then turns the query tree into an executable plan of operators. Once the query starts being executed, Store operators that have already been selected for materialization materialize their input. Store operators that have not been chosen for materialization yet buffer their input tuple flow and use dynamic estimates to decide whether materializing the result is beneficial or not. If the result is not deemed beneficial, the store operator cancels buffering and just passes its input tuples along to its parent. After the query has been executed, each operator annotates its equivalent node in the recycler tree with measured run-time parameters. The entire process is outlined in Figure 13.

2.4.3 Ongoing Research

Recycling in pipelined database systems is an active topic of research at CWI and VectorWise. Various policies have been designed and evaluated:

- policies that base exclusively on workload history evidence to make materialization decisions. This means that the earliest time a result will be materialized is the second time it occurs in a workload.
- Speculative policies, that depending on query structure are capable to materialize results even the first time they occur. Such a policy tends only to be beneficial for very small results that have a high computational cost (e.g. Aggregates).
- Proactive policies that detect common patterns in sub-queries that slightly differ from each other, and issue a slightly more expensive query that subsumes these multiple subexpressions.

Apart from the SkyServer evaluation, experiments are also targeting the TPC-H decision support benchmark, and specifically the throughput run in TPC-H, where many similar (but not exactly equal) queries are run concurrently.

2.4.4 Potential for RDF stores

Recycling is a general technique that can be applied directly also in RDF stores.

RDF stores manage typically a single table, and each SPARQL typically results in a query plan with many self-joins. Even small queries easily lead to more than 10 such joins, and large queries can even generate hundreds. To execute such self-joins at any form of efficiency, it is important to have all the POGS columns somehow indexed. This indexing is sometimes done with B-trees (or simply ordered tables, in column stores). Only the selections in a query plan with restrictions on both P and O (aka selections) lead to efficient sequential access to sorted indices. All the subsequent self-joins lead to random accesses. This is the reason that RDF stores live and die by being memory resident.

In order to reduce the amount of self-join, it has been proposed to reduce the amount of self-joins, by creating path indices (pre-materialized partial joins) over certain often used paths. However, the selection of such paths presents a workload analysis problem. Also, in many cases users are focusing on certain subsets of subjects (e.g. only recent orders, or orders from a certain type or originating from certain departments) to start the join with. Hence, creating a path index for all orders may be wasteful.

We see recycling as a way to create such partial path indices in a fully self-tuned way. Note also that the presence of such partial path indices (aka the Recycler Cache) should be taken into account in the query optimization phase. In particular, in case of run-time query optimization, the contents of the recycler cache could be another great source of alternative access paths to sample potential query strategies and derive join fan-out and selectivity estimations.

Caching and re-using intermediates is planned to be investigated in LOD2 under Task 2.4: Adaptive Caching of Intermediate Joins and Inferences.
3. Enhancing Virtuoso

Shortly before the official start of the LOD2 project, work has at OGL started to incorporate vectorized execution (Section 2.2) and Compressed Column Storage (Section 2.1) and, as a single project for the upcoming v7 release of Virtuoso Universal engine. CWI has provided feedback and also assessed the source code to provide advice as part of this process, which is still ongoing.

Note that vectorized execution can be combined with row-storage, but column stores without vectorized execution make little sense. Both were developed at the same time, but columnar storage in Virtuoso needed quite a bit of specific work and features. Start of 2011, early beta versions of Virtuoso version 7 have been released that use vectorized execution on the old row-store representation of the data. By July 2011, all needed features for columnar storage have been completed such that recent betas offer columnar storage as an option. Virtuoso version 7 has not been released yet, as other system features, such as cluster execution, are still in the quality control pipeline; however its release is expected in 2011 still.

This section provides an overview of what has been done so far, and an outlook on further steps.

3.1 Virtuoso Column Store and Vectored Execution

The Virtuoso work on column-wise compressed storage and vectored execution is motivated by a pursuit of performance for schema-last data models such as RDF. The intuition is that since a triple or quad oriented data model is already in principle property-centric, just as a relational column store is, then such a model, if properly implemented, ought to achieve performance characteristics approaching those of a relational column store.

