chatwithengineers

A Survey of Query Execution Engines (from Volcano to Vectorized Processing)

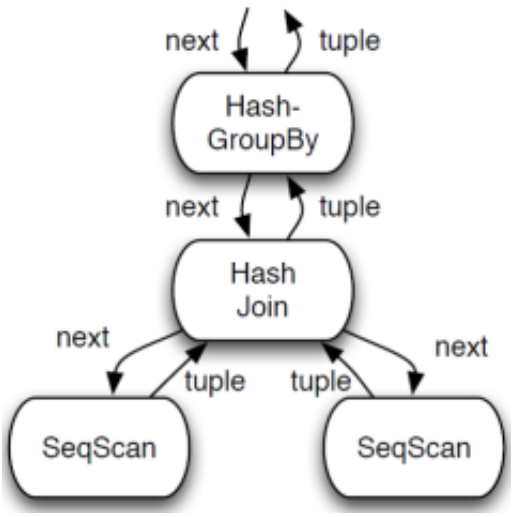
Open source analytical processing systems such as Drill, Hive, Impala, Presto, Spark, and Tajo have recently gained popularity. These projects introduce themselves with various cutting-edge techniques that are not easy to understand at a glance. One of the most popular techniques is vectorized query processing. In this article, I'm going to give you comprehensive understanding about it. Rather than focusing on only vectorized query processing, I'll explain various query execution engines for better understanding around query processing. In detail, I'll present what query execution engine is, various processing models, and some examples of open source implementations. I hope this article would be helpful to understand even more analytical systems in Big Data world.

Query Execution Engine and Different Processing Models

Query execution engine (shortly called execution engine) in an analytical system is a core component that actually evaluates data and results in query answers. SQL engines like Hive, Impala as well as even general purpose engines like Spark commonly have execution engines. An execution engine dominantly affects the performance of data processing. For better performance, various models of the design and implementation have been

introduced in database community. There are mainly three or four execution engine models. Most execution engines follow Volcano-style engine model [14].

In Volcano-style engine model, an execution engine takes an execution plan (also called evaluation plan), generates a tree of physical operators, evaluates one or more tables through physical operators, and finally results in a query answer. A physical operator is an evaluation algorithm that implements an operator of relational algebra (or logical plan). Physical operators are implemented though Volcano-style iterator interface, in which each operator implements an API consisting of *open()*, *next()*, and *close()* methods. *open()* initializes external resources such as memory and file open, and *close()* releases them. As shown in Figure 1, the *next()* call of each physical operator in an operator tree produces one new tuple recursively obtained from *next()* calls of child physical operators in the tree. This iteration interface is common in all models that I'll describe later.



<Figure 1. A tree of physical operators and Volcano-style iteration>

Volcano-style (Tuple-at-a-time) Processing

Let me talk about the Volcano-style engine model in more detail. It was firstly

introduced in Volcano system [14]. It is the most common execution engine model in real-world DBMSs, such as MySQL, SQLite, and earlier versions of Hive.

As I mentioned earlier, each *next()* in a physical operator recursively invokes next() of child operators, and then it finally produces a single tuple. It's why this model is also called *"tuple-at-a-time processing model"*. While this model is very intuitive and easy to implement, it is not CPU friendly in terms of processing. There are many drawbacks to cause performance degradation [2, 13]. Here, I'm going to explain few significant points.

```
Latency Comparison Numbers
--------------------------
execute typical instruction                 1   ns   1/1,000,000,000 sec
L1 cache reference                        0.5 ns
Branch mispredict                           5   ns
L2 cache reference                          7   ns   14x L1 cache
Mutex lock/unlock                          25   ns
Main memory reference                     100   ns   20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy            3,000   ns
Send 1K bytes over 1 Gbps network      10,000   ns
Read 4K randomly from SSD*            150,000   ns   ~1GB/sec SSD
Read 1 MB sequentially from memory    250,000   ns
Round trip within same datacenter     500,000   ns
Read 1 MB sequentially from SSD*    1,000,000   ns   1 ms  ~1GB/sec SSD, 4X memory
Disk seek                          10,000,000   ns   10 ms  20x datacenter roundtrip
Read 1 MB sequentially from disk   20,000,000   ns   20 ms  80x memory, 20X SSD
Send packet CA->Netherlands->CA   150,000,000   ns   150 ms

Notes
-----
1 ns = 10^-9 seconds
1 us = 10^-6 seconds = 1,000 ns
1 ms = 10^-3 seconds = 1,000 us = 1,000,000 ns
```

