Course code : **CSE3009**
Course title : **No SQL Data Bases**
Module : **5**
Topic : **4**

Column-store internals

Objectives

This session will give the knowledge about

- Vectorized Processing
- Compression

Vectorized Processing

The query execution layer of database systems generally categorized into the following

- The "Volcano-style" iterator model (tuple-at-a-time pipelining)
- Full materialization

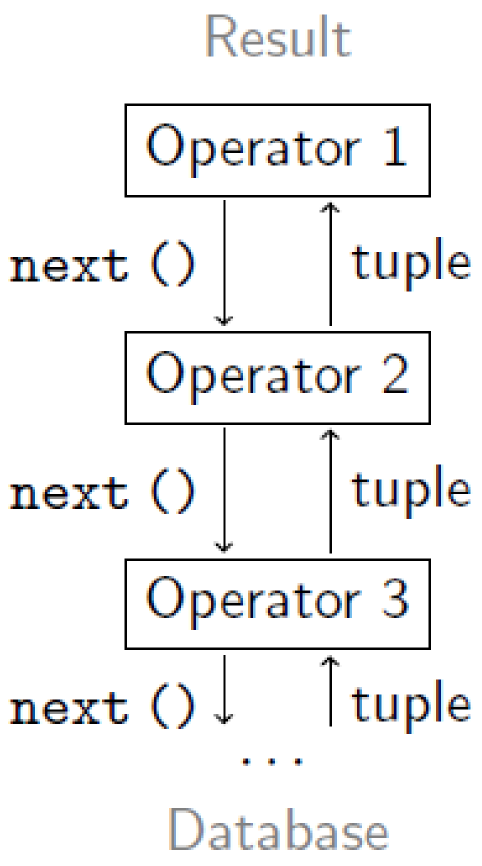
Tuple-at-a-time pipelining:

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 by obtaining input data by calling the next() method.

Tuple-At-A-Time Processing

Most systems implement the Volcano iterator model:

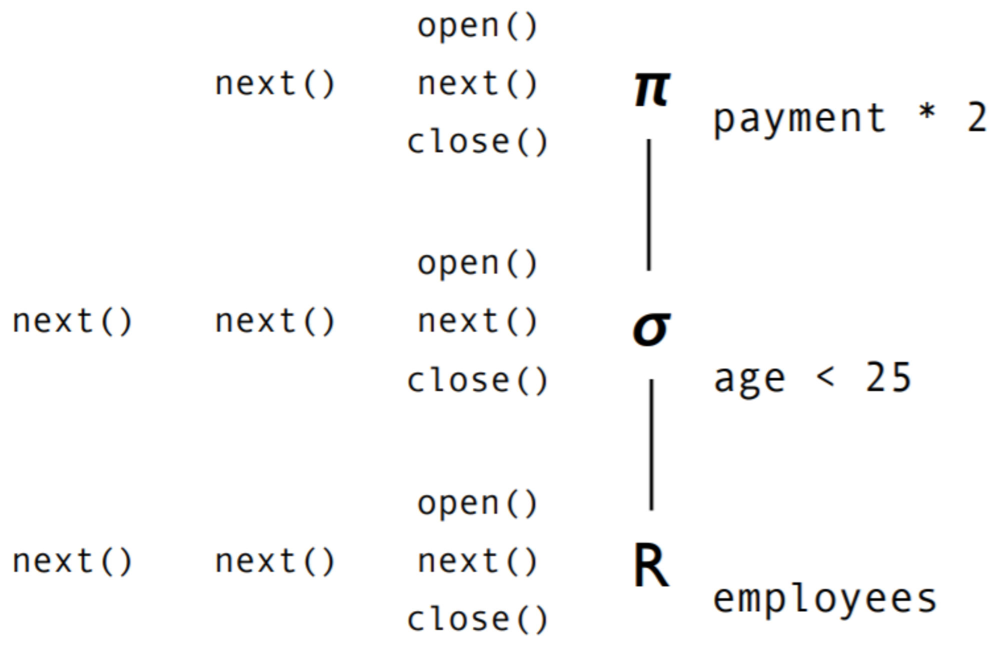
- Operators request tuples from their input using next ().
- Data is processed tuple at a time.
- Each operator keeps its own state.



Tuple-At-A-Time Processing

Each database operator (relational algebra) implements a common interface:

Open()	Next()	Close()
Reset internal state and prepare to deliver first result tuple.	Deliver next result tuple or indicate EOF.	Release internal data structures, locks, etc.