3.1.1 Comparison With Previous Column Store Work

Relational column stores have usually been optimized for queries touching a large fraction of the rows on a table, while quite often touching only a fraction of the columns. The typical workload consists of bulk loads followed by queries, with trickle updates following. Schemes that store data in load/append order (MonetDB) or keep ordered structures with deltas periodically merged into the main persistent storage (Vertica, VectorWise) are common. Page sizes are typically large, i.e. from 64K to 4512K or more. Value-based indices are possible but are not an integral part of many designs.

The Virtuoso column store is intended for managing multi-part value based indices stored in a column-wise compressed format. The motivation for specifically going for sorted indices is twofold. Firstly, an RDF workload typically stores all data in a single table, thus even in applications where a relational column store can justify scanning a table for hash join build, a quad or triple store cannot justify this as easily, since typically a small fraction of the table would in fact be selected while all of it would be scanned.

- Therefore, even with extensive use of hash joins, an ordered storage format is needed so as to limit scans.
- Secondly, sorted data compresses much better than unsorted data. Especially with a quad or triple-based data model, compression is important since space overheads would otherwise be prohibitive.
- Further, many RDF workloads that we have observed in fact consist of small lookup queries for which a value based index representation is far better than alternatives.

Some column stores (InfoBright, VectorWise) implement a system of column summaries where a separate data structure holds summary information such as minimum and maximum value for a range of rows of a particular column. This can substitute for value based indices especially
when the value of the column correlates with the ordering of the column. Virtuoso does not at present implement such a mechanism but could easily do so.

The Virtuoso column store coexists with the Virtuoso row store. In fact, a single table can have indices of which some are stored column-wise and some row-wise. The transaction semantics for both row and columns are identical. In both cases, the database only contains clustered indices where the index leaves of the primary key store the dependent columns. Secondary indices refer to the row on the primary key by value, by including the necessary parts of the primary key.

The column-wise index starts with a row-wise sparse index with one entry per every few thousand rows. The set of rows per entry of the top level index is called a segment. The columns in the segment are then stored contiguously in any number of compression entries. These compression entries are stored on pages, with each column of a segment consisting of an integer number of compression entries. Each compression entry specifies the compression. Compression types include:

- run length,
- run length with deltas,
- integers with deltas,
- dictionaries,
- arrays of fixed length integers,
- arrays of variable length items.

Compression entries like integer arrays and "run length + delta" enforce a homogeneous data type on the content whereas other formats like dictionary and variable length array allow for run time types. In this way, there is no need to declare data types or preferred compression formats since the data at hand always determines what compression is applied. Also overheads associated with run-time typing are incurred only in the actually few cases where heterogeneous types occur next to each other.

Compression works best when contiguous values come from the same property, hence there is an extra benefit to having P as the leading key part in the RDF indices.

Virtuoso uses unusually small pages for a column store (8K). This has the advantage of efficiently supporting random order inserts in place. Quite often, a single row insert will only need to shift data inside a single compression entry, at most a few hundred bytes. In an append-mostly column store with few or no value-based secondary indices this issue does not present itself but in the case of storing triples in multiple index orders the convenience of a random inserts is emphasized.

In the case of an insert a compression entry may grow or split and in turn lead the page in question to split if there is no more space. This in turn may lead the segment to split, leading to the need for an extra row in the row-wise sparse index. This in turn may lead the index leaf space to split, up towards the index root, B-tree fashion.

This complex hierarchy of nested splits is relatively hard to implement but once functional it performs well and reliably. We note that there are extra constraints, for example column pages referred from one index leaf page may not be referred from another. This needs to be enforced when splitting index leaf pages. The rationale for this is that any column-wise insert that does not split the row-wise index leaf must be capable of being handled without affecting other index leaves. This is important for keeping granularity of concurrency control down and for keeping the already many special cases of splitting at a manageable level.
3.1.2 Bulk Load Concurrency

The typical RDF bulk load situation has a mostly ascending insert sequence on one key and a primarily random insert order on another. To get reasonable throughput, the bulk load must run on as many threads as possible. To minimize page access contention, small pages are advantageous and limiting all non-splitting inserts to within a single row-wise leaf page is advantageous. Note that in a typical RDF use case, a single row-wise leaf page will have about 70-100 segments, each with 4000 to 8000 logical rows. Thus even a single row-wise leaf page is relatively coarse for concurrency control (often 500000 rows or more).

