Foundations and Trends® in Databases
Vol. 5, No. 3 (2012) 197–280

DOI: 10.1561/1900000024

The Design and Implementation of Modern Column-Oriented Database Systems

Daniel Abadi
Yale University
dna@cs.yale.edu

Peter Boncz
CWI
P.Boncz@cwi.nl

Stavros Harizopoulos
Amiato, Inc.
stavros@amiato.com

Stratos Idreos
Harvard University
stratos@seas.harvard.edu

Samuel Madden
MIT CSAIL
madden@csail.mit.edu

Contents

Abstract

In this article, we survey recent research on *column-oriented* database systems, or *column-stores*, where each attribute of a table is stored in a separate file or region on storage. Such databases have seen a resurgence in recent years with a rise in interest in analytic queries that perform scans and aggregates over large portions of a few columns of a table. The main advantage of a column-store is that it can access just the columns needed to answer such queries. We specifically focus on three influential research prototypes, MonetDB [46], VectorWise [18], and C-Store [88]. These systems have formed the basis for several well-known commercial column-store implementations. We describe their similarities and differences and discuss their specific architectural features for compression, late materialization, join processing, vectorization and adaptive indexing (database cracking).

D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos and S. Madden. *The Design and Implementation of Modern Column-Oriented Database Systems*. Foundations and Trends$^{®}$ in Databases, vol. 5, no. 3, pp. 197–280, 2012.

DOI: 10.1561/1900000024.

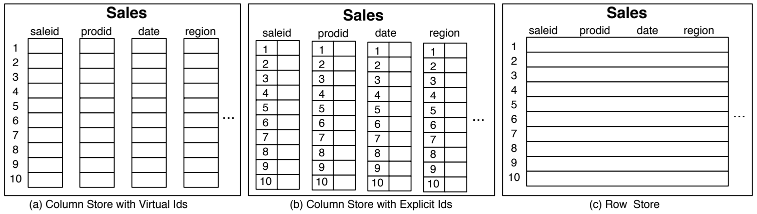
1

Introduction

Database system performance is directly related to the efficiency of the 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.

Column-stores. In recent years, there has been renewed interest in so-called *column-oriented systems*, sometimes also called *column-stores*. Early influential efforts include the academic systems MonetDB [46], VectorWise [18][1] and C-Store [88] as well as the commercial system SybaseIQ [66]. VectorWise and C-Store evolved into the commercial systems Ingres VectorWise [99] and Vertica [60], respectively, while by late 2013 all major vendors have followed this trend and shipped column-store implementations in their database system offerings, highlighting the significance of this new technology, e.g., IBM [11], Microsoft [63], SAP [26] and Oracle.

Column-store systems 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. A similar benefit is true while transferring data from main memory to CPU registers, improving the overall utilization of the available I/O and memory bandwidth. Overall, taking the column-oriented approach to the extreme allows for numerous innovations in terms of database architectures. In this paper, we discuss modern column-stores, their architecture and evolution as well the benefits they can bring in data analytics.



(a) Column Store with Virtual Ids

(b) Column Store with Explicit Ids

(c) Row Store

Figure 1.1: Physical layout of column-oriented vs row-oriented databases.

Data Layout and Access Patterns. Figure 1.1 illustrates the basic differences in the physical layout of column-stores compared to traditional row-oriented databases (also referred to as *row-stores*): it depicts three alternative ways to store a `sales` table which contains several attributes. In the two column-oriented approaches (Figure 1.1(a) and Figure 1.1(b)), each column is stored independently as a separate data object. Since data is typically read from storage and written in storage in blocks, a column-oriented approach means that each block which holds data for the sales table holds data for one of the columns only. 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 `prodid` and `date` columns, and only the data blocks corresponding to these columns would need to be read from storage (we will explain the differences between Figure 1.1(a) and Figure 1.1(b) in a moment). On the hand, in the row-oriented approach (Figure 1.1(c)), there is just a single data object containing all of the data, i.e., each block

[1] Initially named MonetDB/X100.

in storage, which holds data for the sales table, contains data from all columns of the table. In this way, there is no way to read just the particular attributes needed for a particular query without also transferring the surrounding attributes. Therefore, for this query, the row-oriented approach will be forced to read in significantly more data, as both the needed attributes and the surrounding attributes stored in the same blocks need to be read. Since data transfer costs from storage (or through a storage hierarchy) are often the major performance bottlenecks in database systems, while at the same time database schemas are becoming more and more complex with fat tables with hundreds of attributes being common, a column-store is likely to be much more efficient at executing queries, as the one in our example, that touch only a subset of a table's attributes.

Tradeoffs. There are several interesting tradeoffs depending on the access patters in the workload that dictate whether a column-oriented or a row-oriented physical layout is a better fit. If data is stored on magnetic disk, then if a query needs to access only a single record (i.e., all or some of the attributes of a single row of a table), a column-store will have to seek several times (to all columns/files of the table referenced in the query) to read just this single record. However, if a query needs to 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 a query needs to access a single record, only one seek is needed as the whole record is stored contiguously, and the overhead of reading all the attributes of the record (rather than just the relevant attributes requested by the current query) will be negligible relative to the seek time. However, as more and more records are accessed, the transfer time begins to dominate the seek time, 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.

Column-store Architectures. Although recent column-store systems employ concepts that are at a high level similar to those in early research proposals for vertical partitioning [12, 22, 55, 65], they include many architectural features beyond those in early work on vertical partitioning, and are designed to maximize the performance on analytic workloads on modern architectures. The goal of this article is to survey these recent research results, architectural trends, and optimizations. Specific ideas we focus on include:

- **Virtual IDs [46].** 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 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 1.1(a) vs Figure 1.1(b)). In some column-stores, each attribute is stored as a fixed-width dense array and each record is stored in the same (array) position across all columns of a table. In addition, relying on fixed-width columns greatly simplifies locating a record based on its offset; for example accessing the i-th value in column A simply requires to access the value at the location $startOf(A) + i * width(A)$. No further bookkeeping or indirections are needed. However, as we will discuss later on and in detail in Section 4, a major advantage of column-stores relative to row-stores is improved compression ratio, and many compression algorithms compress data in a non-fixed-length way, such that data cannot simply be stored in an array. Some column-stores are willing to give up a little on compression ratio in order to get fixed-width values, while other column-stores exploit non-fixed width compression algorithms.

- **Block-oriented and vectorized processing [18, 2].** 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.

- **Late materialization [3, 50].** Late materialization or late tuple reconstruction refers to delaying the joining of columns into wider tuples. In fact, for some queries, column-stores can completely avoid joining columns together into tuples. In this way, late materialization means that column-stores not only store data one column-at-a-time, they also process data in a columnar format. For example, a select operator scans a single column at a time with a tight for-loop, resulting in cache and CPU friendly patterns (as opposed to first constructing tuples containing all attributes that will be needed by the current query and feeding them to a traditional row-store select operator which needs to access only one of these attributes). In this way, late materialization dramatically improves memory bandwidth efficiency.

- **Column-specific compression [100, 2].** 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.

- **Direct operation on compressed data [3].** Many modern column-stores delay decompressing data until it is absolutely necessary, ideally until results need to be presented to the user. Working over compressed data heavily improves utilization of memory bandwidth which is one of the major bottlenecks. Late materialization allows columns to be kept in a compressed representation in memory, whereas creating wider tuples generally requires decompressing them first.

- **Efficient join implementations [67, 2].** Because columns are stored separately, join strategies similar to classic semi-joins [13] are possible. For specific types of joins, these can be much more efficient than traditional hash or merge joins used in OLAP settings.

- **Redundant representation of individual columns in different sort orders [88].** 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 application's query workload, substantial performance gains can be achieved. C-Store calls groups of columns sorted on a particular attribute *projections*. Virtual IDs are on a per-projection basis. Additionally, low-cardinality data that is stored in sorted order can be aggressively compressed.

- **Database cracking and adaptive indexing [44].** Database cracking avoids sorting columns up-front. Instead, a column-store with cracking can adaptively and incrementally sort (index) columns as a side-effect of query processing. No workload knowledge or idle time to invest in indexing is required. Each query partially reorganizes the columns it touches to allow future queries to access data faster. Fixed-width columns allow for efficient physical reorganization, while vector processing means that we can reorganize whole blocks of columns efficiently in one go, making adaptive indexing a realistic architecture feature in modern column-stores.

- **Efficient loading architectures [41, 88].** 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.

Are These Column-store Specific Features? Some of the features and concepts described above can be applied with some variations to row-store systems as well. In fact, most of these design features have been inspired by earlier research in row-store systems and over the years several notable efforts both in academia and industry tried to achieve similar effects for individual features with add-on designs in traditional row-stores, i.e., designs that would not disturb the fundamental row-store architecture significantly.

For example, the EVI feature in IBM DB2 already in 1997 allowed part of the data to be stored in a column-major format [14], providing some of the I/O benefits modern column-stores provide. Similarly, past research on fractured mirrors [78] proposed that systems store two copies of the data, one in row-store format and one in column-store format or even research on hybrid formats, such as PAX [5], proposed that each relational tuple is stored in a single page as in a normal row-store but now each page is internally organized in columns; this does not help with disk I/O but allows for less data to be transferred from main-memory to the CPU. In addition, research on index only plans with techniques such as indexing anding, e.g., [69, 25], can provide some of the benefits that late materialization provides, i.e., it allowed processors to work on only the relevant part of the data for some of the relational operators, better utilizing the memory hierarchy. In fact, modern index advisor tools, e.g., [21], always try to propose a set of "covering" indexes, i.e., a set of indexes where ideally every query can be fully answered by one or more indexes avoiding access to the base (row-oriented) data. Early systems such Model 204 [72] relied heavily on bitmap indexes [71] to minimize I/O and processing costs. In addition, ideas similar to vectorization first appeared several years ago [74, 85] in the context of row-stores. Futhrermore, compression has been applied to row-stores, e.g., [30, 82] and several design principles such as decompressing data as late as possible [30] as well as compressing both data and indexes [31, 47] have been studied.

What the column-stores described in this monograph contribute (other than proposing new data storage and access techniques) is an architecture designed from scratch for the types of analytical applications described above; by starting with a blank sheet, they were free to push all these ideas to extremes without worrying about being compatible with legacy designs. In the past, some variations of these ideas have been tried out in isolation, mainly in research prototypes over traditional row-store designs. In contrast, starting from data storage and going up the stack to include the query execution engine and query optimizer, these column-stores were designed substantially differently from traditional row-stores, and were therefore able to maximize the benefits of all these ideas while innovating on all fronts of database design. We will revisit our discussion in defining modern column-stores vs. traditional row-stores in Section 4.9.

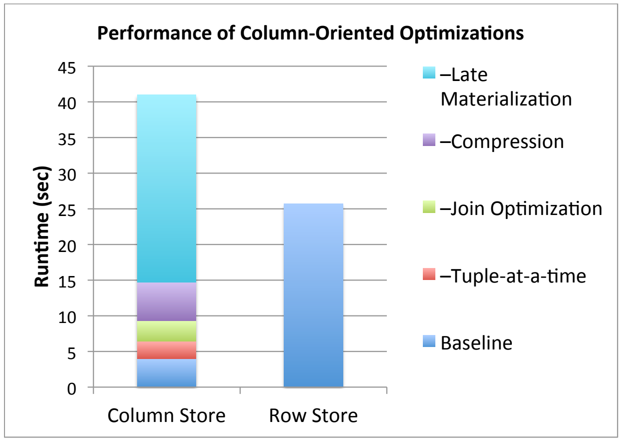


Figure 1.2: Performance of C-Store versus a commercial database system on the SSBM benchmark, with different column-oriented optimizations enabled.