latency_comparison_numbers.txt hosted with ❤ by GitHub view raw

<Figure 2. Latency Comparison Numbers [9]>

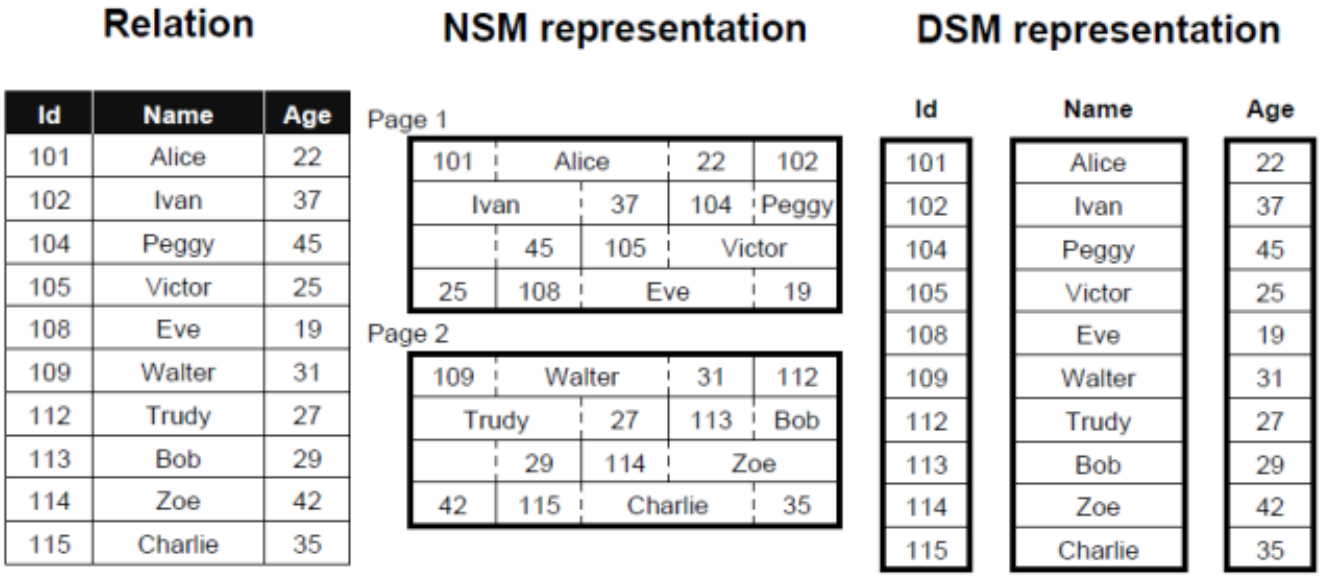
Firstly, *next()* is implemented via so-called virtual function call. In compilers, a virtual function call involves the lookup of a function pointer in the vtable, requiring additional CPU instructions. Also, a virtual function call is an indirect jump, usually causing a branch misprediction. A branch misprediction can cause many bubbles in CPU pipelines. This branch misprediction penalty is not trivial as shown in Figure 2. In Intel Skylake processors, this penalty is at least 16-17 cycles [5]. It significantly decreases instructions per CPU-cycle (IPC). Note that operators recursively call *next()* of child operators in order to return a single result tuple. Thus, a large function call overhead occurs for each tuple. The similar overhead also occurs when each expression in select list is evaluated. These overhead is called *interpretation overhead*. Figure 3 shows a typical aggregation query. Think about how many virtual functions are called only for a single tuple. According to [12], only 10% of the time for TPC-H Q1 in MySQL using Volcano-style iteration spent on actual evaluation.

```
select
    l_returnflag,  l_linestatus, sum(l_quantity) as sum_qty,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
    sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
  avg(l_quantity) as avg_qty,
    avg(l_extendedprice) as avg_price, avg(l_discount) as avg_disc,
  count(*) as count_order
from
    lineitem
where
    l_shipdate <= '1998-09-01’
group by
    l_returnflag,
    l_linestatus
order by
    l_returnflag,
    l_linestatus
```