Random inserts and consequent page splits will result in half-filled pages. To ameliorate this, the system performs from time to time a recompression, merging logically adjacent partly filled pages. This applies as well to column pages as to row-wise index leaves and to the row-wise sparse index level immediately above the leaves.

This operation is Virtuoso's analog of the Vertica write-optimized store merge or of the VectorWise checkpoint. We note that the recompression is local and easily parallelizable and does not interfere with concurrency since it only touches pages with no present activity and primarily from the less recently touched end of the buffer pool.

3.1.3 Vectored Execution

Vectored execution means that the query execution engine always deals with sets of multiple intermediate result tuples. The benefits of this have been extensively explained in Section 2.2.

While this technique is most often found with column stores, it is likewise applicable to row stores, also with significant benefit. We can say that a row store can sometimes live without vectored execution because a single row access is relatively simple but a column store is more heavily dependent on this technique because the single row access overhead is relatively larger, hence more in need of amortization through getting multiple rows in a single operation.

Converting a row-wise tuple at a time engine into a vectored one is relatively straightforward. Unlike most RDBMS execution engines which have a pull-based iterator model where the query is like an expression tree with the final select at the top, Virtuoso implements a bottom-up model where operators form a pipeline where the previous stage calls the next. MonetDB is in a sense similar in that it too compiles queries into a series of instructions executed from start to end but Virtuoso differs from this in that it does not materialize full intermediate results but rather pipelines execution row by row or vector by vector.

The vectored upgrade consists of introducing two new types of query variable, called vector and reference. The vector is an array of consecutive values, most often homogenous but also heterogeneous values are possible since all parts of the engine support columns where each row is separately typed at run time. As a query is a sequence of operators of which each gets \( n \) rows of input from which each may produce 0 or more rows of output, vectors assigned by different stages of the operator pipeline are not aligned. For this purpose, there is the reference query variable type.

Consider the SQL query:

```sql
select p_partkey, sum (l_extendedprice) 
from part, lineitem
where l_partkey = p_partkey and p_name like ' %green%'
group by l_partkey option (order, loop);
```

The option at the end means join order is left to right and joins use index and not hash.
The group by operation must reconstruct a tuple of p_partkey and l_extendedprice. It does so by using an array of parameter row numbers assigned by the join to lineitem, where each row of output contains the row number of the corresponding row of input, thus this can be used for finding which row of the partkey vector corresponds to which of the l_extendedprice vector.

Such chains of indirections may be multiple operations deep. No data is copied for purposes of alignment and the indirections may be traversed in a vectored manner, resolving the corresponding row of input at an arbitrary prior stage of the pipeline for many rows of results at the same time.

3.1.4 Vectored Expressions and Procedures
Virtuoso compiles an expression into a series of instructions operating on query variables. Instructions include three-address operations for arithmetic, function call, conditional branches etc. Some of the query variables have vectors as values, whereupon the instruction is executed on all the values simultaneously, producing a vector of results. This eliminates most of the interpretation overhead.

A condition may evaluate differently for different elements of the input vectors. To this effect, a conditional piece of code is evaluated with a bitmap indicating which vector elements are to be considered for the instructions. This also applies to stored procedures. If the procedures contain SQL operations like insert or select, these operations get a vector of inputs, also if executed inside conditional sections. In this manner all the benefits of vectored execution are kept for procedure code also.

3.1.5 Vectored Index Lookup
The more items one looks up in an index at a time, the cheaper the lookup cost per item will be. For large joins, once the items to be fetched are pre-sorted in the order of the index, consecutive lookups will hit consecutive places in the index, so that consecutive lookup targets will frequently reside on the same page or at least on sibling pages of the index. We note that index lookup keys are often integers or in RDF applications IRI ID's, which are cheap to sort using digit sort, making the expense of sorting the keys almost linear. Even though digit sort is in principle linear, a non-linearity is introduced if it starts missing the CPU cache. This is however more than compensated for by having better cache locality on the index itself and by having to rarely make index lookups all the way from the index top to the leaf.

