14–17 minutes

Why Use A Column-Store DBMS?

Three technologies that advanced AI, business intelligence, and data analytics

Introduction

Since the turn of this millennium, there are a few significant advances in database architecture that have changed the database landscape. From these architectural advances, column-store database management systems (DBMS) have emerged. Column-store DBMSs have enabled the rapid growth in data analytics, artificial intelligence, and machine learning. This article discusses highlights of three particular architectural advances and challenges that they address.

Photo by Eduardo Soares on Unsplash

Why Use a Column-store DBMS?

Data analytics requires fast and efficient database query execution. Data analytics applications like artificial intelligence, machine learning, and business intelligence present significant challenges for traditional row-oriented database management systems (DBMS).

The most widely used DBMSs are row-oriented. In their underlying databases, the data are represented as tables organized by rows and columns—like spreadsheets. Each row is a record (tuple) with attributes that represent the corresponding columns for that row. Row-oriented databases are best suited for online transaction processing (OLTP) such as sales transactions, customer relationship management systems, banking, e-commerce, etc. These applications typically process records row by row, where each record for example, identifies a customer, product, or transaction that has related attributes.

Column-store, column-oriented, and columnar are terms that are used interchangeably to describe these new architectures. A column-store DBMS addresses many of the challenges that data analytics applications confront with traditional row-oriented relational

databases. Column-stores address these challenges by embracing new architectures that enable them to apply advanced techniques, enabled by more powerful hardware. This article will explore some of these architectures and how they benefit data analytics.

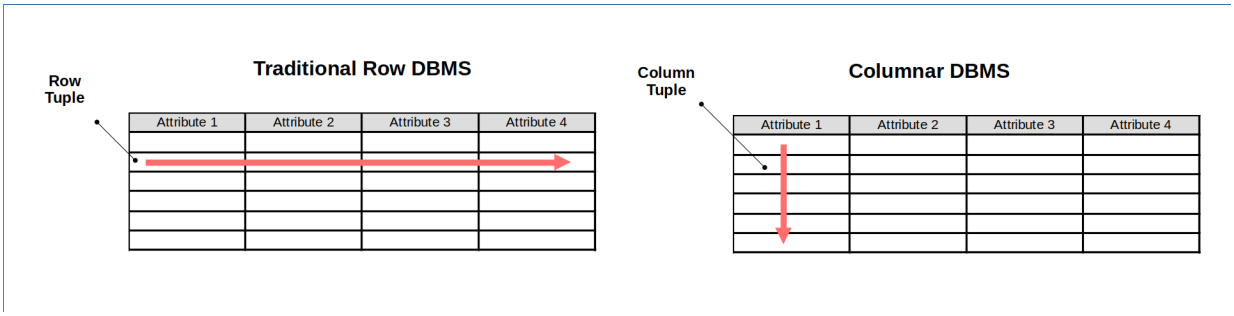


Figure 1. Row vs. column tuples in traditional row-store databases and column-store databases.

What is a column-store DBMS?

In contrast to row-oriented OLTP applications, data analytics applications—predominantly used for online analytics processing (OLAP)—require data to be retrieved primarily by columns, not rows. As a result, row-oriented databases are not ideally suited for OLAP applications. Getting even a single column from a traditional database would potentially require every row in a database to be retrieved. Applications like data mining, machine learning, and artificial intelligence use large volumes of data, on the order of terabytes. While retrieving a single column from a traditional database may be feasible for a relatively small database, it is not feasible for a large one. Performance, cost, complexity, and hardware capacity make it unfeasible. Column-store databases are optimized to retrieve column data faster—by one to two orders of magnitude—than row-oriented databases because they don't need to read every row in a table just to retrieve data from a single column and they take advantage of modern CPUs that support single instruction, multiple

data (SIMD) parallel processing features. [1], [6]

For a simple example (See figure 2), consider a sales transaction database in which a transaction record consists of six fields: *customer ID, customer name, product ID, and purchase date*. If you are doing business intelligence analysis and you want to gain insights into buying patterns by product ID, or seasonal buying patterns by date, you would need to read each row in the database and scan for the target attribute in each row. You can see how large tables could quickly consume a lot of CPU time and storage. In a column-store database, you would retrieve only the column(s) of interest.