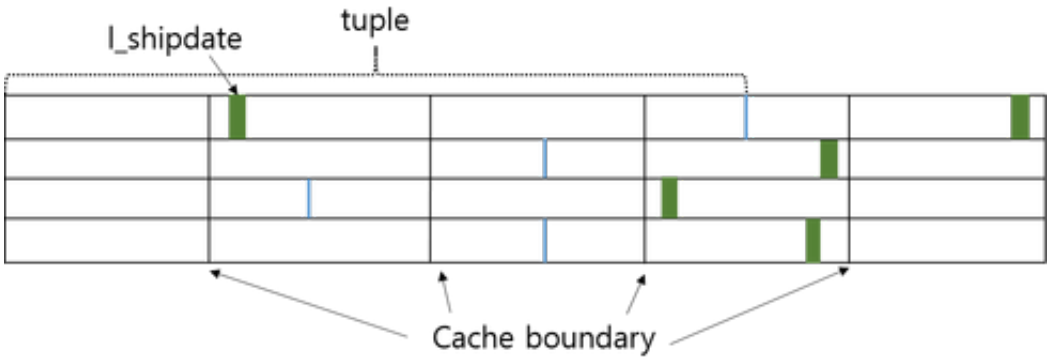
tpch_q1.sql hosted with ❤ by GitHub view raw

<Figure 3. TPC-H Q1>

Second, accessing a row causes many cache misses. Volcano-style processing uses row-wise tuple representation called N-ary storage model (NSM). In this model, a tuple is stored as a sequence of bytes in a memory space as shown Figure 4. Each operator mainly reads some fields in each row. As shown in Figure 5, in order to load *l_shipdate* field in a row CPU will load more bytes that begin at the cache boundary into a cache, but most of the cached data will be irreverent. Subsequent accesses to fields will cause frequent cache misses, leading to the reference of memory. CPU will be stall while waiting for fetching one cache line from memory. As shown in Figure 2, a main memory reference takes roughly 100 ns, which may be a time for CPU to execute hundreds of instructions. Consider how much CPU performance would be degraded significantly if multiple cache misses occur to produce a single tuple.



<Figure 4. Relational table and its row representations in NSM and DSM (source: [2])>



< Figure 5. Memory representation of rows>

Third, all routines of a tree of physical operators run tightly interleaved. The footprint of combined instructions is likely to be large to not fit in L1 instruction cache. It can frequently cause instruction cache misses. For example, in Figure 2 *HashGroupby*, *HashJoin*, and two *SeqScan* will run sequentially for each *next()* call. In real world queries, tens of operators of a single tree are usual. The latest Intel CPU Skylake just has 32KB L1 instruction cache per core [6]. Also, each operator maintains various states, such as cursors, hash tables, and partially aggregated values. For *next()* call, the execution engine should access most states of a physical operator tree. It can frequently cause data cache misses. As I mentioned earlier, both cache misses cause significant performance degradation.

Block-oriented Processing

Block-oriented processing [6] is based on the volcano-style iteration, but *next()* call returns a list of (100 – 1000) rows instead of a single tuple. Block-oriented processing amortizes the cost of virtual function calls by using tight loops in each operator. Thus, block-oriented processing reduces much of the interpretation overhead and increases instruction and data cache hit ratio than that of volcano-style processing. As I mentioned earlier, however, accessing fields in a row can still cause frequent data cache misses as shown in Figure 5. Block-oriented processing also has limitations for compilers to

exploit modern CPU features like SIMD.