Performance Example. To illustrate the benefit that column-orientation and these optimizations have, we briefly summarize a result from a recent paper [1]. In this paper, we compared the performance of the academic C-Store prototype to a commercial row-oriented ("row-store") system. We studied the effect of various column-oriented optimizations on overall query performance on SSBM [73] (a simplified version of the TPC-H data warehousing benchmark). The average runtime of all queries in the benchmark on a scale 10 database (60 million tuples) is shown in Figure 1.2. The bar on the left shows the performance of C-Store as various optimizations are removed; the "baseline" system with all optimizations takes about 4 seconds to answer all queries, while the completely unoptimized system takes about 40 seconds. The bar on the right shows the performance of the 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 system. One reason that the unoptimized column-store does not do particularly well is that the SSBM benchmark uses relatively narrow tables. Thus, the baseline I/O reduction from column-orientation is reduced. In most real-world data-warehouses, the ratio of columns-read to table-width would be much smaller, so these advantages would be more pronounced.

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.

Monograph Structure. In the rest of this monograph, we show how the architecture innovations listed above contribute to these kinds of dramatic performance gains. In particular, we discuss the architecture of C-Store, MonetDB and VectorWise, describe how they are similar and different, and summarize the key innovations that make them perform well.

In the next chapter, we trace the evolution of vertically partitioned and column-oriented systems in the database literature, and discuss technology trends that have caused column-oriented architectures to become more favorable for analytic workloads. Then, in Chapters 3 and 4, we describe the high level architecture and detailed internals primarily of C-Store, MonetDB and VectorWise but also those of subsequent commercial implementations. Finally, in Chapter 5, we discuss future trends and conclude.

2

History, trends, and performance tradeoffs

While column-oriented storage techniques appeared as early as the 1970s, it was not until the 2000s that column-store research gained recognition and commercial column-stores took off. In this chapter, we trace the history of column-stores, examine technology and application trends that led to the recent resurgence of column-oriented systems, and finally present a summary of recent studies on the fundamental performance tradeoffs between column- and row-stores.

2.1 History

The roots of column-oriented database systems can be traced to the 1970s, when transposed files first appeared [65, 12]. TOD (Time Oriented Database) was a system based on transposed files and designed for medical record management [90]. One of the earliest systems that resembled modern column-stores was Cantor [55, 54]. It featured compression techniques for integers that included zero suppression, delta encoding, RLE (run length encoding), and delta RLE–all these are commonly employed by modern column-stores (we discuss column-oriented compression in later chapters). A dynamic programming algorithm was

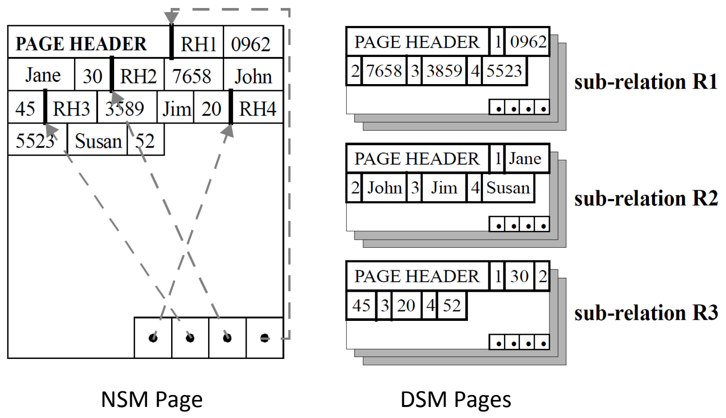


Figure 2.1: Storage models for storing database records inside disk pages: NSM (row-store) and DSM (a predecessor to column-stores). Figure taken from [5].

used to choose compression methods and related parameters.

Research on transposed files was followed by investigations of vertical partitioning as a technique for table attribute clustering. At the time, row-stores were the standard architecture for relational database systems. A typical implementation for storing records inside a page was a slotted-page approach, as shown on the left part of Figure 2.1. This storage model is known as the *N*-ary Storage Model or NSM. In 1985, Copeland and Khoshafian proposed an alternative to NSM, the Decomposition Storage Model or DSM–a predecessor to column-stores [22] (see left part of Figure 2.1). For many, that work marked the first comprehensive comparison of row- and column-stores. For the next 20 years, the terms DSM and NSM were more commonly used instead of row- or column-oriented storage. In the DSM, each column of a table is stored separately and for each attribute value within a column it stores a copy of the corresponding surrogate key (which is similar to a record id or RID), as in Figure 1.1(b). Since surrogate keys are copied in each column, DSM requires more storage space than NSM for base data. In addition to storing each column in the same order as the original table (with a clustered index on surrogate keys), the authors proposed to store a non-clustered index on each column's attribute values, providing a fast way to map any attribute value to the corresponding surrogate key.

An analysis (based on technology available at the time) showed that DSM could speed up certain scans over NSM when only a few columns were projected, at the expense of extra storage space. Since DSM slowed down scans that projected more than a few columns, the authors focused on the advantages of DSM pertaining to its simplicity and flexibility as a storage format. They speculated that physical design decisions would be simpler for DSM-based stores (since there were no index-creation decisions to make) and query execution engines would be easier to build for DSM. The original DSM paper did not examine any compression techniques nor did it evaluate any benefits of column orientation for relational operators other than scans. A follow-up paper focused on leveraging the DSM format to expose inter-operator parallelism [59] while subsequent research on join and projection indices [58] further strengthened the advantages of DSM over NSM.

Although the research efforts around DSM pointed out several advantages of column over row storage, it was not until much later, in the 2000s, that technology and application trends paved the ground for the case of column-stores for data warehousing and analytical tasks.

2.2 Technology 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 [24]. 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 vs. 3msec). The differences in these trends (10,000x vs. 100x vs. 10x) have had a significant impact on the performance of database workloads [24].

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 began to increasingly rely on sequential access patterns, to the point that most DBMS architectures are built around the premise that completely sequential access should be done whenever possible. However, as database sizes increased, scanning through large amounts of data became slower and slower. A bandwidth-saving solution was clearly needed, yet most database vendors did not view DSM as viable replacement to NSM, due to limitations identified in early DSM implementations [22] where DSM was superior to NSM only when queries access very few columns. In order for a column-based (DSM) storage scheme to outperform row-based (NSM) storage, 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, in the 1990s, column-stores were mostly widely used in main-memory systems. 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. Around 1980, the time required to access a value in main memory and execute an instruction were about the same. By the mid 1990s, memory latency had grown to hundreds of CPU cycles. The MonetDB project [16, 46] was the first major 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 [19]. A 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).

PAX (for Partition Attributes Across) adopted a hybrid NSM/DSM approach, where each NSM page was organized as a set of mini columns [5]. It retained the NSM disk I/O pattern, but optimized cache-to-RAM communication (seeking to obtain the cache latency benefits identified in Monet without the disk-storage overheads of DSM with its explicit row IDs). Follow on projects included data morphing [39], a dynamic version of PAX, and Clotho [84] which studied custom page layouts using scatter-gather I/O.

Fractured Mirrors [78] leveraged mirroring for reliability and availability. The idea is to store one copy of the data in NSM format and one separate copy in DSM, thereby achieving the performance benefits of both formats. Since data is often replicated anyway for availability, Fractured Mirrors allows one to get the benefits of both formats for free.

Around the same time (1996), one of the first commercial column-store systems, SybaseIQ [28, 29, 66], emerged, demonstrating the benefits that compressed, column-oriented storage could provide in many kinds of analytical applications. Although it has seen some commercial success over the years, it failed to capture the mindshare of other database vendors or the academic community, possibly due to a combinations of reasons, e.g., because it was too early to the market, hardware advances that later favored column-storage (and triggered database architecture innovations) such as large main memories, SIMD instructions, etc. where not available at the time, and possibly because it lacked some of the architectural innovations that later proved to be crucial for the performance advantages of column-stores, such as (extreme) late materialization, direct operation on compressed data throughout query plans, etc. Sybase IQ did offer some early variations of those features, e.g., compressing columns separately, or performing joins only on compressed data, avoiding stitching of tuples as early as loading data from disk, etc. but still it did not offer an execution engine which was designed from scratch with both columnar storage and columnar execution in mind.

Other than Sybase IQ, additional early signs of exploiting columnar storage in industry appeared in systems such IBM EVI [14] which allowed part of the data to be stored in column format or SAP BW/Trex which offered columnar storage but was a text search engine as opposed to a full blown relational engine.

By the 2000s column-stores saw a great deal of renewed academic and industrial interest. 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 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 pioneering modern column-store systems, C-Store [88] and VectorWise [18]. These systems provided numerous innovations over the state-of-the-art at the time, such as column-specific compression schemes, operating over compressed data, C-store projections, vectorized processing and various 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 (e.g., Vertica, Ingres VectorWise, Paraccel, Infobright, Kickfire, and many others) that were influenced by these systems. This was later followed by announcements and acquisitions by traditional vendors of row-store systems (such as Oracle, Microsoft, IBM, HP, SAP and Teradata) that added column-oriented systems and features to their product lineups.

Some notable examples in industry include IBM BLU [79] which originated from the IBM Blink project [11] and provided innovations mainly in providing an architecture tightly integrated with the ability to process compressed columns, as well as SAP HANA [26] which stores data both in a row-format and in a column-format to combine online analytical processing and online transaction processing in a single system. In addition, Microsoft soon followed with extensions in the architecture of the SQL Server row-store which brought features such as column-oriented storage, vectorized processing and compression [62, 61]; initially columns were used as an auxiliary accelerator structure, i.e., column indexes [63] but subsequent versions provide a more generic architecture which allows also base data to be stored in columnar form [62].

2.3 Fundamental Performance Tradeoffs

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. This was due to various overheads in reconstructing a tuple from multiple columns, accessing multiple locations on disk, and processing extraneous, per-column information. In order for column-stores to become competitive with 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 with row-store scans. This was demonstrated by several studies over the past ten years.

- In Fractured Mirrors [78], 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, amortized the header over-

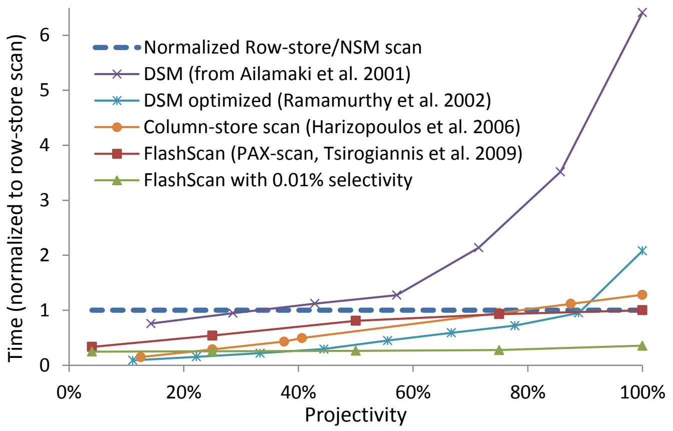


Figure 2.2: Performance of column-oriented scans for various degrees of projectivity (percentage of a full-row tuple that needs to be reconstructed) compared to (normalized) row-store scan, from various studies over the years.

head across multiple column values (Graefe also describes an efficient method for storing columns inside a B-tree [33]), and used chunk-based tuple reconstruction.

- In [40] the authors provide a comparison of column and row scanners using a from scratch implementation, a stand-alone storage manager, as well as 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 was accessed, column-store scans were shown to be only 20-30% slower than row-store scans.

- Finally, in [89] 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 in this case 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 2.2 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 [5] 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; if selectivity is high, then column-stores can minimize the amount of intermediate results they create which otherwise represents a significant overhead.

3

Column-store Architectures

In this chapter, we describe the high level architecture of the three early column-oriented research prototypes: C-Store, MonetDB and VectorWise. These architectures introduced the main design principles that are followed by all modern column-store designs. This chapter highlights the design choices in each system; many core design principles are shared but some are unique in each system. The next chapter discusses those individual features and design principles in more detail as well as it provides query processing and performance examples.