Tuple-At-A-Time Processing - Limitations

Pipeline-parallelism

- Data processing can start although data does not fully reside in main memory
- Small intermediate results

All operators in a plan run tightly enclosed.

- Their combined instruction footprint may be large.
- Instruction cache failures.

Operators constantly call each other’s functionality.

- Large function call overhead.

The combined state may be too large to fit into caches.

- Data cache failures.

Full materialization

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.

Tuple-At-A-Time VS Full materialization

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.

Summary

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.

Vectorized execution

“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.

Vectorized execution

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.

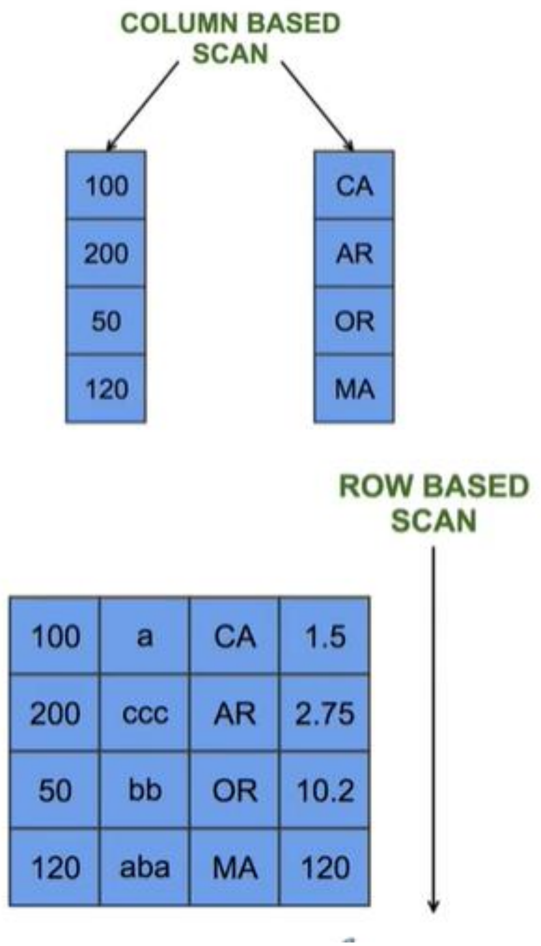
Thus, vectorized execution combines pipelining (avoidance of materialization of large intermediates) with the array-loops code patterns that make MonetDB fast.

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.

Vectorized execution

Vectorized Execution

- Traditional:
  - Tuple at a time iterator based model.
  - Push tuple-at-a-time through query plan tree
- Vectorized:
  - Columnar processing.
  - A vector containing fixed number of values from a column.
  - Improves the overall utilization of both CPU-Memory and disk I/O bandwidth.
  - Better utilization of CPU cache by passing blocks (vectors) of tuples between operators.
  - Cache lines are potentially filled with data of interest.
  - Efficient tight for-loop access.



Advantages of vectorized processing

- Reduced interpretation overhead. The amount of function calls performed by the query interpreter goes down when compared to the tuple-at-a-time model.
- Better cache locality. VectorWise tunes the vector size such that all vectors needed for evaluating a query together comfortably fit in the CPU cache.
- Compiler optimization opportunities. Vectorized primitives are responsive to some of the most productive compiler optimizations, and typically also trigger compilers to generate SIMD instructions.

Advantages of vectorized processing

- Block algorithms. 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.
- Profiling. Since vectorized implementations of relational operators perform all expression evaluation work in a vectorized fashion,

Conclusion

- vectorized query execution system can support both vertical (column) and horizontal (record) tuple representations in a single execution framework
- sequential operators (project, selection) work best on vertical vectors (exploiting automatic memory prefetching and SIMD opportunities),
- random access operator (hash-join or -aggregation) work best using blocks of horizontal records, due to cache locality.

Compression

- Intuitively, data stored in columns is more compressible than data stored in rows.
- Compression reduces the size of data on disk by compressing the SSTable.
- 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

Example: customers (name, phone, e-mail,address).

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.

Compressing one column-at-a-time

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.

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.

Exploiting extra CPU cycles

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).

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

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

Some of the “heavierweight” compression schemes that optimize for compression ratio 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 control the SIMD instruction set.

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

Summary

This session will give the knowledge about

- Vectorized Processing
- Compression