Column-at-a-time Processing

Column-at-a-time processing [10] is an early model for columnar execution. Column-at-a-time processing is based on block-oriented processing, but it uses vertically decomposed storage model (DSM) [1], where columns are dense-packed arrays. Figure 4 shows an example of rows represented in DSM. Besides, *next()* in each operator processes a full table and store intermediate data in a column-at-a-time fashion.

```
1  int uselect_bt_void_int_bat_int_const(oid *output, int *input, int value, int size) {
2    oid i;
3    int j = 0;
4    for (i = 0; i < size; i++) {
5      if (input[i] > value) {
6        output[j++] = i;
7      }
8    }
9    return j;
10 }
```

uselect_bt_void_int_bat_int_const.c hosted with ❤ by GitHub view raw

< Figure 6. Comparison primitive between an integer column and a constant value (source [2])>

In this model, the building block operators are called *primitives*. They are simple functions that process arrays of values in a tight loop. This approach gives compilers more optimization opportunities to exploit modern CPU features. First of all, code footprints of loops can be easily fit into L1 instruction cache. Eliminated function call, loop unrolling (see Figure 7), less control and data hazards can make branches even more predictable and maximize the depth of CPU pipelining. Auto-vectorization by compilers can

use *single instruction multiple data* (SIMD) instructions, exploiting data-level parallelism in modern CPUs. Consequently, this model achieves higher CPU efficiency.

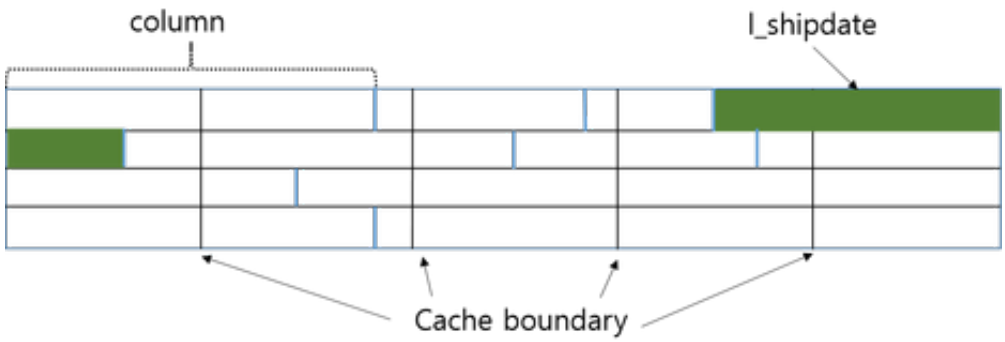
```
for (i = 0; i < 128; i++) {
  a[i] = b[i] + c[i];
}
```

```
// unrolled loop

for (i = 0; i < 128; i += 4){
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
  a[i+2] = b[i+2] + c[i+2];
  a[i+3] = b[i+3] + c[i+3];
}
```

<Figure 7. An example of loop unrolling – a loop can be optimized to be an unrolled loop.>

See an example primitive of column-at-a-time model shown in Figure 6. *uselect_bt_void_int_bat_int_const* takes 4 parameters, where *output* is an array of output values, *input* is an input column, *value* is a constant value for comparison, and *size* is the number of input column values. This primitive performs a simple tight loop to check if *i* th column value is bigger than a given constant value.