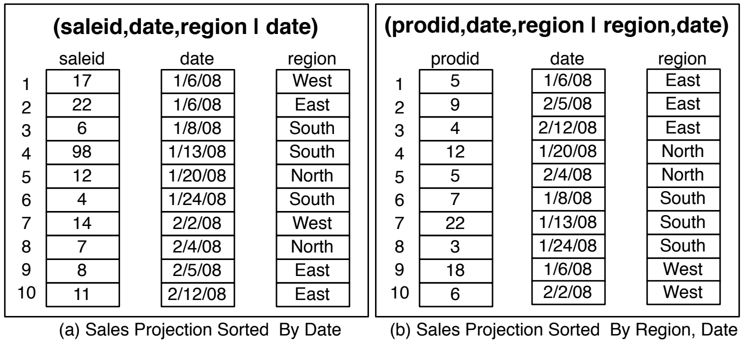
3.1 C-Store

In C-Store, the primary representation of data on disk is as 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. 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 in a columnar form.

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 3.1 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.

Each column in C-Store is compressed and for each column a different compression method may be used. The choice of a compression method for each column depends on a) whether the column is sorted or not, b) on the data type and c) on the number of distinct values in the 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 X products of the same class is represented as a single `(X, product class)` pair, rather than X distinct tuples. More details on compression methods used in column-stores are discussed in Chapter 4.

C-Store does not support secondary indices on tables, but does



(saleid,date,region | date)

	saleid	date	region
1	17	1/6/08	West
2	22	1/6/08	East
3	6	1/8/08	South
4	98	1/13/08	South
5	12	1/20/08	North
6	4	1/24/08	South
7	14	2/2/08	West
8	7	2/4/08	North
9	8	2/5/08	East
10	11	2/12/08	East

(a) Sales Projection Sorted By Date

(prodid,date,region | region,date)

	prodid	date	region
1	5	1/6/08	East
2	9	2/5/08	East
3	4	2/12/08	East
4	12	1/20/08	North
5	5	2/4/08	North
6	7	1/8/08	South
7	22	1/13/08	South
8	3	1/24/08	South
9	18	1/6/08	West
10	6	2/2/08	West

(b) Sales Projection Sorted By Region, Date

Figure 3.1: Two different projections of the Sales table.

support efficient indexing into sorted projections through the use of *sparse indexes.* A sparse index is a small tree-based index that stores the first value contained on each physical page of a column. A typical page in C-Store would be a few megabytes in size. Given a value in a sorted projection, a lookup in this tree returns the first page that contains that value. The page can then be scanned to find the actual value. A similar sparse index is stored on tuple position, allowing C-Store to efficiently find a given tuple offset in a column even when the column is compressed or contains variable-sized attributes.

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 Chapter 4.

Finally, in addition to complete vertical partitioning, C-Store was conceived as a shared-nothing massively parallel distributed database system, although the academic prototype never included these features (the commercial version, Vertica, does). The idea behind the parallel design of C-Store is that projections are horizontally partitioned across multiple nodes using hash- or range-partitioning, and queries are pushed down and executed as much as possible on each node, with partial answers merged to produce a final answer at the output node. Most of C-Store's parallel design was based on the design of early shared nothing parallel systems like Gamma [23], so we do not concentrate on these features here.

3.2 MonetDB and VectorWise

In this section, we first discuss the architecture of MonetDB, while subsequently we focus on VectorWise.

MonetDB. MonetDB is designed from scratch focusing on handling analytical workloads efficiently on modern hardware. MonetDB stores data one column-at-a-time both in memory and on disk and exploits bulk processing and late materialization. It solely relies on memory mapped files avoiding the overhead and complexity of managing a buffer pool. MonetDB differs from traditional RDBMS architecture in many aspects, such as its:

- Execution engine, which uses a column at-a-time-algebra [19],
- Processing algorithms, that minimize CPU cache misses rather than IOs [17],
- Indexing, which is not a DBA task but happens as a by-product of query execution, i.e., database cracking [50],
- Query optimization, which is done at run-time, during query incremental execution [4] 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 [19].

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 way, 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 fixed-width and dense arrays, i.e., columns. Such code is well-supported by compiler technology to extract maximum performance from CPUs through techniques such 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 fewer instructions - 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 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. Both base data and intermediate results are 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. BATs are essentially in-memory (or memory mapped) arrays. The BAT algebra operators consume and produce BATs. For example a select operator consumes a single input BAT, applies a filter to every value in the underlying array and produces a new BAT with the qualifying tuples.

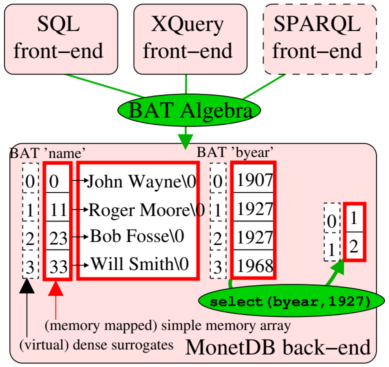


Figure 3.2: MonetDB: BAT Algebra Execution.

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 3.2 shows query plans in BAT Algebra being generated by various front-ends to be executed in the MonetDB back-end.

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 in traditional systems, 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.

More recent research has shown that further advantages can be achieved by going the extreme route and compiling code on-the-fly (i.e., during query processing). The rational is that compiled code optimally fits query patterns and data layouts for a specific query, increasing the performance of scan intensive workloads, e.g., [43, 70], with operators that perform only the required actions for the particular scan at hand, minimizing indirections, function calls and thus cache misses.

To handle updates, MonetDB uses a collection of pending updates columns for each base column in a database. Every update action affects initially only the pending updates columns, i.e., every update is practically translated to an append action on the pending update columns. Every query on-the-fly merges updates by reading data both from the base columns and from the pending update columns. For example, when applying a filter on column X, there will be one select operator applied directly on column X and another select operator applied on the pending updates columns of X, while subsequently qualifying pending inserts are merged or pending deletes are removed from the corresponding intermediate result. Periodically, pending columns are merged with their base columns.

VectorWise. While MonetDB pioneered many core column-store design principles, still it misses some of the signature design points later introduced by C-store and VectorWise. 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 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. For example, if a select operator consumes its complete input column in one go, then it needs to materialize a result which represents all qualifying tuples, resulting in a significant overhead especially as we scale to bigger data inputs. Together, these aspects make MonetDB vulnerable to swapping when its working set starts exceeding RAM.

These problems were addressed by a new system, developed in the same research group at CWI, called VectorWise [96]. VectorWise marked a new from scratch development to address the shortcomings of MonetDB and to provide an architecture tailored for modern hardware. The main innovation in VectorWise is its vectorized execution model which strikes a balance between full materialization of intermediate results in MonetDB and the high functional overhead of tuple-at-a-time iterators in traditional systems. Essentially, VectorWise processes one block/vector of a column at a time as opposed to one column-at-a-time or one tuple-at-a-time.

VectorWise does perform explicit I/O, in an advanced way, adaptively finding synergy in the I/O needs of concurrent queries through its Active Buffer Manager (ABM) and Cooperative Scans [98]. VectorWise also provides a novel way of handling updates (Positional Delta Trees [41]), and new high-speed compression algorithms [100]. We discuss those features in detail in the next chapter.

3.3 Other Implementations

Subsequent designs from industry share the fundamental principles of VectorWise and C-Store (which we discuss in more detail in the next chapter). There are two main architectures that have been used by industry in adopting a column-store or a column-store-like design.

Columnar Storage Only. The first approach involves storing data one column-at-a-time on disk but relies on a standard row-store execution engine to process queries. That is, every query may access only the columns referenced, thus seeing some savings in terms of I/O but once all relevant data arrives in memory, it is immediately stitched into a tuple N-ary format and is fed to a classic row-store engine. In this way, such a design is relatively easy to adopt as it only requires a mapping of columns to tuples when loading from disk, but it also means that such designs cannot exploit the run-time benefits of operating on one column-at-a-time which allows for better exploitation of the whole memory hierarchy. Some characteristic examples of such implementations at the moment of writing this monograph include Teradata/Asterdata and EMC/Greenplum. An advantage of such a design is that it allows for a smoother transition phase to a completely new architecture as well as that it allows for databases where some of the data may be stored in the original row format while other parts of the data may be stored in a column format and still both kinds of data can be processed by the same execution engine.

Native Column-store Designs. The second and more advanced direction has to do with vendors that adopted the full column-store model, providing both a columnar storage layer and an execution engine which is tailored for operating on one column-at-a-time with late tuple reconstruction. Then, this new engine is integrated with a traditional row-oriented execution engine.

IBM BLU/BLINK. A prime example of the second paradigm is IBM BLU [79] which originated from the IBM BLINK project [11, 52]. Essentially, IBM BLU sits on the side of the standard row-store DB2 engine and becomes responsible for part of the data. The optimizer then knows which queries to push to the BLU engine and which queries to push to the standard engine. In this way, queries which may benefit from column-store execution may do so and vice versa and in fact queries may scan data from both row-oriented and column-oriented tables.

Other than exploiting standard column-store design practices such as late materialization, IBM BLINK/BLU also introduced novel techniques especially in the area of compression. Frequency partitioning [81] is used in order to maximize the storage savings gained by compression but at the same time remain within the general design column-store principles. The general idea is that columns are reorganized with the intention to minimize the variability in each data page. That is, each page is compressed separately with dictionary compression and by minimizing the possible values within a page (by reorganizing the column), IBM BLU reduces the amount of different codes needed to represent the data. With fewer codes (compared to having a single dictionary for the whole column) IBM BLU may use less bits to write these codes which in turn reduces the storage space needed to store the referenced data.

Frequency partitioning means that, contrary to other systems, IBM BLU allows for variable width columns. Each page has its own dictionary and code length; within each page all values/codes are fixed-width but different pages of the same column may use a different value width in order to maximize the amount of data that fits in the page. Thus, similarly to other column-stores, IBM BLU can exploit operator designs which rely on tight for-loops and are cache and CPU friendly; it only needs to adjust as it moves from one page to the next. This leads to a slightly more complex page design; it is not purely based on array storage but now needs to store information regarding the dictionaries and other metadata unique to each page such as the mapping of the local tuples to the global order. Given that frequency partitioning reorganizes data and given that this happens typically at the level of individual columns, this means that the various columns may be stored in different order and thus there needs to be a way to be able to link tuples together across all column of the same table. We discuss frequency partitioning in more detail in the next chapter along with other core column-store design principles.

Microsoft SQL Server Column Indexes. Another notable paradigm of a major vendor adopting a columnar architecture is SQL Server from Microsoft [62]. SQL Server provides native support for columnar storage and column-oriented execution, adopting many of the critical design features that are common in column-stores, such as vectorized processing and heavily exploiting compression. These features have been integrated with the traditional row-store design of SQL Server, providing the flexibility of choosing the appropriate physical design depending on the workload. Columns can be used either as “column indexes”, i.e., auxiliary data, enhancing scans over specific attributes for part of the workload or they can also be used as the primary storage choice for purely scan intensive scenarios.

4

Column-store internals and advanced techniques

Having seen the basics of column-stores in the previous chapters, in this chapter we discuss in more detail specific design features which go beyond simply storing data one column-at-a-time and which differentiate column-stores from traditional architectures. In particular, we focus on vectorized processing, late materialization, compression and database cracking.

4.1 Vectorized Processing

Database text-books generally contrast two strategies for the query execution layer, namely the “Volcano-style” iterator model [32], which we also refer to as tuple-at-a-time pipelining, versus full materialization. In tuple-at-a-time pipelining, one tuple-at-a-time is pushed through the query plan tree. The `next()` method of each 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. 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, each query operator works in isolation, fully 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.

To illustrate the differences between the above two models, assume the following query: $select\ avg(A)\ from\ R\ where\ A < 100$. With tuple-at-a-time pipelining the select operator will start pushing qualifying tuples to the aggregation operator one tuple-at-a-time. With full materialization, though, the select operator will first completely scan column A, create an intermediate result that contains all qualifying tuples which is then passed as input to the aggregation operator. Both the select and the aggregation operator may be implemented with very efficient tight for loops but on the other hand a big intermediate result needs to be materialized which for non-selective queries or for big data, exceeding memory size, becomes an issue.

We now turn our attention to a third alternative called "vectorized execution" pioneered in VectorWise, which strikes a balance between full materialization and tuple pipelining. This model separates query progress control logic from data processing logic. Regarding control flow, the operators in vectorized processing are similar to those in tuple pipelining, with the sole distinction that the `next()` method of each operator returns a vector of N tuples as opposed to only a single tuple. 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 combines pipelining (avoidance of materialization of large intermediates) with the array-loops code patterns that make MonetDB fast.