CustomerID	CustomerName	ProductID	PurchaseDate
1	Alpha	111	2023-12-15
2	Beta	111	2023-03-27
3	Gamma	999	2023-07-14
4	Delta	999	2023-01-30
5	Epsilon	333	2023-03-13
6	Zeta	111	2023-03-30

Figure 2. Sales transaction database example.

Architectural features and techniques

To address the *performance and efficiency* challenges inherent in row-oriented databases, database architects have explored a variety of architectural approaches. Traditional row-oriented relational databases were conceived in the late 1960s and early 1970s when hardware limitations influenced their design.

Solutions to performance and efficiency challenges began to accelerate in the late 1990s and early 2000s with academic projects that became the foundations for commercial products. Three notable academic projects were **C-Store**, **MonetDB**, and **VectorWise**. C-Store evolved into Ingres VectorWise and Vertica. VectorWise borrowed architectural design techniques from both MonetDB and C-Store, and evolved into Actian Vector. [1]

At the same time, technology advancements in CPUs, cache memory, and storage technology made it possible to apply new architectural techniques that previously weren't feasible.

There are three **advanced architectural techniques** that involve changes to data structures, data compression, data organization, and indexing. They are:

- **Late materialization**
- **Column-at-a-time query processing**
- **Vectorization**

Data model. In a traditional database, rows are stored as tuple objects and columns are attributes of those objects. In order to operate on a column as a tuple object during a query, the column tuple must be constructed from the common attributes from each row. This is called *materialization*. In column-store architectures, column objects are a feature of the architecture, *eliminating the slower, inefficient process of tuple construction*.

C-Store took a read-optimized approach to their database implementation to improve the inherent inefficiencies associated with traditional databases. OLAP applications are primarily read-oriented, so focusing on read optimization of column data was the first priority.

Toward that end, C-Store developed a novel data model to physically store data. That model employed a *vertical partitioning* technique called *projection* to store column data in separate compressed files on disk.

Projection. Projections in C-Store are redundant column representations implemented as on-disk files, some of which potentially represent a single column in multiple sort orders. (See figure 3.) This technique eliminates the overhead incurred by materialization required on each query in traditional databases.

ProductID	PurchaseDate
111	2023-12-15
111	2023-03-27
111	2023-03-30
333	2023-03-13
999	2023-07-14
999	2023-01-30

Projection 1
Sorted by ProductID

ProductID	PurchaseDate
999	2023-01-30
333	2023-03-13
111	2023-03-27
111	2023-03-30
999	2023-07-14
111	2023-12-15

Projection 2
Sorted by PurchaseDate

Figure 3. Projections sorted by Product ID and by PurchaseDate.

Late materialization

By design, all three column-store DBMSs discussed in this article delay materialization until as late as possible in the query process to minimize materialization overhead. This strategy is call *late materialization*.

Simplified operators. In order to execute column queries in traditional databases, operators such as `SELECT` and `JOIN` need to create intermediate tuples early in the query execution process (*early materialization*), which can incur significant performance overhead. MonetDB implemented a solution using column-at-a-time operators

that used a reduced set of simplified math operators.

Advantages of late materialization. In OLAP applications, there are four advantages [1] that late materialization offers:

1. SELECT and AGGREGATE operators
2. Decompression/recompression
3. Direct column operators
4. Fixed width columns

SELECT and AGGREGATE operators. By operating directly on columns, CPU time wasted on tuple construction in traditional DBMS systems is significantly reduced. There is no need to reconstruct columns from row tuples. The SELECT and AGGREGATE operator can focus on only relevant data. Also, some intermediate tuples are eliminated during the query execution process. As a result, waiting until the end of the process (aggregation) reduces the time spent reconstructing the data into a form suitable for user presentation.