3.1.6 Dynamic Vector Size
The usefulness of vectored index lookup depends on frequently finding consecutive hits on the same or at least sibling pages. Getting larger batches of input to the join operator increases the likelihood of this. Therefore Virtuoso constantly measures how many rows of data it has seen on index pages visited during a vectored lookup and compares this to the number of entries selected. If at the end of processing a batch of lookups it turns out that hits were sparse, the operator will request the operators upstream of it to make a larger batch next time, in the hope of increasing hit density. This is seen to have significant performance benefits, up to doubling or even tripling the throughput of a join.

3.1.7 Vectored Insert
Columnar inserts into ordered data lead to multiple update locations, and cause extra I/O with respect to row-inserts. Also, compression means that data needs to be moved and recompressed. Vectorizing the insert process, and inserting as many tuples that are located near each other in one go can help amortize these overheads.

When multiple index entries are inserted in a batch, the operation begins as a vectored lookup that finds the page on which the first insert falls. After this, there is a lookup local to the page for
finding whether any other insert positions in the batch fall on the page. The inserts are then
applied in a single merge pass over the page.

The vectored insert on a column-wise page begins similarly, locating the segment into which
the first insert falls. The vectored search inside the segment finds the insert positions for this
segment and in so doing consumes the corresponding number of search parameters. When the
positions that fall on the segment are known, there is a check for positions that fall between the
end of the segment and the start of the next. If the segment in question is the last of the page, the
parent page is located to see what the first value of the next segment is. In this manner
consecutive inserts to any place in the index are always serviced by a single lookup.

Once the insert positions are known, the compression entries corresponding to these in each
column of the segment are updated. There is a format-specific function for inserting values without
full recompression.

In some cases, for example when more rows are inserted than are already in the compression
entry, a recompression is cheaper anyway.

The segment receiving the insert will split if its post-insert row count is greater than a set limit,
typically 16K rows, or if its longest column takes more than a set number of pages, typically 16
pages. A single insert may insert up to 48K rows into a segment, as the row number within the
segment is 16 bits. Thus a segment may split into more than two parts after an insert.

After a segment is updated or split, the row-wise leaf page may also split. This happens in
the usual B-tree manner but we note some special cases for the columns involved. Two segments
whose row-wise leaf rows are on different row-wise leaf pages may not refer to the same page of
column data, hence the column pages that fall at the boundary, after the last segment of the left
side of the row-wise leaf split, must also be split.
3.2 Performance results

In the following, we provide a preliminary report on the performance improvements achieved by the new compressed column storage and vectorized execution.

3.2.1 BSBM

We use the Berlin SPARQL Benchmark (BSBM V3.1) using a dual Xeon 5520 (2.2GHz, 1333MHz DDR3, 8 cores with SMT, 16 threads total) with 72GB of RAM. At the time of writing, only the vectorized row-execution experiments had been completed, so regrettably the column-storage experiments are missing here.

<table>
<thead>
<tr>
<th>BI 100Mt</th>
<th>tup</th>
<th>vec</th>
</tr>
</thead>
<tbody>
<tr>
<td>bi-power</td>
<td>100%</td>
<td>223%</td>
</tr>
<tr>
<td>bi-tput</td>
<td>100%</td>
<td>983%</td>
</tr>
<tr>
<td>bi-tput (x8)</td>
<td>486%</td>
<td>1433%</td>
</tr>
</tbody>
</table>

Table 2: BSBM BI Use case

The Business Intelligence use case of BSBM provides a modes 2.2x improvement for row-wise vectored execution. In the throughput, the benefits are more pronounced (10x) especially in case of a single client. Virtuoso 6 got a 5x boost from using 8 clients, and this increases to a 14x boost with vector row-wise.

<table>
<thead>
<tr>
<th>explore</th>
<th>100Mt-tup</th>
<th>100%</th>
<th>385%</th>
</tr>
</thead>
<tbody>
<tr>
<td>100Mt-vec</td>
<td>154%</td>
<td>946%</td>
<td></td>
</tr>
<tr>
<td>1Gt-tup</td>
<td>100%</td>
<td>141%</td>
<td></td>
</tr>
<tr>
<td>1Gt-vec</td>
<td>354%</td>
<td>1394%</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: BSBM Explore Use case

In the Explore use case we see that row-vectoring in the sequential case delivers a 3.5x improvement on large data (1gt), whereas the benefit is modest on smaller data (1.5x on 100Mt). Whereas parallel query does not bring more than 1.5-4x in tuple-at-a-time row-format (385% on small data and 141% on the 1Gt dataset), the amortization of index lookups reduces lock contention in such that parallel scaling improves by 3x (385% vs. 946%) and 10x (141% vs. 1394%) using the vector-row implementation on resp. small and large data. Generally speaking, for row-storage we see modest improvements due to vectored execution, trending to more significant on larger data and using parallelism.