The typical size for the vectors used in vectorized processing is such that each vector comfortably fits in L1 cache ($N = 1000$ is typical in VectorWise) as this minimizes reads and writes throughout the memory hierarchy. Given that modern column-stores work typically on one vector of one column at a time (see also discussion on late materialization in Section 4.4), this means that only one vector plus possible output vectors and auxiliary data structures have to fit in L1. For example, a query with multiple predicates one multiple columns will typically apply the predicates independently on each column and thus only one vector of a single column at a time has to fit in the cache (a detailed query processing example with late materialization is shown in Section 4.4).

There are numerous advantages with vectorized processing. We summarize the main ones below:

- **Reduced interpretation overhead.** The amount of function calls performed by the query interpreter goes down by a factor equal to the vector size compared to the tuple-at-a-time model. On computationally intensive queries, e.g., TPC-H Q1, this can improve performance by two orders of magnitude.

- **Better cache locality.** VectorWise 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 (as 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 strikes a balance between tuple-at-a-time processing and full materialization; control now stays for as many iterations as the vector size 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 amenable 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.

- **Parallel memory access.** 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. It was shown in [96] to be possible to vectorize memory lookups in all major relational database operators, e.g., sorting, hash-table lookup, as well as hash-table probing. Such lookups often incur cache-misses, in which case code that through out-of-order speculation generates multiple parallel misses often performs four times faster than non-vectorized memory lookups.

- **Profiling.** Since vectorized implementations of relational operators perform all expression evaluation work in a vectorized fashion, i.e., array-at-a-time for hundreds or even thousands of tuples in one go, the overhead of keeping performance profiling measurements for each individual vectorized operation is low (as bookkeeping cost is amortized over hundreds or thousands of tuples). This allows vectorized engines to provide highly detailed performance insight into where CPU cycles are spent.

- **Adaptive execution.** Building on the latter point, performance profile information on vectorized primitives can also be exploited at run-time, *during* the execution of a query. For example, Vectorwise decides adaptively in case of arithmetic operations on vectors where only a subset of the values in the arrays is selected by some predicate, whether to compute the result only for the selected tuples iteratively, or for all tuples in the array. The latter strategy, while performing extra work, leads to a tight loop without if-then-else, where SIMD instructions can be used, making it overall faster as long as the percentage of selected tuples is relatively high. The Micro Adaptivity mechanism of VectorWise [77] generalizes the concept of using run-time statistics to optimize query processing. An adaptive "Multi Armed Bandid" algorithm has the task of choosing at run-time the best "flavor" (alternative implementation) for a vectorized function. Periodically, during the query – during which a vectorized primitive may be called millions of times – it tests all alternative implementations and subsequently uses the best performing implementation most of the time. This approach resists differences in compilers and compiler flags (by linking in the same function multiple times, compiled differently, available as different flavors) as well as hardware changes (eliminating the need of detailed cost modeling, cost model maintenance and calibration) and can also react to changes in the data distribution during the query.

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 in its current form, was developed and analyzed in the column storage context of VectorWise, the principle can also be applied to row stores as it is not tied to the storage manager. In fact, past research in row-stores has been experimenting with such concepts as early as in 1994 [85] where the first efforts appeared towards minimizing cache misses and instruction misses. Subsequent efforts proposed even more generic solutions either for block based query processing [74] or even by adding so called buffer operators [95] within traditional tuple-at-a-time row-store query plans which simulate the effect of blocking by not allowing tuples to be propagated through a plan until buffers are full.

In [101] 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 opens 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.

4.2 Compression

Intuitively, data stored in columns is more compressible than data stored in rows. Compression algorithms perform better on data with low information entropy (i.e., with high data value locality), and values from the same column tend to have more value locality than values from different columns.

Compressing one column-at-a-time. For example, assume a database table containing information about customers (name, phone number, e-mail address, snail-mail address, etc.). Storing all data together in the form of rows, means that each data page contains information on names, phone numbers, addresses, etc. and we have to compress all this information together. On the other hand, storing data in columns allows all of the names to be stored together, all of the phone numbers together, etc. Certainly phone numbers are more similar to each other than to other fields like e-mail addresses or names. This has two positive side-effects that strengthen the use of compression in column-stores; first, compression algorithms may be able to compress more data with the same common patterns as more data of the same type fit in a single page when storing data of just one attribute, and second, more similar data implies that in general the data structures, codes, etc. used for compression will be smaller and thus this leads to better compression. Furthermore, if the data is sorted by one of the columns, which is common with column-store projections, that column will be super-compressible (for example, runs of the same value can be run-length encoded).

Exploiting extra CPU cycles. Usually, the bottom line goal of a database system is performance, i.e., processing one or more queries as fast as possible, not compression ratio. Disk space is cheap, and is getting cheaper rapidly. However, compression does improve performance (in addition to reducing disk space); if data is compressed, then less time is spent in I/O during query processing as less data is read from disk into memory (and from memory to CPU). Another important motivation here is that as CPUs are getting much faster compared to memory bandwidth, the cost of accessing data costs more in terms of CPU cycles than it did in the past. Intuitively, this means that now we have more CPU cycles to spare in decompressing compressed data fast which is preferable to transferring uncompressed and thus bigger data at slow speeds (in terms of waisted CPU cycles) through the memory hierarchy.

Fixed-width arrays and SIMD. 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. With SIMD instructions we can decompress or process multiple compressed values with one instruction as long as they are packed into fixed-width and dense arrays (that nicely fit into SIMD registers of modern processors), maximizing parallelism. Since column-stores exploit fixed-width dense arrays anyway, they can exploit SIMD execution even with uncompressed data. However, with compression, more (compressed) data values will fit in a SIMD register compared to when we do not compress the columns and as a result we are able to process even more data with a single instruction at a time. For example, a modern processor has SIMD registers that typically fit 4 4-byte integers at a time and thus a column-store without compression may process 4 values at a time in this case. If data is compressed by a factor of 2 though, then we will be able to fit 8 compressed integers in the SIMD register and we can process 8 values at a time, increasing parallelism.

Overall, compression has been shown to heavily improve performance in modern column-stores and it is now an integral part of all column-stores in industry. In addition, it is interesting to note that the extra storage space which is gained on disk due to compression can be used towards materializing auxiliary data structures/copies, i.e., such as the projections proposed in C-Store. In turn, this improves performance even more because now queries may enjoy better access patters. There have been several research studies on compression in column-stores [2, 100, 43, 15]. Most of this work was pioneered in C-Store and VectorWise but significant advances were made in industry as well. In particular, the IBM BLINK project [11] proposed the so called frequency partitioning scheme which provides a much more tight integration of compression with a column-store architecture.

Frequency partitioning. The main motivation of frequency partitioning is to increase the compression ratio while still providing an architecture that relies on fixed-width arrays and can exploit vectorization. This requires a more tight design of compression with the system architecture. With frequency partitioning a column is reorganized such as in each page of a column, we have as low information entropy as possible. To do this IBM BLINK reorganizes each column based on the frequency of values that appear in the column, i.e., frequent values are stored together in the same page(s). This allows the system to use compact per page dictionaries for dictionary compression, requiring fewer codes which can in turn be stored with fewer bits (compared to having a single dictionary for the whole column). For example, if we only have to distinguish between two values in a single page of a column, then we only need a single bit for the dictionary codes in this page. Within each page, all values/codes are fixed-width which allows for operators with CPU and cache friendly access patterns as in a typical column-store architecture, while the system performs vectorized processing at the granularity of one page-at-a-time.

Compression algorithms. There have been several research studies that evaluate the performance of different compression algorithms for use with a column-store [2, 100, 43, 42, 15]. 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).

There are numerous possible compression schemes that can be applied, i.e., run-length encoding, bit-vector encoding, dictionary compression and patching. We describe those techniques in the following sections.

4.2.1 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). For example, with ideas such as C-Store projections where each column may be stored in multiple projections/orders, many columns end-up being sorted (or secondarily sorted) and thus there are many opportunities for RLE-type encoding.

Given that RLE replaces arbitrary blocks of values with a single triple at a time, it results in variable width and variable length columns. This implies that we cannot use the kind of operators described previously for fixed-width columns as well as that tuple reconstruction becomes a little bit more complicated. This is a tradeoff one has to balance against the storage gains and thus the I/O and performance improvements that RLE brings on a particular column based on the underlying data distribution.

4.2.2 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 bits 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 bitstring if the i^{th} value in the column is equal to the domain element that the bitstring 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 [71]), there has been much work on further compressing these bit-maps and the implications of this further compression on query performance [68, 8, 7, 53, 91, 93, 92].

4.2.3 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 [76, 11]. 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 with a global dictionary.

One benefit of dictionary compression is that it can result in fixed width columns if the system chooses all codes to be of the same width. This requires sacrificing a little bit in terms of the ultimate storage gains, but allows for CPU efficient access patterns. In fact, it is interesting to note that even though plain MonetDB does not exploit compression throughout its architecture (this was introduced in VectorWise) it still uses dictionary compression for (variable width) string columns in order to transform them into fixed-width columns (of codes).

One practical point of consideration is how to dictionary compress efficiently, which depends on fast hashing. One particularly fast technique is cuckoo hashing [97].

4.2.4 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 [31]. 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.

4.2.5 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 [100], rather than storing escape values in the codes, uses these to maintain a linked list. Decompression first compresses all codes regardless 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 optimizations opportunities provided by block-wise processing.

4.3 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. Another example, is when dictionary compression uses order preserving encoding; that is, the code representing a column value is guaranteed to be smaller than all codes representing bigger values and bigger than all codes representing smaller values. In this way, comparison actions during filtering, e.g., within a select operator, may be performed directly on the codes without decompressing the data; we only need to encode the filter bounds/pivots instead.

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 the 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 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 [2]. 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.

4.4 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.

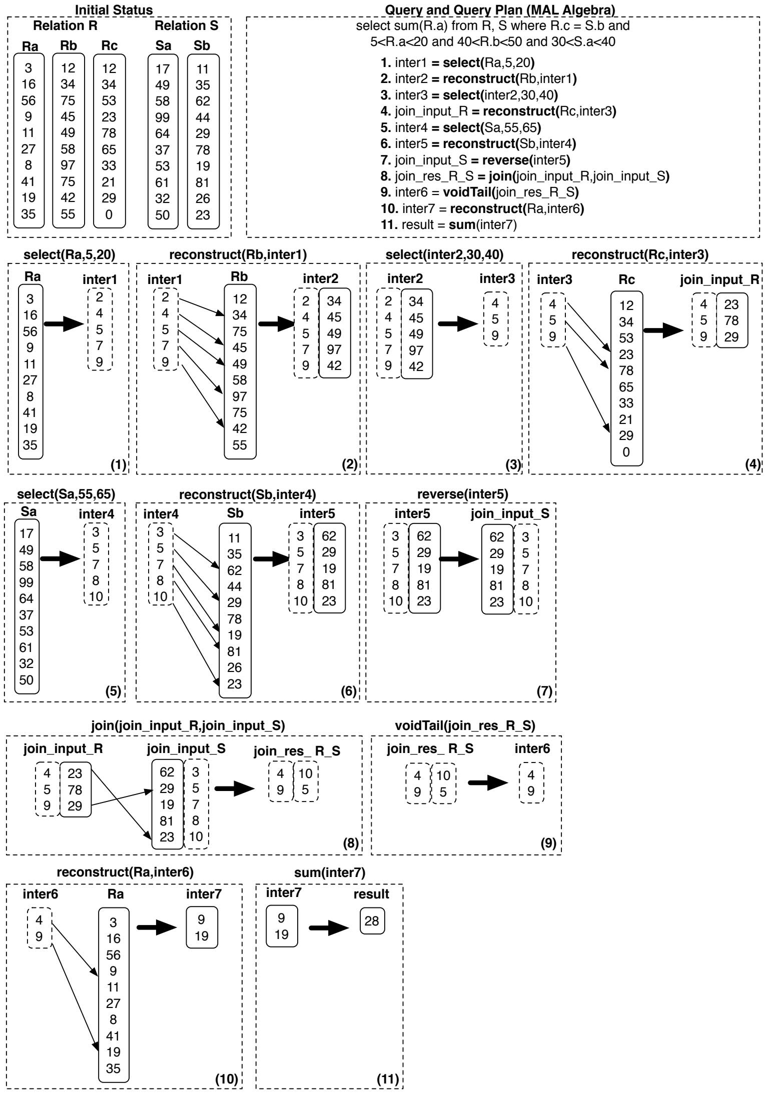
Naive column-stores [38, 40] store data on disk (or in memory) column-by-column, read in (to CPU from disk or memory) only those columns relevant for a particular query, construct tuples from their component attributes, and execute normal row-store operators on these rows to process (e.g., select, aggregate, 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 VectorWise, C-Store, Vertica, and to a lesser extent, SybaseIQ, 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 offsets within a column) of values that passed the predicates are 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.