Decompression/recompression. Compression is a common technique used in column-store DBMSs. It saves disk space, memory space, and reduces the volume of data necessary to transfer from disk or memory to the CPU.

The first of the early academic projects to apply compression techniques to column-store databases was C-Store. C-Store developed query operators capable of operating on compressed data, eliminating the decompression/compression overhead inherent in traditional databases because tuple reconstruction must be performed on uncompressed data. Late materialization delays tuple reconstruction until the end of the query process. [2]

Direct column operators. Cache performance improves by

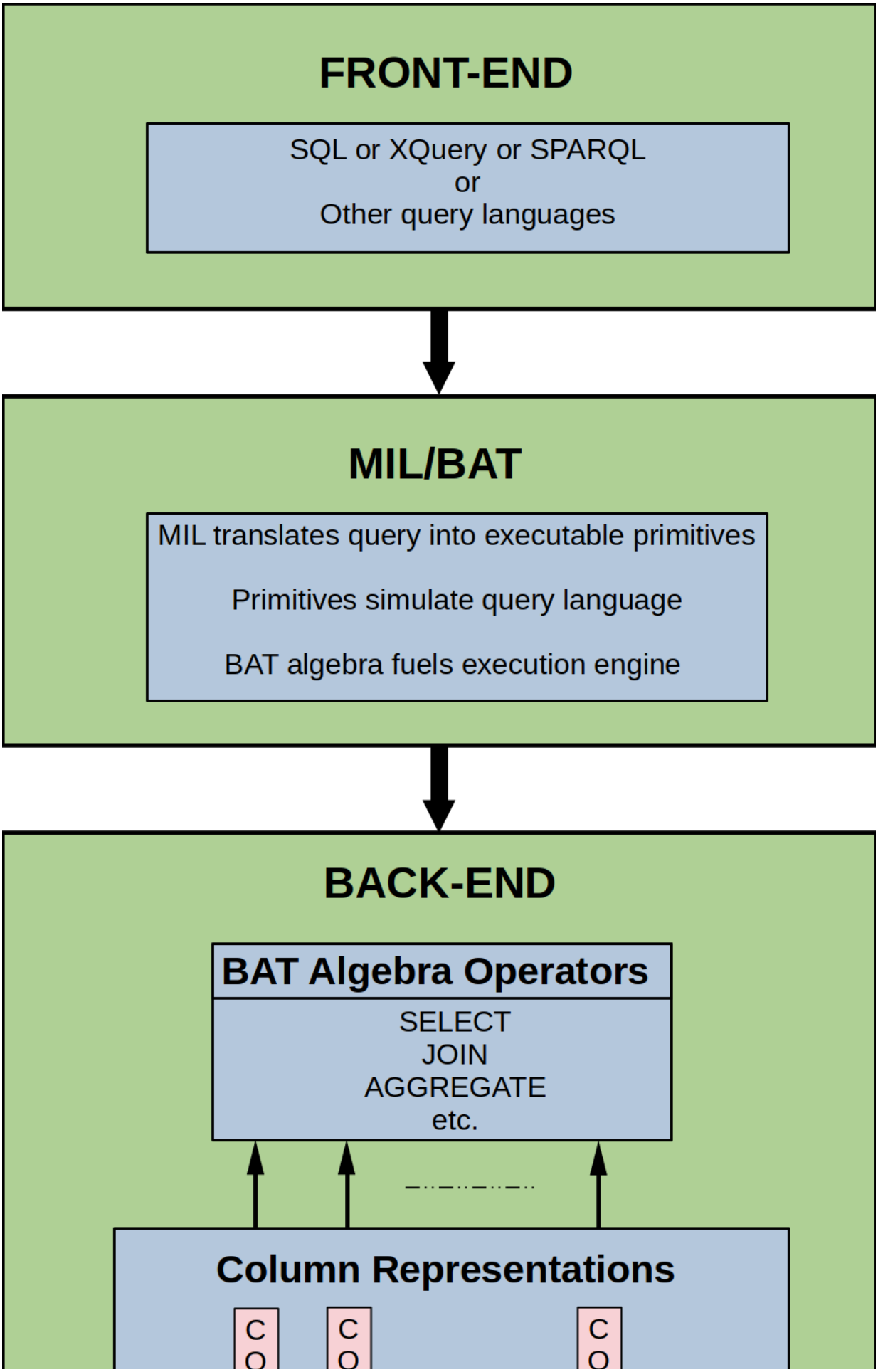
operating directly on column data. Operating directly on columns minimizes the number of decompression/tuple construction cycles during query execution and eliminates the need to scan through superfluous attribute data.