<Figure 8. Memory representation of rows in DSM>

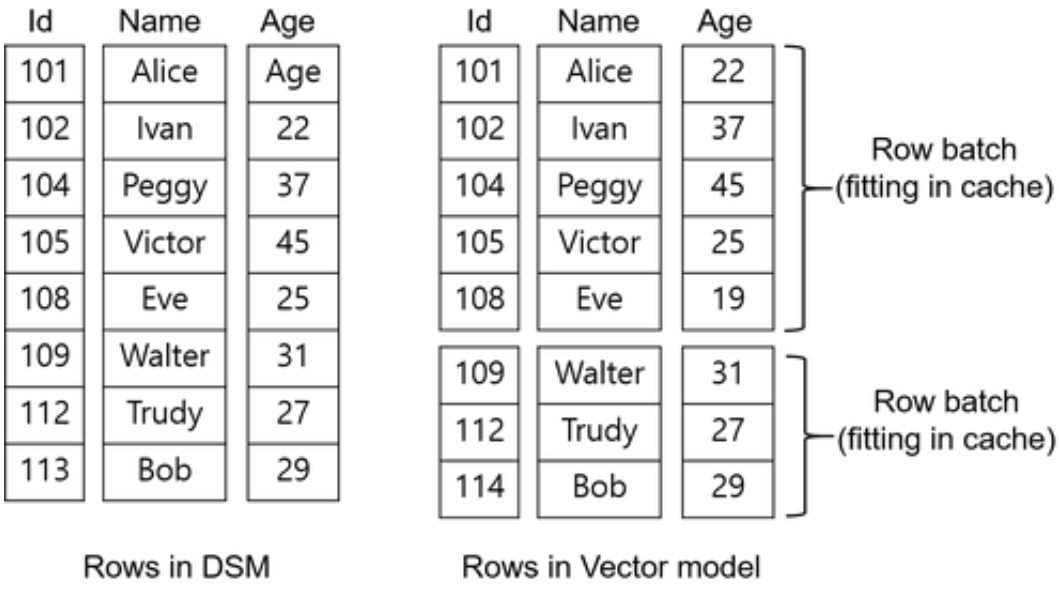
Unlike the previous two models using row representation in NSM, the column-at-a-time processing uses DSM where values of each column are packed in a dense array. The access to values of rows is direct. As shown in

Figure 6 and 8, a primitive consumes and produces arrays of values with a good cache locality. Thus, this model significantly improves CPU efficiency even more than the previous two models do.

However, column-at-a-time processing model has a drawback. Since this model processes a full table in a column-at-a-time, it can lead to a large amount of intermediate data to be materialized in memory, on disk, or both, causing data cache misses and I/O overheads.