Example. Figure 4.1 shows a simple example of a late materialization query plan and execution in a modern column-store. Here, we assume that intermediate results are represented with position lists and, in order to focus solely on the late materialization issues, and for ease of presentation, no compression is used and we show the example using bulk processing. The query in Figure 4.1 is a select-project-join query; it essentially filters three columns of two separate tables (R,S) and then joins these two tables on two columns, while subsequently it performs a sum aggregation on one of the tables (R). Figure 4.1 shows graphically the various steps performed to answer this query, as well as it shows the query plan in MAL algebra and how each MAL operator behaves.

Late materialization means that we always operate on individual columns, i.e., in Figure 4.1 the select operators filter each column independently, maximizing utilization of memory bandwidth as only the relevant data is read for each operator. In this way, after having filtered column $R.a$ in Step 1 of Figure 4.1, a position list contains the positions of the qualifying tuples. Recall that positions are used as row IDs. All intermediate results which are position lists in Figure 4.1 are marked with a dashed line. Then, in Step 2, we reconstruct column $R.b$ which is needed for the next filtering action. Since the positions in the position list ($inter1$) are ordered (as we sequentially scanned $R.a$ in Step 1), we can project the qualifying $R.b$ values in a cache-friendly skip sequential access pattern. In Step 3, we scan intermediate result $inter2$ to apply the second filter predicate (on $R.b$). In the same way as before, this results in a new intermediate array that contains the qualifying positions which we then use in Step 4 to fetch the qualifying values from column $R.c$ (which we need for the join).

Subsequently, we filter column $S.a$ in the same way as we did for table R and we fetch the qualifying values from the other join input column ($S.b$). In Step 7 we reverse this intermediate result in order to feed it in the proper order to the join operator. The join operator in Step 8 operates on the individual join input columns, and it produces two position lists which may be used for projecting any attributes needed in



```
select sum(R.a) from R, S where R.c = S.b and
5<R.a<20 and 40<R.b<50 and 30<S.a<40

1. inter1 = select(Ra,5,20)
2. inter2 = reconstruct(Rb,inter1)
3. inter3 = select(inter2,30,40)
4. join_input_R = reconstruct(Rc,inter3)
5. inter4 = select(Sa,55,65)
6. inter5 = reconstruct(Sb,inter4)
7. join_input_S = reverse(inter5)
8. join_res_R_S = join(join_input_R,join_input_S)
9. inter6 = voidTail(join_res_R_S)
10. inter7 = reconstruct(Ra,inter6)
11. result = sum(inter7)
```

Figure 4.1: An example of a select-project-join query with late materialization.

the select clause by either table. In this case, we need only the position list for table R so we "void" the tail of the join result and use this position list in order to fetch the qualifying values from column $R.a$ which we need for the aggregation. Finally, in Step 11, we perform the aggregation (again on one column-at-a-time) enjoying CPU and cache friendly access patterns, reading only the relevant data.

Every time we fetch the values of a column given a position list (which is the result of a previous operator) we say that we perform a tuple reconstruction action. Such actions have to be performed multiple times within a query plan, i.e., at least $N - 1$ times for each table, where N is the number of attributes of a given table referenced in a query. Tuple alignment across columns and enforcement of sequential access patterns reduces the costs of tuple reconstruction actions. In Figure 4.1, we demonstrated an architecture where intermediate results are materialized in the form of row-id lists (positions). However, as we discussed earlier, many more alternatives are possible (typically depending on selectivity) such as using bit vectors or filtering columns independently and merging results as in [80].

Another interesting observation is that with C-Store projections, tuple reconstruction becomes a more lightweight action. Given that each projection is sorted by one leading column, then a query which selects a range on this column immediately restricts its actions for tuple reconstruction to the range defined by the first selection. Since the projection is sorted on this attribute, this is a contiguous range on the projection, which in turn means that any tuple reconstruction actions take place only in a restricted horizontal partition as opposed to the whole projection; this inherently provides better access patterns as there will be less cache misses. Sideways database cracking [50] (to be discussed later on) provides the same effect but in a self-organizing way, i.e., partially sorting columns as the workload evolves, adapting to workload patterns and avoiding creating whole projections a priori.

Advantages of late materialization. The advantages of late materialization are four-fold [3]. 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 [5]. 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.

Multi-column blocks. There are several directions to further improve performance of tuple reconstruction or even to eliminate it altogether in some cases. The high level idea is to store data in groups of

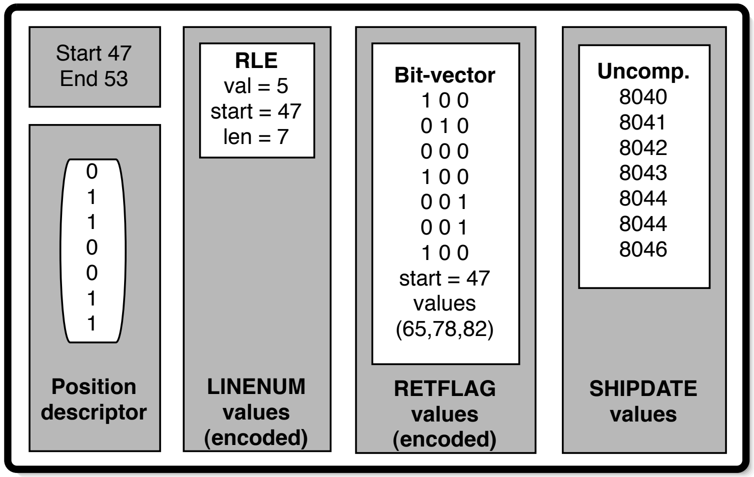


Figure 4.2: An example multi-column block containing values for the SHIPDATE, RETFLAG, and LINENUM columns. The block spans positions 47 to 53; within this range, positions 48, 49, 52, and 53 are active (i.e., they have passed all selection predicates).

columns, as opposed to one column-at-a-time, in what is called multi-column blocks [3], or vector blocks [18] or even column-groups [11].

A multi-column block or vector block contains a cache-resident, horizontal partition of some subset of attributes from a particular relation, stored in their original compressed representation. Figure 4.2 shows an example. One way to think about multi-column blocks is that it is a storage format similar to PAX [5] with the difference that not all attributes of a relational table have to be in the same page. 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 4.2) 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.

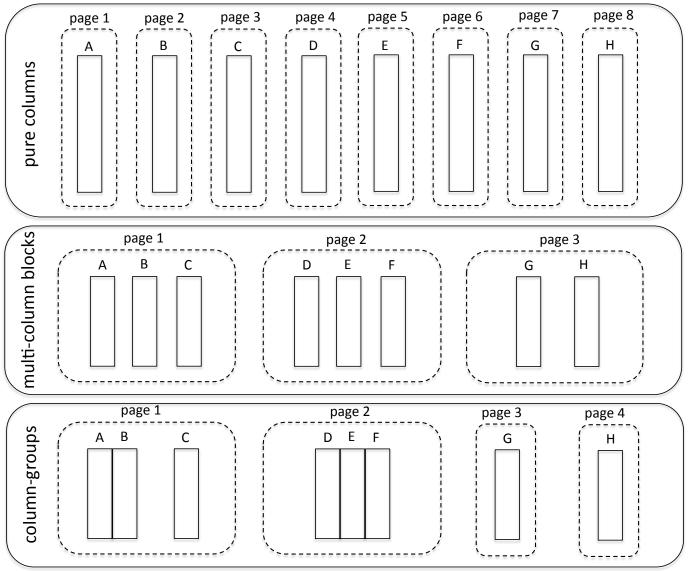


Figure 4.3: An example of a alternative storage formats in modern column-stores.

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 [3] 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.

Column-groups, used in IBM Blink [11], propose an even more flexible layout where some of the columns stored in the same page of a multi-column block may also be stored in a row format (thus forming a column-group). By row-store format here we do not mean traditional row-store formats as in slotted pages; instead data is still in the form of fixed-width dense arrays, but a subset of the columns in a page may be “glued” together, forming a matrix. This can be beneficial for operators that need to work over all these columns, avoiding completely the need for intermediate results and tuple reconstruction (as long as no other attributes are required by a query).

Similar directions to column-groups and multi-column blocks appeared in the row-store context with ideas such as multi-resolutions blocks [94]. The idea was that each row-store page may contain only few of the table’s attributes, which helps to avoid loading unnecessary data attributes from disk for queries that do not want to access all attributes. Internally, the pages were still organized in a slotted format and processing was done in a standard row-store engine.

An example of the various alternative storage formats in modern column-stores is shown in Figure 4.3. Each page, may hold one column-at-a-time, or multiple columns, internally organized in a columnar format or even glued together in fixed-width rows.

Column-groups and variations are supported by most modern column-stores including Vertica, VectorWise, and IBM BLU. One drawback of multi-column blocks is that one needs to make such decisions a priori, i.e., to decide which columns will be grouped together at loading time; this requires workload knowledge and rather stable workload patterns. Research work on VectorWise shows that it is even beneficial to construct such column-groups on-the-fly during query processing when the expected benefit in terms of access patterns outweighs the transformation costs [101], while vision systems expect features such as continuous adaptation of storage formats, i.e., the proper column-groups, based on query patterns [45].

4.5 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. In the case of hash joins (which is the typical join algorithm used) this results in much more compact hash tables which in turn results in much better access patterns during probing; a smaller hash table leads to less cache misses. 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:

$$\begin{array}{|c|}\hline 42\\ \hline 36\\ \hline 42\\ \hline 44\\ \hline 38\\ \hline \end{array} \bowtie \begin{array}{|c|}\hline 38\\ \hline 42\\ \hline 46\\ \hline 36\\ \hline \end{array} = \begin{array}{|c|}\hline 1\\ \hline 2\\ \hline 3\\ \hline 5\\ \hline \end{array} \quad \begin{array}{|c|}\hline 2\\ \hline 4\\ \hline 2\\ \hline 1\\ \hline \end{array}$$