Fixed width columns. Fixed-width columns can enhance performance in multiple ways. Compression/decompression can be done faster because column values can be dense arrays leveraging CPU SIMD instructions, locating data by offset is simpler, physical reorganization (sorting) can be done faster, and parallelism can be maximized with hardware SIMD instructions.

Column-at-a-time query processing

MonetDB was the first of the three projects to introduce column-specific DBMS technology. The major innovation in this project was column-at-a-time processing. Queries are traditionally executed one tuple (row) at a time, where tuple predicates are interpreted as each operator is executed. Some traditional databases spend as 90% of the query time in tuple-at-a-time interpretation. [6] By contrast, MonetDB's column-at-a-time processing was made possible by implementing a simplified set of memory-mapped operators and removing predicate interpretation overhead from inside the operator code resulting in improved efficiency by increasing query performance for operations like `SELECT` and `JOIN`.

Architecture and relational algebra are the keys to MonetDB's efficiency. The MonetDB architecture (See figure 4.) consists of a front-end user interface and a back-end query optimization engine. The front-end translates the query language (e.g. SQL) into an internal relational algebra representation. The back-end runs relational algebra optimizers to execute the query.


FRONT-END
SQL or XQuery or SPARQL
or
Other query languages
MIL/BAT
MIL translates query into executable primitives
Primitives simulate query language
BAT algebra fuels execution engine
BACK-END
BAT Algebra Operators
SELECT
JOIN
AGGREGATE
etc.
Column Representations

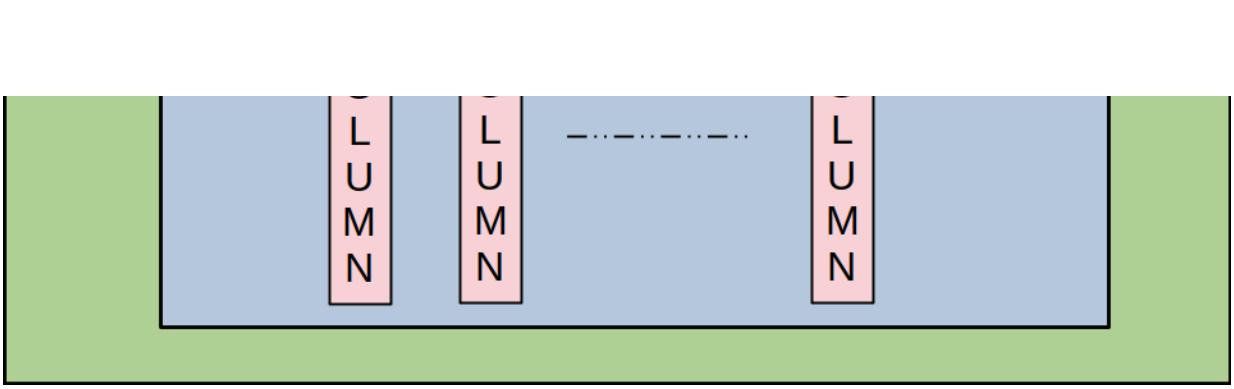


Figure 4. Front-end/back-end architecture.