**Column Version.** We currently miss a full BSBM evaluation using column storage, but can provide some initial data points. TheExplore use case for 1Gt gets a further improvement of 20% over row-vectorized using 16 threads (1700% vs 141%, hence **12x** faster vs row-tuple). Also, the load speed of column-wise storage is higher; whereas the row-format achieves an 130K/s triple input rate, this is 150K/sec for columns. The space occupancy dramatically lowers from 27 bytes/triple in row-format to 7 bytes/triple for column format. Note that all above benchmarks are on hot data, where all the accessed portions of the database fit in RAM. Performance on workloads not fitting in RAM is an order of magnitude slower even if Fash drive are used (and multiple orders of magnitude slower on magnetic disk). Thus, the new vectorized-column-wise Virtuoso7 is not only **4-12x** faster than Virtuoso6, but can also efficiently handle datasets that are **4x** larger.

In the next section, we provide micro-benchmark results that assess join performance comparing vector size, and column vs. row format.
3.2.2 Join Micro-Benchmark

Table 4 gives execution times for a join with different vector sizes and compares row-wise and column-wise indices. The join does 40 million random lookups with a single integer key, so that all lookups find one row and the lookup keys are given with no order and no locality.

The experiment is run with different numbers of threads to measure whether the system will get bound by aggregate memory throughput, cutting down on exploitation of multiple cores. The system tested is again the dual Xeon 5520 (2.2GHz, 1333MHz DDR3, 8 cores, 16 threads total).

All performance is normalized to tuple-at-a-time (tup, i.e. vector size=1) using row-storage. All sequential results (green) compare to the sequential row-tup experiment which ran in 133 seconds. All parallel results (red) using 16 threads on 8 cores compare to the parallel row-tup experiment, which ran in 30 seconds. Hence the baseline speedup thanks to parallelism was 4.5x and when comparing sequential with parallel one should multiply difference by this factor 4.5.

<table>
<thead>
<tr>
<th>joinbench</th>
<th>row</th>
<th>row (x16)</th>
<th>col</th>
<th>col (x16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tup</td>
<td>100%</td>
<td>100%</td>
<td>91%</td>
<td>76%</td>
</tr>
<tr>
<td>vec-10K</td>
<td>165%</td>
<td>125%</td>
<td>498%</td>
<td>428%</td>
</tr>
<tr>
<td>vec-100K</td>
<td>251%</td>
<td>171%</td>
<td>1076%</td>
<td>818%</td>
</tr>
<tr>
<td>vec-1M</td>
<td>492%</td>
<td>265%</td>
<td>1115%</td>
<td>476%</td>
</tr>
<tr>
<td>vec-auto</td>
<td>477%</td>
<td>250%</td>
<td>1011%</td>
<td>468%</td>
</tr>
<tr>
<td>seq</td>
<td>1079%</td>
<td>374%</td>
<td>2492%</td>
<td>1349%</td>
</tr>
<tr>
<td>hash-best</td>
<td>973%</td>
<td>208%</td>
<td>1006%</td>
<td>336%</td>
</tr>
<tr>
<td>hash-worst</td>
<td>380%</td>
<td>70%</td>
<td>476%</td>
<td>158%</td>
</tr>
</tbody>
</table>

Table 4: Row-Store vs. Column-Store Execution
with varying vector size (1..1M, “tup”=1),
with and without parallelism (1 vs. 16 threads), and
comparing to a join with a sequential access pattern (seq)
and various Hash Join alternatives.

We see that column-wise indices perform slightly worse than row-wise when hits are sparse (91% and 76%) and tuple-at-a-time execution is used. This is not unexpected, as locality and amortization of e.g. column compression needs a vectorized approach.