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:

```
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” [64, 89]. 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
2	4
3	2
5	1

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:

2	1
4	2
2	3
1	4

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	4
2	1
2	3
4	2

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 [17] 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 multi-column 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.

Finally, since the rebirth of column-stores in the early 2000s, the work on MonetDB and C-store joins triggered a plethora of research work towards efficient main-memory joins, e.g., [9, 10, 6]. What all these efforts have in common is that they follow the high level practices first adopted in column-store operators such as a focus on main-memory performance, being sensitive to hardware properties and trends, being cache conscious, exploiting SIMD instructions, avoiding random access and pointer chasing, etc.

4.6 Group-by, Aggregation and Arithmetic Operations

Regarding the possible relational operators, so far we discussed selections and joins, in addition to tuple reconstruction. In this section, we talk about other relational operators in column-stores such as group-by, aggregation and arithmetic operators. Overall, these operators take advantage of late materialization and vectorization as well as a layout format which is based on fixed-width dense arrays, and thus being able to work on only the relevant data at a time, exploiting SIMD instructions, and CPU- and cache- friendly patterns.

Group-by. Group-by is typically a hash-table based operation in modern column-stores and thus it exploits similar properties as discussed in the previous section. In particular, we may create a compact hash table, i.e., where only the grouped attribute can be used, leading in better access patterns when probing.

Aggregations. Aggregation operations make heavy use of the columnar layout. In particular, they can work on only the relevant column with tight for-loops. For example, assume `sum(), min(), max(), avg()` operators; such an operator only needs to scan the relevant column (or intermediate result which is also in a columnar form), maximizing the utilization of memory bandwidth. An example is shown in Figure 4.1 in Step 11, where we see that the sum operator may access only the relevant data in a columnar form.

Arithmetic operations. Other operators that may be used in the select clause in an SQL query, i.e., math operators (such as `+,-*,/`) also exploit the columnar layout to perform those actions efficiently. However, in these cases, because such operators typically need to operate on groups of columns, e.g., `select A+B+C From R ...`, they typically have to materialize intermediate results for each action. For example, in our previous example, a `inter=add(A,B)` operator will work over columns A and B creating an intermediate result column which will then be fed to another `res=add(C,inter)` operator in order to perform the addition with column C and to produce the final result. Vectorization helps in minimizing the memory footprint of intermediate results at any given time, but it has been shown that it may also be beneficial to on-the-fly transform intermediate results into column-groups in order to work with (vectors of) multiple columns [101], avoiding materialization of intermediate results completely. In our example above, we can create a column-group of the qualifying tuples from all columns (A, B, C) and perform the sum operation in one go.

4.7 Inserts/updates/deletes

Inherently, column-stores are more sensitive to updates compared to row-stores. By storing each column separately in a separate file, this means that each record/tuple of a relational table is stored in more than one files, i.e., in as many files as the number of attributes in the table. In this way, in order to perform even a single update action on a single row, we need multiple I/O actions (as many as the attributes in the table) in order to update all files. In contrast, a row-store can perform a single update with a single I/O. The use of column-groups can reduce the cost of updates but still we need to access multiple files.

Furthermore, column-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. 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 de-compressed, 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 and MonetDB, 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 (inserts, deletes, and updates) in an in-memory structure. MonetDB uses plain columns, i.e., for every base column in the schema there are two auxiliary columns to store pending inserts and pending deletes; an update is a delete followed by an insert. C-Store proposed that the write optimized store could also use a row-format which speeds up updates even more as only one I/O is needed to write a single new row (but merging of updates in the column format becomes potentially more expensive). 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. However, there are several optimizations that can be applied. 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. For example, a select operator is applied independently on all three columns (base, inserts, deletes) and only qualifying tuples are merged and pushed further in the query plan. 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) 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 to keep 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, by, for example, 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 [41] to be exactly those required to implement optimistic concurrency control.

Finally, a recent research trend is towards supporting both full OLTP and OLAP functionality in a single system, by adopting many of the principles pioneered in column-stores for fast OLAP processing. System Hyper [57, 56] is the most representative example in this area and its main design feature is that it relies on hardware-assisted page shadowing to avoid locking of pages during updates. In addition, SAP HANA [26] stores data in both column and row formats to enable both kinds of functionalities.

4.8 Indexing, Adaptive Indexing and Database Cracking

In this section, we discuss indexing and adaptive indexing approaches in column-stores. Even though column-stores allow for very efficient scans which typically significantly outperform traditional row-store scans, still there is plenty of performance to gain by properly exploiting indexing. Performing a scan in a column-store boils down to simply traversing an array with a tight for-loop. Although this can be very efficient and CPU friendly, working with column-store indexes can be one or more orders of magnitude faster [44]. Regarding the shape of a column-store index, it was shown that it is more beneficial to work over fully sorted columns as opposed to maintaining an in-memory tree structure such as an AVL-tree on top of a column [44]. Tree structures bring random access when traversing the tree, while on the other hand if we fully replicate and sort a base column we can exploit efficient binary search actions during a range select.

Indexing. C-Store proposed the concept of projections, i.e., to replicate each table multiple times and each replica may be ordered by a different attribute. In addition, each replica does not necessarily have to contain all of the table's attributes. A query may use a single covering projection which is ordered (ideally) by the attribute which participates in the query's most selective predicate, thus minimizing the effort needed to search as well as minimizing the effort needed for tuple reconstruction. Given that columns compress very well, materializing these extra projections does not bring a significant storage overhead when compared to a traditional row-store system. Of course, the amount and the kinds of projections needed depends on the workloads and having extra projections brings an overhead for updates (very much as it is with the case of indexes in a traditional system).

Another form of indexing which is typically used in column-stores are zonemaps, i.e., to store light-weight metadata on a per page basis, e.g., min/max. For example, Netezza uses this kind of indexing to speed up scans, i.e., by eliminating pages which are known not to contain qualifying tuples. Other attractive ideas include the use of cache conscious bitmap indexing [86] which creates a bitmap for each zone as opposed to having simply min/max information.

Database Cracking and Adaptive Indexing. Indexing, in all of its forms, requires idle time for set-up effort and workload knowledge; however, these are becoming more and more scarce resources. In the rest of this section, we discuss the early efforts on database cracking [44] in column-stores. Database cracking pioneered the concept of adaptive indexing in modern database systems in the context of the MonetDB system and introduced a column-store architecture tailored for adaptive indexing [48, 49, 50, 51, 37, 35, 36, 83]. We discuss both the basics of these efforts and also the reasons why it is that such efforts flourished within the column-store context by exploiting key features of column-store architectures.

One of the fundamental issues with traditional, i.e., non-adaptive, indexing approaches is that we need to make fixed up-front decisions regarding which indexes we are going to create. Creating every possible index is not feasible because of space and time restrictions; there is not enough space to store all possible indexes but more crucially there is typically not enough idle time to create all those indexes. In this way, we need to make a decision on how to tune a database system, i.e., choose a subset of the possible indexes to create. However, making such choices requires workload knowledge; we need to know how we are going to use the database system, the kinds of queries we are going to ask, which data is more important for the users, etc. As we enter more and more into the big data era, more and more application scenarios exhibit a non-predictable behavior (ad-hoc), meaning there is no workload knowledge to allow for index selection. In addition, more and more applications require as fast as possible to achieve good performance for new data; in other words there is no time to spend in analyzing the expected workload, tuning the system and creating indexes.

Such dynamic and online scenarios are the main motivation for adaptive indexing. The main idea is that the system autonomously creates only the indexes it needs. Indexes are created (a) adaptively, i.e., only when needed, (b) partially, i.e., only the pieces of an index needed are created and (c) continuously, i.e., the system continuously adapts. With database cracking, a database system can be used immediately when the data is available; the more the system is used, the more the performance approaches the optimal performance that would be achieved if there was enough idle time and workload knowledge to fully prepare all indexes needed for the current workload.

The main innovation is that the physical data store is continuously changing with each incoming query q, using *q as a hint on how data should be stored.*

Assume a query requests $A<10$. In response, a cracking DBMS

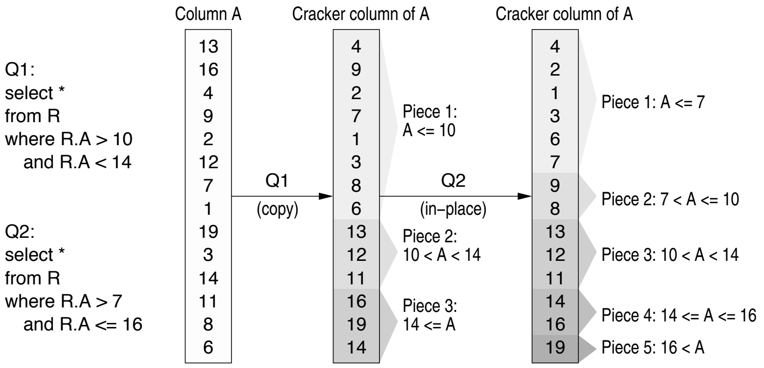


Figure 4.4: Cracking a column [48].

clusters all tuples of A with $A<10$ at the beginning of the respective column C, while pushing all tuples with $A\geq 10$ to the end. A subsequent query requesting $A\geq v_1$, where $v_1\geq 10$, has to search and crack only the last part of C where values $A\geq 10$ reside. Likewise, a query that requests $A<v_2$, where $v_2\leq 10$, searches and cracks only the first part of C. All crack actions happen as part of the query operators, requiring no external administration. Figure 4.4 shows an example of two queries cracking a column using their selection predicates as the partitioning bounds. Query Q1 cuts the column in three pieces and then Q2 enhances this partitioning more by cutting the first and the last piece even further, where its low and high bound fall.

Cracking brings drastic improvements in column-store performance. For example, in recent experiments with the Sloan Digital Sky Survey that collects query and data logs from astronomy, MonetDB with cracking enabled finished answering 160.000 queries, while plain MonetDB was still half way creating the proper indices and without having answered a single query [37]. Similarly, in experiments with the business standard TPC-H benchmark, perfectly preparing MonetDB with all the proper indices/projections took 3 hours, while MonetDB with cracking enabled answered all queries in a matter of a few seconds with zero preparation [50]. Figure 4.5 depicts such an example from the analysis in [50] on TPC-H factor 10. The plain column-store (MonetDB) achieves good improvement over the row-store system (MySQL) even when the row-store uses B-tree indexing (MySQL presorted). Using column-store projections (uncompressed) brings even further improvements (MonetDB/presorted) but at a huge initialization cost; in these experiments it takes 3 hours to prepare the perfect projections for the TPC-H benchmark. On the other hand, when cracking is enabled MonetDB can immediately start processing queries without any preparation; after a few queries cracking reaches optimal performance, i.e., similar to that of the perfectly prepared system which had to spend a significant amount of time preparing (and in addition assumes we had good workload knowledge).

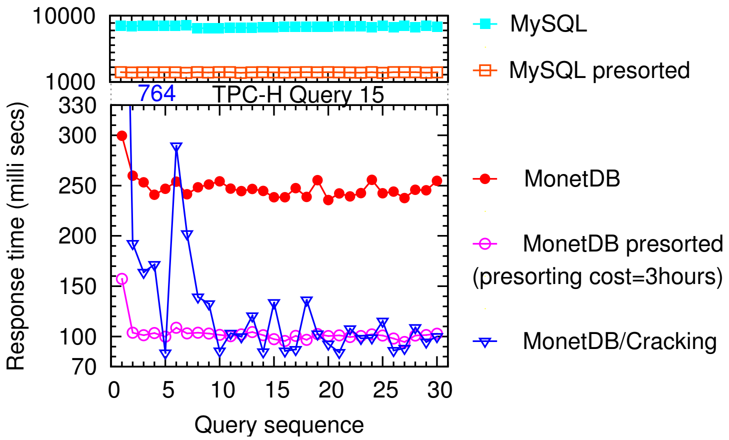


Figure 4.5: Adaptively and automatically improving column-store performance with cracking [50].

The terminology “cracking” reflects the fact that the database is partitioned (cracked) into smaller and manageable pieces. Cracking gradually improves data access, eventually leading to a significant speed-up in query processing [48, 50], even during updates [49]; as it is designed over a column-store it is applied at the attribute level; a query results in reorganizing the referenced column(s), not the complete table; it is propagated across multiple columns on demand, depending on query needs with *partial sideways cracking* [50], whereby pieces of cracker columns are dynamically created and deleted based on storage restrictions. In [35], the authors show how to enable concurrent queries via limited concurrency control effort, relying purely on latches as cracking read queries change only the index structure while the index contents remain intact. In addition, stochastic cracking [37] performs non-deterministic cracking actions by following query bounds less strictly. This way it allows for a more even spread of the partitioning across a column, preventing the lingering of large unindexed areas that are expensive to crack in the future.