On the front-end, MonetDB can connect with numerous query tools, including the most commonly used SQL tools, as well as OQL, XQuery, and SPARQL. A Monet interpreter language (MIL) interface translates the corresponding query language into a relational algebra language (BAT algebra) that the back-end execution engine understands. Binary association tables (BAT) are vertical fragmentation model representations of database columns. Combinations of memory-mapped BAT algebra operators are applied to each column to simulate query language operators (e.g. SQL). [3], [4]

Vectorization

While MonetDB pioneered column-at-a-time query processing which dramatically improved OLAP performance over the traditional row tuple-at-a-time approach, there was still more untapped performance to be had. By 2005, modern hardware with multi-core CPUs and large cache memory were dramatically more advanced than the limited hardware resources available when traditional databases were first developed.

MonetDB improved on the row tuple approach by vertical partitioning and running queries in main memory. Vertical partitioning eliminated the inefficiencies associated with runtime materialization thus dedicating more CPU cycles to query operations. Executing queries

with memory-mapped column operators and storing data structures in the same binary column format, in memory and on disk, improved data transfer speed by reducing the amount of data transferred from disk.

Using a combination of column-at-a-time and tuple-at-a-time techniques, the MonetDB team started the X100 project to further improve performance by taking advantage of CPU cache memory improvements and developing a new vector data structure. A vector consists of multiple tuples, where vector operators process N-tuples-at-a-time. (See Figure 5.) Vector operators execute queries in a vector-at-a-time pipeline similar to a tuple-at-a-time pipeline, but with increased bandwidth. [6] The X100 project was subsequently renamed VectorWise.