Generally though, we can say that the new columnar storage can bring up to 11x (1115%) performance advantage, and using the default dynamic vector size (auto) is 10x. Using parallel execution, the benefit with respect to parallel row tuple-wise execution is lower but still substantial: as much as 8x (818%) and still 4.5x with the automatic vector setting. The numbers seem to suggest tuning the automatic vector size down with the parallelism degree.

Interestingly, hash-based joins, which are supported in SQL but not generated for SPARQL plans in Virtuoso, indeed seem hard to tune and not better performing. For sequential execution, picking the best hash strategy with proper tuning parameters leads to 10x speedup which is on-par with the vectorized nested-loop index lookup joins, but when parallel threads are used performance breaks down. This is unexpected but may be accounted for by the fact of the column-wise lookup making better use of CPU cache, whereas the hash join misses the cache on virtually each lookup.

There is also a comparison with sequential lookups. The baseline Virtuoso V6 performance (row-tuple) for random lookup used be 10x worse than sequential lookup (merge-join) in-RAM performance, but vectorized column storage performance is now within a factor 2.5x of merge-join.
3.2.3 RDF-H

One basic premise of the LOD2 project is that a schema-last data model like RDF must be applicable to mainstream database workloads without prohibitive cost in space or performance. In order to explore this, we have translated the industry standard TPC-H benchmark to RDF, which we call RDF-H. The data is generated with the TPC supplied dbgen tool and trivially converted to RDF. The queries are straightforwardly translated to SPARQL. Since Virtuoso offers both SPARQL and SQL, we can run the same logical queries against the same data in two different data models and observe the differences. We use the Virtuoso column store with SQL, storing the TPC-H data in a TPC-H rules compliant layout with multicolumn tables and secondary indices on only l_partkey and o_custkey.

With RDF-H, the data is stored in a column-wise quads table with the standard indices on PSOG, POGS and distinct projections on SP, OP and GS.

Some queries like Q2 and Q8 run in almost the same time in SQL and SPARQL. Other queries like Q1 take almost 9 times longer in SPARQL. The variation from query to query is substantial. Some queries like Q18 that depend on ordered aggregation in SQL are relatively worse still with SPARQL since the query optimizer does not recognize the ordered aggregation opportunity in the SPARQL version.

All TPC-H queries have been mapped to SPARQL, and can be found on:

http://sourceforge.net/p/bibm/code/15/tree/trunk/bibm/tpch/sparql/

Below is the SPARQL text for TPC-H Q2 translated to SPARQL.

```sparql
define sql:signal-void-variables 1
prefix rdfh: <http://lod2.eu/schemas/rdfh#>

select
  ?s_acctbal,
  ?s_name,
  ?nation_name,
  ?p_partkey,
  ?p_mfgr,
  ?s_address,
  ?s_phone,
  ?s_comment
where {
  ?ps a rdfh:partsupp;
      rdfh:ps_has_supplier ?supp;
      rdfh:ps_has_part ?part ;
      rdfh:ps_supplycost ?minsc .
  ?supp a rdfh:supplier ;
      rdfh:s_acctbal ?s_acctbal ;
      rdfh:s_name ?s_name ;
```


```sql
 rdfh:s_has_nation ?s_has_nation ;
 rdfh:s_address ?s_address ;
 rdfh:s_phone ?s_phone ;
 rdfh:s_comment ?s_comment .
?s_has_nation rdfh:n_name ?nation_name ;
 rdfh:n_has_region ?s_has_region .
?s_has_region rdfh:r_name "REGION%" .
?part a rdfh:part ;
  rdfh:p_partkey ?p_partkey ;
  rdfh:p_mfgr ?p_mfgr ;
  rdfh:p_size %SIZE% ;
  rdfh:p_type ?p_type .
{ select ?part min(?s_cost) as ?minsc
 where {
  ?ps a rdfh:partsupp ;
   rdfh:ps_has_part ?part ;
   rdfh:ps_has_supplier ?ms ;
   rdfh:ps_supplycost ?s_cost .
  ?ms rdfh:s_has_nation ?m_has_nation .
  ?m_has_nation rdfh:n_has_region ?m_has_region .
  ?m_has_region rdfh:r_name "REGION%" .
  } }
  filter (?p_type like "TYPE")

} order by
  desc (?s_acctbal)
  ?nation_name
  ?s_name
  ?p_partkey
limit 100
```
3.3 Virtuoso Innovation Roadmap

Our initial look at RDF-H leads to two immediate conclusions:

- The query plan space explodes as we get up to 18 self-joins inside a select.
- Cardinality estimates fail when what used to be a single table lookup becomes a join.