Subsequent efforts [51] extended the original cracking to adopt a partition/merge-like logic with active sorting steps or with less active radix partitioning, while working on top of column-stores where each attribute is stored as a collections of vectors as opposed to a single array. While original cracking can be seen as an incremental quicksort (where the pivots are driven by the queries), these latest cracking versions explore the space between incremental quicksort and incremental external merge sort to devise a series of adaptive indexing algorithms (from very active to very lazy).

Database cracking followed a radically different approach; up to this point the query processing time was considered sacred and nothing else could happen other than processing the current query. Cracking, on the other hand, goes ahead and refines indexes on-the-fly, gaining both an immediate and a long term performance benefit. This is a direct side-effect of exploiting certain column-store architecture features. In particular, bulk processing and columnar storage enabled these adaptive indexing ideas. By storing data one column at a time, as dense and fixed width arrays, stored in continuous memory areas, means that database cracking may easily rearrange an array at minimum cost compared to having to deal with traditional slotted pages where locating even a single value may require an indirection. In addition, bulk processing means that each operator fully consumes its input column before the query plan proceeds to the next operator. For database cracking this means that each operator may work on top of a single column in one go, performing efficiently all refinement actions. Vectorized processing works equally well with the main difference that each vector is independently cracked and depending on the policy data may also move across vectors [51].

The C-store projections discussed in previous sections are a form of indexing as they allow multiple orders to be stored with a leading attribute which is sorted for each distinct projection. At a high level, the cracking architecture and in particular the sideways cracking architecture may be seen as a way to achieve the same result as C-store projections but in an adaptive way, i.e., we do not need to decide up front which projections we are going to create and we do not need to invest idle time in creating all projections up-front in one go.

Other than the benefits in terms of not needing workload knowledge and idle time, database cracking also allows one to use database systems without the complexity of tuning. As such it can bring down the cost of setting up and using database systems as with cracking one does not need a database administrator to take indexing decisions and to maintain the indexing set-up over time as the workload fluctuates.

The concept of gradually and on-the-fly adapting index structures as is done in database cracking has not been studied in the past in database research. One work that comes close to some of the concepts of cracking is the work on partial indexes [87] which allows one to create a traditional non-adaptive index on part of a table only, thus avoiding indexing data which potentially is not relevant for queries (assuming good workload knowledge).

4.9 Summary and Design Principles Taxonomy

The design principles described in this chapter have been adopted by most column-store database systems and provide a common ground for all subsequent main-memory and cache conscious designs.

As it is evident by the plethora of those features, modern column-stores go beyond simply storing data one column-at-a-time; they provide a completely new database architecture and execution engine tailored for modern hardware and data analytics.

In many cases, pure columnar storage helps to maximize utilization

	Row-stores	Column-stores
Minimize Bits Read		
(1) Skip loading of not-selected attributes	Vertical partitioning, e.g., [12] PAX [5] Multi-resolution blocks [94] Column indexes [63]	Columnar storage
(2) Work on selected attributes only (per-operator)	Index only plans, e.g., [69, 25] Index anding, e.g., [80]	Late materialization
(3) Skip non-qualified values	Indexes, e.g, [34] Multi-dimensional clustering, e.g., [75] Zone maps	Projections Cracking
(4) Skip redundant bits	Compression, e.g., [30]	Per-column compression
(5) Adaptive/partial indexing	Partial indexes [87]	Database cracking
Minimize CPU Time		
(1) Minimize instruction and data misses	Block processing [74] Buffer operators [95] Cache conscious operators [85]	Vectorized execution
(2) Minimize processing for each bit read	Operating on compressed data, e.g., [30]	Operating on compressed columns
(3) Tailored operators	Compiled queries, e.g., [70]	RISC style algebra

Figure 4.6: Taxonomy of design features that define modern column-stores and similar concepts that have appeared in isolation in past row-store research.

of these new design principles, i.e., compression is much more effective when applied at one column-at-a-time or vectorization and block processing help minimize cache misses and instruction misses even more when carrying one column-at-a-time, i.e., only the column relevant and for one (vector-based) operator. In this sense, we can say that a modern column-store system as it was redefined mainly by VectorWise and C-Store is a system that includes all those design principles, not just a column-oriented storage layout.

As we discussed throughout this and previous chapters some of the design principles that came together in modern column-stores have been investigated in some form or another in the past in the context of traditional row-stores. There is no system, however, to provide a holistic and from scratch design and implementation of a complete DBMS with all those design principles until MonetDB, VectorWise and C-Store were proposed. Essentially, they marked the need for a complete redesign of database kernels, inspired by decades of research in DBMS architectures in the database community.

Figure 4.6 summarizes the discussion in this chapter by providing a list of features and design principles that altogether define modern column-stores along with pointers to similar but isolated features that have appeared in the past in the context of row-stores.

5

Discussion, Conclusions, and Future Directions

In this section, we briefly compare MonetDB, VectorWise and C-Store. We also discuss the feasibility of emulation of a column-store in a row-oriented database, and present conclusions and future work.

5.1 Comparing MonetDB/VectorWise/C-Store

Read-optimized database systems clearly benefit from CPU-efficient query execution. To this end, all three architectures – MonetDB, VectorWise and C-Store – use some form of block-oriented execution [74], but in different ways. MonetDB takes the extreme with its column-at-a-time execution that implies full materialization. Both C-Store and VectorWise allow pipelined execution, with blocks of tuples rather than single tuples being passed between operators. In C-Store this mostly happens as a side-effect of compressed execution, where blocks of tuples are kept as long as possible in a compressed format. VectorWise, in contrast, follows a vectorized execution strategy in its entire execution and storage architecture.

For handling loads and updates, MonetDB and C-Store follow a similar approach, using a deletion bitmap and temporary tables holding inserts (WOS). Especially in large join queries, this leads to merging overhead where the inserts and deletes need to be applied to the main tuple stream before entering a join. The Positional Delta Tree (PDT) data structure of VectorWise0 significantly reduces this overhead, though the downside of PDTs is that they are quite complex, and insert and delete operations are potentially more expensive.

Plain MonetDB does not employ compression (except automatic string dictionaries), nor does it store tables in any particular order. Both VectorWise and C-Store heavily employ compression, though only C-Store offers compressed execution. C-Store follows the idea of storing data multiple times in projections with a different sort order. VectorWise uses a highly sparse index on all columns storing minimum and maximum values for tuple ranges, which can reduce I/O for range predicates. It further reduces I/O pressure using cooperative scans.

5.2 Simulating Column/Row Stores

One common question about column-oriented databases is whether it is possible to emulate a column-based system using a conventional row-oriented system. There are two methods that can be used to perform this kind of emulation: using a fully vertically partitioned design and creating an index on every column [1]. We will discuss each of these techniques in turn, and list the associated disadvantages of each one.

Vertical Partitioning. The most straightforward way to emulate a column-store approach in a row-store is to fully vertically partition each relation, as was done in the early column-stores described above [58]. In a fully vertically partitioned approach, some mechanism is needed to connect fields from the same row together (column stores typically match up records implicitly by storing columns in the same order, but such optimizations are not available in a row store). To accomplish this, the simplest approach is to add an integer “position" column to every table (this is often preferable to using the primary key because primary keys can be large and are sometimes composite). For example, given the following employee table:

Name	Age	Salary
Smith	40	56,000
Johnson	29	34,000
Williams	56	78,000
Jones	34	35,000

We would add a simple position column:

Position	Name	Age	Salary
1	Smith	40	56,000
2	Johnson	29	34,000
3	Williams	56	78,000
4	Jones	34	35,000

We would then create one physical table for each column in the logical schema, where the i^{th} table has two columns, one with values from column i of the logical schema and one with the position of that value in the original table (this column is typically a dense sequence of integers starting from 1):

1	Smith
2	Johnson
3	Williams
4	Jones

1	40
2	29
3	56
4	34

1	56,000
2	34,000
3	78,000
4	35,000

Queries are then rewritten to perform joins on the position attribute when fetching multiple columns from the same relation.

The main disadvantages of this approach are (1) the additional space overhead of the position data (and the associated performance cost when scanning these columns due to the additional I/O) (2) the need to implement a query rewriting layer to convert queries over the original logical schema to the new physical schema (3) the explosion in the number of joins per query that overwhelm the optimizer of most row-store DBMS implementations and cause heuristics to be used that can occasionally have disastrous effects on query performance (4) since each column is stored in a separate table, each column now contains a (potentially large) tuple header for each (much smaller) data value (column-stores do not include a header for each column — instead they store a singe tuple header in a separate column) (5) the inability to use column-oriented compression schemes such as RLE (though this disadvantage can be alleviated with some cleverness — [20]).

Creating an index on every column. This approach involves creating a separate index for each column of each table in a database system. This solves most of the problems of vertical partitioning described above (at least problems (2), (3), and (4)), but has its own set of problems. The most obvious problem is the space and update overhead of the large number of indexes. However, the more subtle problem is that each index will not generally store the values inside the index in the same order that they appear in the raw table. Therefore, materializing two (or more) attributes into rows (which, as described in Section 4.4, needs to happen at some point during query execution) requires a full-fledged join on tuple-id (unlike the vertically partitioned case, where this join can be performed by simply merging together the two columns, here the two sets of values are entirely unaligned). Hence, in practice, the database optimizer will do column projection using the original row-store table, which results in the column-store I/O benefits of only having to read in the necessary columns being completely negated.

5.3 Conclusions

We described a number of architectural innovations that make modern column-stores like MonetDB, VectorWise and C-Store able to provide very good performance on analytical workloads. These include compression, vectorization, late materialization, and efficient column-oriented join methods. These ideas have found their way into a several commercial products—both direct descendants of the academic projects (e.g., VectorWise and Vertica) as well as a number of other products (e.g., Aster Data, Greenplum, Infobright, Paraccel, etc). Even row-oriented stalwart Oracle has implemented some column-oriented techniques in their database appliance, Exadata (in particular, they have implemented PAX page layouts and column-oriented compression). These products are reputed to provide one to two orders of magnitude better performance than older-generation row-oriented systems on typical data warehousing and analytics workloads, and have been quite successful commercially (with VectorWise, Vertica, Greenplum, and Aster Data all having been recently acquired).

Despite the academic and commercial success of column-oriented systems, there are still several interesting directions for future research. In particular, there is a substantial opportunity for hybrid systems that are partially column-oriented. For example, systems that store groups of frequently accessed columns together could provide better performance than a pure column-store. Additionally, systems that adaptively choose between column and row-oriented layouts for tables, depending on access patterns over time, will likely become important, as requiring users to determine which type of layout for their data is not ideal [45]. Microsoft recently announced a columnar storage option with vectorized query processing [61] for its SQLServer product. While these features are still limited (the system is read-only, only vectorizes a subset of operators and data types, and does not make automatic data layout decisions), it is a step in this direction.

We also expect that column-oriented ideas will start to find their way into other data processing systems, such as Hadoop/MapReduce [27], which is increasingly being used for analytic-style processing on very large data sets.

References

[1] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 967–980, 2008.

[2] Daniel J. Abadi, Samuel R. Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 671–682, 2006.

[3] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel R. Madden. Materialization strategies in a column-oriented DBMS. In *Proceedings of the International Conference on Data Endineering (ICDE)*, pages 466–475, 2007.