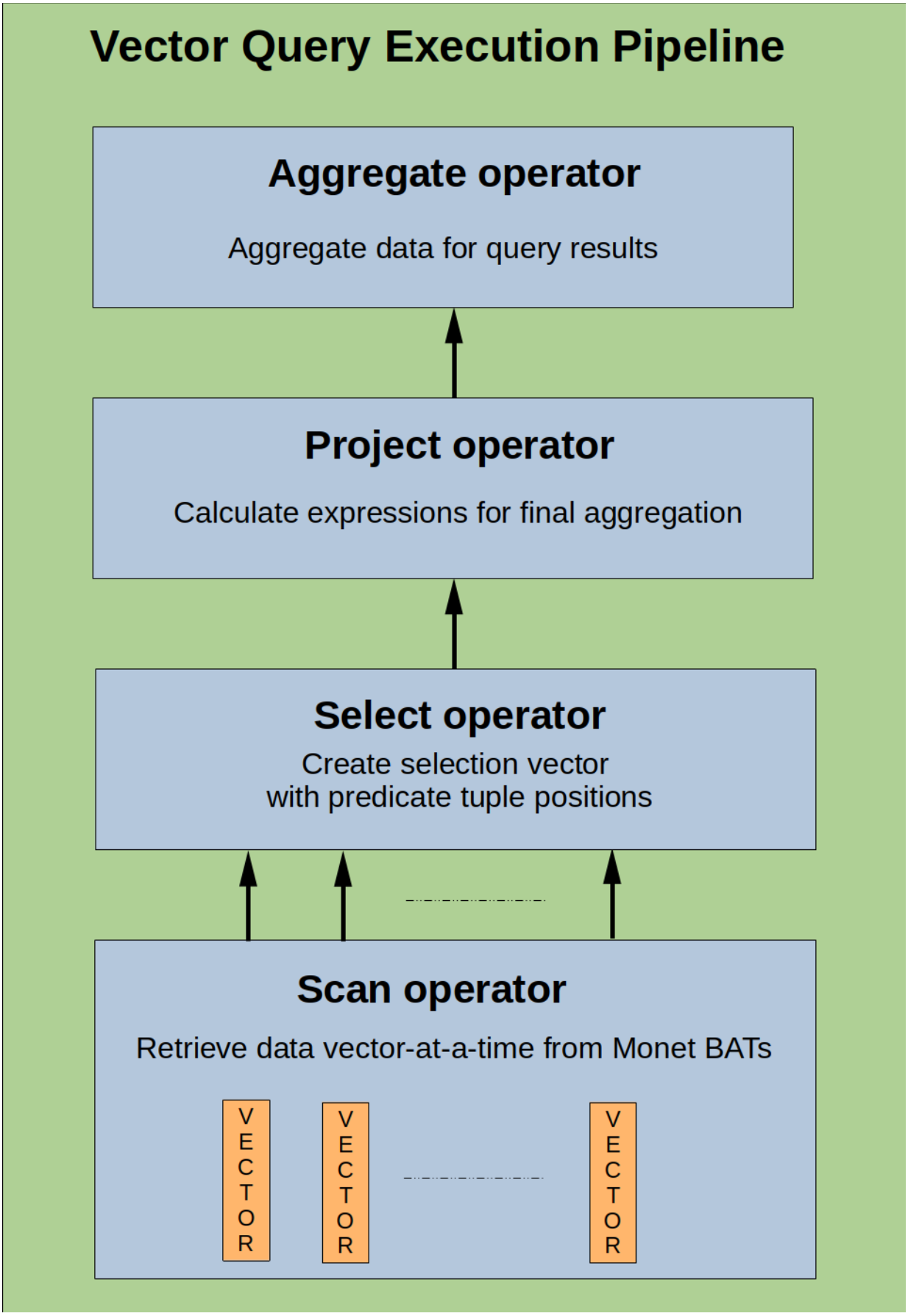


Figure 5. Vector query execution pipeline.

Summary

In addition to the three advanced techniques that we discussed, C-Store, MonetDB, and VectorWise apply various other techniques, including but not limited to:

- Column-specific compression
- Hybrid row/column implementation
- Database cracking and adaptive indexing
- Efficient loading

While a variety of techniques have been and continue to be employed to address the needs of column-store DBMS applications, late materialization, column-at-a-time processing, and vectorization may have been the most significant contributions, given the success of the commercial products that have adopted and deployed them.

References

[1] Abadi, D., Boncz, P., Harizopoulos, S., Idreos, S., & Madden, S. (2013). “The design and implementation of modern column-oriented database systems.” Foundations and Trends® in Databases, 5(3), 197-280., https://stratos.seas.harvard.edu/files/stratos/files/columnstoresfntdbs.pdf

[2] Michael Stonebraker, et al., “C-Store: A Column-oriented DBMS”, Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005, https://web.stanford.edu/class/cs345d-01/rl/cstore.pdf

[3] Nes, Stratos Idreos Fabian Groffen Niels, and Stefan Manegold Sjoerd Mullender Martin Kersten. “MonetDB: Two decades of research in column-oriented database architectures.” Data

Engineering 40 (2012), https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=a0deb3876b526994ac567c129a4cecb5e6ff757f

[4] Peter A. Boncz, Martin L. Kersten, University of Amsterdam, “MIL primitives for querying a fragmented world“, VLDB Journal (1999) 8: 101–119, http://www.cs.unibo.it/~montesi/CBD/Articoli/MIL%20primitives%20for%20querying%20a%20fragmented%20world.pdf

[5] Marcin Zukowski, Peter Boncz, Niels Nes, S´andor H´eman Centrum voor Wiskunde en Informatica, “MonetDB/X100 - A DBMS In The CPU Cache”, https://ir.cwi.nl/pub/11098/11098B.pdf

[6] Boncz, Zukowski, Nes, “MonetDB/X100: Hyper-Pipelining Query Execution”, https://www.researchgate.net/profile/Niels-Nes/publication/45338800_MonetDBX100_Hyper-Pipelining_Query_Execution/links/0deec520cd1e8a3607000000/MonetDB-X100-Hyper-Pipelining-Query-Execution.pdf

End