In the following two sections we propose means of dealing with these phenomena.

3.3.1 Query Optimization and Dealing with Plan Space Explosion

SPARQL queries against a triple model database can result in several times more joins than the semantically equivalent query against a database with multicolored tables. The search space for all permutations of join order is in principle the factorial of the number of tables being joined, hence it is crucial to drastically prune this. This requirement is not very strong on relational TPC-H, where there are only 6 or 7 tables being joined in each query, going through even all the permutations is affordable. However, when moving to SPARQL, we find 18 tables joined in the SPARQL version of Q2, not counting tables in subqueries.

Due to this, it becomes especially important to make a good initial guess for a join order so as to interrupt any plan generation that exceeds this cost. Further, it is important to recognize sequences of joins that have any of the following characteristics:

- Retrieving leaf attributes that are only returned and not used for further joining, for example ?s_acctbal, ?s_name, ?p_partkey, in the previous Q2. Since each supplier is known through sampling to have one s_acctbal and one s_name, it makes no sense to try plans that permute these two join steps.

- Cardinality reducing join sequences. For example, the join from s_nation to n_region to r_name with the name given as a constant is known to be cardinality reducing because r_name is known to have 5 distinct values of which one is selected whereas each supplier is known to have one nation and each nation to have one region. This knowledge is never explicitly stated but is derived from sampling the data. When there is a clear cardinality reducing join sequence, trying other join orders for the constituent tables makes no sense. Also, if a partial plan allows placing such a sequence next, it does not make sense to try to place anything else before this known cardinality reducing sequence.

By recognizing these patterns, the join order search space is brought down closer to the original relational situation. These techniques are not RDF specific and are equally justified in SQL applications but there the need for these is often not as great.

3.3.2 Sampling

Virtuoso normally stores quads first ordered on P. Thus, consecutive values of O are likely of the same type and at least on the POGS index also ordered. This layout makes it easy to measure the cardinality of each predicate, i.e. how many distinct O’s there are for one S and vice versa. This gives a fair estimate of join selectivity as long as the P is known. Through the same sampling we also see how many quads there are for a given P. This may vary greatly, especially with web data.

This sampling is done periodically, triggered by a need to know and is refreshed if there are significant numbers of inserts or deletes on the index in question. Further, when literal values are specified in queries for either S or O, a sample is taken at query optimization time to see the selectivity of the condition. A cache of recent samples is kept, since it is likely that consecutive queries in a session will involve the same literals.
3.3.3 Quads and Hash Joins
A hash join is advantageous specially when joining a large table to a small one where most rows of the large table do not find a hit. In relational cases, the opportunity is relatively easy to detect based on statistics. With triples this is harder. The cost of using a hash join where it in fact does not fit is high, since this entails building a potentially large hash table.

Virtuoso has previously not used hash join for RDF data. With Virtuoso 7, there is a new implementation of hash join using cuckoo hash. This is limited to running in memory but is capable of partitioned hash join, i.e. only part of the build relation need be materialized at one time, running the rest of the query for each partition of the build side of the hash join. The previous hash join implementation could overflow to disk but did not have partitioned hash join.

Further, the SQL optimizer before Virtuoso 7 did not create plans where results of a join were used on the build side of a hash join. Thus, the build was always from a single table scan, optionally with a selection.

The Virtuoso 7 optimizer does make plans with complex operations on the build side of hash joins. For example, the SPARQL Q2 above will generate a plan where a hash table is built containing all the suppliers in a nation of the given reason. This is then used for selecting qualifying rows of the partsupp relation. The same technique applies equally to SQL and SPARQL.

However, use of this technique entails an increased risk if the join on the build side turns out at run time not to be selective. This makes the matter still problematical for SPARQL usage.

3.3.4 Optimizing for Order
Consider RDF-H Q1, a scan of many properties of lineitem with a non-selective condition on date. In SQL, the choice of a plan is clear, as there is not even an index on the date, thus we get a scan of the table with selection on the date and then aggregation for the rest. In SPARQL where there is an index on every O value, we get a plan that uses the POGS index for resolving the condition on the date and then we get values of S in no specific order and with no locality.