[4] R. Abdel Kader, P. Boncz, S. Manegold, and M. van Keulen. ROX: run-time optimization of xqueries. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 615–626. ACM, 2009.

[5] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 169–180, 2001.

[6] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively Parallel Sort-Merge Joins in Main Memory Multi-Core Database Systems. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 5(10):1064–1075, 2012.

[7] Sihem Amer-Yahia and Theodore Johnson. Optimizing queries on compressed bitmaps. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 329–338, 2000.

[8] G. Antoshenkov. Byte-aligned data compression. U.S. Patent Number 5,363,098, 1994.

[9] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 7(1):85–96, 2013.

[10] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the International Conference on Data Endineering (ICDE)*, pages 362–373, 2013.

[11] Ronald Barber, Peter Bendel, Marco Czech, Oliver Draese, Frederick Ho, Namik Hrle, Stratos Idreos, Min-Soo Kim, Oliver Koeth, Jae-Gil Lee, Tianchao Tim Li, Guy M. Lohman, Konstantinos Morfonios, René Müller, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Richard Sidle, Knut Stolze, and Sandor Szabo. Business Analytics in (a) Blink. *IEEE Data Eng. Bull.*, 35(1):9–14, 2012.

[12] Don S. Batory. On searching transposed files. *ACM Transactions on Database Systems*, 4(4):531–544, 1979.

[13] Philip A. Bernstein and Dah-Ming W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, 1981.

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

[15] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 283–296, 2009.

[16] Peter Boncz. Monet: A next-generation DBMS kernel for query-intensive applications. *University of Amsterdam, PhD Thesis*, 2002.

[17] Peter Boncz, Stefan Manegold, and Martin Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 54–65, 1999.

[18] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-pipelining query execution. In *Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR)*, 2005.

[19] Peter A. Boncz and Martin L. Kersten. MIL primitives for querying a fragmented world. *VLDB Journal*, 8(2):101–119, 1999.

[20] Nicolas Bruno. Teaching an old elephant new tricks. In *Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR)*, 2009.

[21] Surajit Chaudhuri and Vivek R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 146–155, 1997.

[22] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 268–279, 1985.

[23] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, 1990.

[24] David DeWitt. From 1 to 1000 mips, November 2009. PASS Summit 2009 Keynote.

[25] Amr El-Helw, Kenneth A. Ross, Bishwaranjan Bhattacharjee, Christian A. Lang, and George A. Mihaila. Column-oriented query processing for row stores. In *Proceedings of the International Workshop On Data Warehousing and OLAP*, pages 67–74, 2011.

[26] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[27] Avrilia Floratou, Jignesh M. Patel, Eugene J. Shekita, and Sandeep Tata. Column-Oriented Storage Techniques for MapReduce. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 4(7):419–429, 2011.

[28] Clark D. French. “One Size Fits All” Database Architectures Do Not Work for DDS. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 449–450, 1995.

[29] Clark D. French. Teaching an OLTP Database Kernel Advanced Data Warehousing Techniques. In *Proceedings of the International Conference on Data Endineering (ICDE)*, pages 194–198, 1997.

[30] G.Graefe and L.Shapiro. Data compression and database performance. In ACM/IEEE-CS Symp. On Applied Computing, pages 22 -27, April 1991.

[31] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *Proceedings of the International Conference on Data Endineering (ICDE)*, pages 370–379, 1998.

[32] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[33] Goetz Graefe. Efficient columnar storage in b-trees. *SIGMOD Rec.*, 36(1):3–6, 2007.

[34] Goetz Graefe. Modern B-Tree Techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.

[35] Goetz Graefe, Felix Halim, Stratos Idreos, Harumi Kuno, and Stefan Manegold. Concurrency Control for Adaptive Indexing. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 5(7):656–667, 2012.

[36] Goetz Graefe, Stratos Idreos, Harumi Kuno, and Stefan Manegold. Benchmarking Adaptive Indexing. In *Proceedings of the TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC)*, pages 169–184, 2010.

[37] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 5(6):502–513, 2012.

[38] Alan Halverson, Jennifer L. Beckmann, Jeffrey F. Naughton, and David J. Dewitt. A Comparison of C-Store and Row-Store in a Common Framework. Technical Report TR1570, University of Wisconsin-Madison, 2006.

[39] Richard A. Hankins and Jignesh M. Patel. Data morphing: an adaptive, cache-conscious storage technique. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 417–428, 2003.

[40] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel R. Madden. Performance tradeoffs in read-optimized databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 487–498, 2006.

[41] S. Héman, M. Zukowski, N.J. Nes, L. Sidirourgos, and P. Boncz. Positional update handling in column stores. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 543–554, 2010.

[42] William Hodak. Exadata hybrid columnar compression. Oracle Whitepaper, 2009. http://www.oracle.com/technetwork/database/exadata/index.html.

[43] Allison L. Holloway, Vijayshankar Raman, Garret Swart, and David J. DeWitt. How to barter bits for chronons: compression and bandwidth trade offs for database scans. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 389–400, 2007.

[44] Stratos Idreos. Database Cracking: Towards Auto-tuning Database Kernels. *CWI, PhD Thesis*, 2010.

[45] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. Here are my Data Files. Here are my Queries. Where are my Results? In *Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR)*, pages 57–68, 2011.

[46] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin L Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[47] Stratos Idreos, Raghav Kaushik, Vivek R. Narasayya, and Ravishankar Ramamurthy. Estimating the compression fraction of an index using sampling. In *Proceedings of the International Conference on Data Endineering (ICDE)*, pages 441–444, 2010.

[48] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR)*, pages 68–78, 2007.

[49] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Updating a cracked database. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 413–424, 2007.

[50] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-organizing tuple reconstruction in column stores. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 297–308, 2009.

[51] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 4(9):585–597, 2011.

[52] Ryan Johnson, Vijayshankar Raman, Richard Sidle, and Garret Swart. Row-wise parallel predicate evaluation. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 1(1):622–634, 2008.

[53] Theodore Johnson. Performance measurements of compressed bitmap indices. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 278–289, 1999.

[54] Ilkka Karasalo and Per Svensson. The design of cantor: a new system for data analysis. In *Proceedings of the 3rd international workshop on Statistical and scientific database management*, pages 224–244, 1986.

[55] Illka Karasalo and Per Svensson. An overview of cantor: a new system for data analysis. In *Proceedings of the 2nd international Workshop on Statistical Database Management (SSDBM)*, pages 315–324, 1983.

[56] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the International Conference on Data Endineering (ICDE)*, pages 195–206, 2011.

[57] Alfons Kemper, Thomas Neumann, Florian Funke, Viktor Leis, and Henrik Mühe. HyPer: Adapting Columnar Main-Memory Data Management for Transactional AND Query Processing. *IEEE Data Eng. Bull.*, 35(1):46–51, 2012.

[58] Setrag Khoshafian, George Copeland, Thomas Jagodis, Haran Boral, and Patrick Valduriez. A query processing strategy for the decomposed storage model. In *Proceedings of the International Conference on Data Endineering (ICDE)*, pages 636–643, 1987.

[59] Setrag Khoshafian and Patrick Valduriez. Parallel execution strategies for declustered databases. In *Proceedings of the International Workshop on Database Machines*, pages 458–471, 1987.

[60] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 5(12):1790–1801, 2012.

[61] P.Å. Larson, C. Clinciu, E.N. Hanson, A. Oks, S.L. Price, S. Rangarajan, A. Surna, and Q. Zhou. Sql server column store indexes. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 1177–1184, 2011.

[62] Per-Åke Larson, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L. Price, Srikumar Rangarajan, Remus Rusanu, and Mayukh Saubhasik. Enhancements to sql server column stores. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 1159–1168, 2013.

[63] Per-Åke Larson, Eric N. Hanson, and Susan L. Price. Columnar Storage in SQL Server 2012. *IEEE Data Eng. Bull.*, 35(1):15–20, 2012.

[64] Zhe Li and Kenneth A. Ross. Fast joins using join indices. *VLDB Journal*, 8:1–24, April 1999.

[65] R.A. Lorie and A.J. Symonds. A relational access method for interactive applications. In *Courant Computer Science Symposia, Vol. 6: Data Base Systems*. Prentice Hall, 1971.

[66] Roger MacNicol and Blaine French. Sybase IQ multiplex - designed for analytics. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1227–1230, 2004.

[67] S. Manegold, P. Boncz, N. Nes, and M. Kersten. Cache-conscious radix-decluster projections. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 684–695, 2004.

[68] A. Moffat and J. Zobel. Compression and fast indexing for multi-gigabyte text databases. *Australian Computer Journal*, 26(1):1–9, 1994.

[69] C. Mohan, Donald J. Haderle, Yun Wang, and Josephine M. Cheng. Single Table Access Using Multiple Indexes: Optimization, Execution, and Concurrency Control Techniques. pages 29–43, 1990.

[70] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 4(9):539–550, 2011.

[71] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 38–49, 1997.

[72] Patrick E. O'Neil. Model 204 architecture and performance. In *Proceeding of the International Workshop on High Performance Transaction Systems*, pages 40–59, 1987.

[73] Patrick E. O'Neil, Elizabeth J. O'Neil, and Xuedong Chen. The Star Schema Benchmark (SSB). `http://www.cs.umb.edu/~poneil/StarSchemaB.PDF`.

[74] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *Proceedings of the International Conference on Data Endineering (ICDE)*, pages 567–574, 2001.

[75] Sriram Padmanabhan, Bishwaranjan Bhattacharjee, Timothy Malkemus, Leslie Cranston, and Matthew Huras. Multi-Dimensional Clustering: A New Data Layout Scheme in DB2. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 637–641, 2003.

[76] M. Poess and D. Potapov. Data compression in oracle. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 937–947, 2003.

[77] Bogdan Raducanu, Peter A. Boncz, and Marcin Zukowski. Micro adaptivity in vectorwise. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 1231–1242, 2013.

[78] Ravishankar Ramamurthy, David Dewitt, and Qi Su. A case for fractured mirrors. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 89 – 101, 2002.

[79] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang. DB2 with BLU Acceleration: So much more than just a column store. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 6(11), 2013.

[80] Vijayshankar Raman, Lin Qiao, Wei Han, Inderpal Narang, Ying-Lin Chen, Kou-Horng Yang, and Fen-Ling Ling. Lazy, adaptive rid-list intersection, and its application to index anding. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 773–784, 2007.

[81] Vijayshankar Raman, Garret Swart, Lin Qiao, Frederick Reiss, Vijay Dialani, Donald Kossmann, Inderpal Narang, and Richard Sidle. Constant-Time Query Processing. In *Proceedings of the International Conference on Data Endineering (ICDE)*, pages 60–69, 2008.

[82] Mark A. Roth and Scott J. Van Horn. Database compression. *SIGMOD Rec.*, 22(3):31–39, 1993.

[83] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. The Uncracked Pieces in Database Cracking. *Proceedings of the Very Large Data Bases Endowment (PVLDB)*, 7(2), 2013.

[84] Minglong Shao, Jiri Schindler, Steven W. Schlosser, Anastassia Ailamaki, and Gregory R. Ganger. Clotho: Decoupling memory page layout from storage organization. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 696–707, 2004.

[85] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache Conscious Algorithms for Relational Query Processing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 510–521, 1994.

[86] Lefteris Sidirourgos and Martin L. Kersten. Column imprints: a secondary index structure. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 893–904, 2013.

[87] Michael Stonebraker. The case for partial indexes. *SIGMOD Record*, 18(4):4–11, 1989.

[88] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel R. Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alexander Rasin, Nga Tran, and Stan B. Zdonik. C-Store: A Column-Oriented DBMS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 553–564, 2005.

[89] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. Query processing techniques for solid state drives. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 59–72, 2009.

[90] Stephen Weyl, James Fries, Gio Wiederhold, and Frank Germano. A modular self-describing clinical databank system. *Computers and Biomedical Research*, 8(3):279 – 293, 1975.

[91] K. Wu, E. Otoo, and A. Shoshani. Compressed bitmap indices for efficient query processing. Technical Report LBNL-47807, 2001.

[92] K. Wu, E. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 99–108, 2002.

[93] K. Wu, E. Otoo, A. Shoshani, and H. Nordberg. Notes on design and implementation of compressed bit vectors. Technical Report LBNL/PUB-3161, 2001.

[94] Jingren Zhou and Kenneth A. Ross. A Multi-Resolution Block Storage Model for Database Design. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)*, pages 22–33, 2003.

[95] Jingren Zhou and Kenneth A. Ross. Buffering Database Operations for Enhanced Instruction Cache Performance. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 191–202, 2004.

[96] M. Zukowski. Balancing vectorized query execution with bandwidth-optimized storage. *University of Amsterdam, PhD Thesis*, 2009.

[97] M. Zukowski, S. Héman, and P. Boncz. Architecture-conscious hashing. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, 2006.

[98] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: dynamic bandwidth sharing in a DBMS. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 723–734, 2007.

[99] Marcin Zukowski and Peter A. Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 35(1):21–27, 2012.

[100] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the International Conference on Data Endineering (ICDE)*, 2006.

[101] Marcin Zukowski, Niels Nes, and Peter Boncz. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*, pages 47–54, 2008.