This makes the other lookups spread at random over the data. If we however recognize that the selection on the date is not selective, we can scan this in order of PSOG, in which case we get the S values in order, creating locality on the other lookups. This second plan runs 4x faster than the previous one.

This is however hard to detect at query optimization time.

3.3.5 The Need for Deep Sampling
We have briefly examined situations where SPARQL is at a disadvantage as opposed to SQL for creating efficient query plans. It is possible that complex rules could be devised for getting the right plan on any given set of queries, but the broader applicability of these would be uncertain and their maintenance difficult.

Therefore we propose to solve this matter by actually partly executing candidate plans in the cost model. Hence, an approach patterned after ROX-sample, the pipelineable version of the Runtime Query Optimization approach outlined in Section 2.3. This is in concept also an extension of the sampling already performed but in implementation terms will be quite different.

We have already above outlined ways of restricting the plan space, which here is doubly essential since the cost for assessing any given plan is likely to go up. Further, we must recognize small queries where the investment in sampling is not justified. This can straightforwardly take place by first trying the most selective condition and if the estimated fan-out stays under the millions of rows there will be no justification for sampling, thus the use of the technique can be limited to cases where this is justified. The data structures used for a partial plan in the cost model and those for an executable plan are different but they are largely isomorphic. Thus, the two
structures can be generated together and even modified together to reflect the traversal of the plan space.

When a new join step is added for evaluation by the cost model, this is just appended to the executable plan. The executable plan will have been executed up to this point, thus variable bindings will exist and only the newly added fragment needs to be executed.

This works well with vectorized execution, since each operator produces a potentially large batch of results, up to millions of rows. Thus generating a single vector worth of results from each step will be sufficient. In order to deal with different data distributions in different parts of a large scan, the scan can be randomized, for which there already exists logic. Expected benefits include solution for the all the aforementioned optimization problems, i.e.

- always correct join order, for example the above Q2 could be evaluated starting at region, going to nation, then supplier etc. Not making this plan depends on knowing that the like on the p_name is selective enough. In SPARQL, this involves fetching strings for the O values, which is already a join step. Thus this is not covered by single level sampling.
- deciding whether to use hash join, having a reliable estimate of the cardinality of the build side and the selectivity of the join. Further we also know the cardinality of the probe side, the higher this is, the better the hash join tends to be in relation to alternatives.
- deciding on index traversal order while keeping in mind the locality of subsequent joins. Since these joins are actually performed, the locality or lack thereof will be known and another traversal order may be tried if needed.
- Estimating cardinality and values of aggregations. Getting the right plan for TPC-H Q18 depends on guessing that the IN predicate with an aggregate subquery on the right hand will be selective. This subquery selects a percent or so of all orders based on the sum of their l_quantity being greater than a threshold. Without sampling, making a guess for this is near impossible.

This technique will be prototyped in the first quarter of 2012.

3.3.6 Compilation Possibilities

We see in [Sompolski et al, 2011] that a combination of vectoring and compilation can bring substantial benefits.

As noted, expressions, hash joins and aggregations are natural places for compilation, as the arity (number of columns) and their data types vary from query to query

Index lookups do not offer as much opportunity for compilation but even there functions for sorting the input for a join and functions for comparing row-wise keys can be generated as needed. We see from experiments with row-wise index random access that a key-layout specific comparison function is up to 30% faster than a generic one.

Further compilation opportunities are found with filtering functions for column-wise compressed data, where the selection predicate can operate directly on the compressed data and where multiple predicates on the same column can be compiled together. Again, the selection of predicates is large and not anticipatable. We note that a cardinality-reducing hash join is also such a predicate, as in invisible join [Abadi et al, 2008]. We also see that such functions must come in many variants, depending on whether they are the first selection criterion (accepting a range of rows as input) or a subsequent criterion accepting a set of selected discrete row numbers as input.
4. Bibliography


BRUNO, N. 2009. Teaching old elephant new tricks. In CIDR.


SOMPOLSKI, J., ZUKOWSKI, M., BONCZ, P. Vectorization vs. compilation in query execution, 2011. In DaMoN 11: Proceedings of the 7th international workshop on Data management on new hardware.