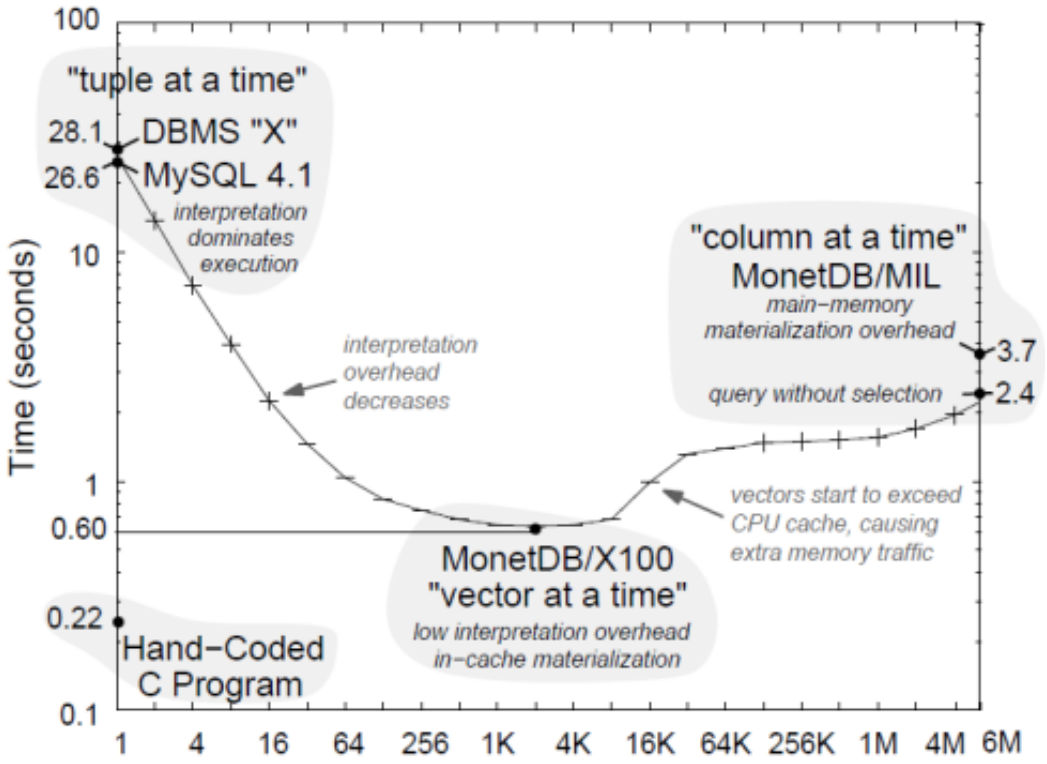
Vectorized Query Processing

Vectorized query processing [12] is currently one of the most advanced columnar execution models. It was designed to solve the drawbacks of column-at-a-time processing [14]. Most parts of vectorized query processing are similar to those of column-at-time-time processing, but for each *next()* vectorized query processing model processes column chunks called vectors (see Figure 9) instead of columns of a full table. Vectors contain a batch of rows that are small enough to fit in CPU cache. Thus, each operator on vectors can do in-cache processing. Consequently, vectorized query processing dramatically reduces data cache misses while keeping the advantages of block-oriented and column-at-a-time processing models.



<Figure 9. Row representations in DSM and Vector models>

Figure 10 shows TPC-H Q1 performance in a vectorized query processing engine (MonetDB/X100) with varying vector sizes. This figure shows how vector size affects cache performance. When the vector size is 1 and 6M, it works like volcano-style and column-at-a-time engines respectively. According to the figure, about 4k is the best vector size.



<Figure 10. TPC-H Q1 performance in MonetDB/X100 with varying vector sizes (source [12])>

Like column-at-a-time model, this model also uses simple functions called *primitives* that process column vectors in a tight loop. They usually perform arithmetic calculations, comparison operations, and logical operations. There are mainly two kinds of primitives: *map* and *select*. A map primitive is just a function transforming one or more input vectors into an output vector. A *select* primitive is a filter function, generating a selection vector according to a predicate (i.e., a boolean condition).

```
map_add_long_int_col_col(int vec_num, long [] result, long [] col1,
                         int [] col2, int [] sel_vec) {
  if (sel_vec == null) {
     for (int i = 0; i = 0; i < vec_num; i++) {
        result[i] = col1[i] + col2[i];
     }
  } else {
    int sel_idx;
    for (int i = 0; I = 0; i < vec_num; i++) {
        sel_idx = sel_vec[i];
        result[sel_idx] = col1[sel_idx] + col2[sel_idx];
    }
  }
}
```

map_add_long_int_col_col.java hosted with ❤ by GitHub view raw

<Figure 11. *add* primitive for long and integer vectors>

```
int sel_le_long_long_col_col(int vec_num, int [] res_sel_vec,
                             long [] col1, long [] col2, int [] sel_vec) {
  int ret = 0;
  if (sel_vec == null) {
    for (int i = 0; i < vec_num; i++) {
      if (col1[i] < col2[i]) {
        res_sel_vec[ret++] = i
      }
```

```
    }
  } else {
     for (int i = 0; i < vec_num; i++) {
      int sel_id = sel_vec[i];
      if (col1[sel_id] < col2[sel_id]) {
        res_sel_vec[ret++] = sel_id;
      }
  }
  return ret;
}
```

sel_le_long_long_col_col.java hosted with ❤ by GitHub view raw

<Figure 12. *select* primitive evaluating a boolean condition in long and int vectors>

Figure 11 shows an *add* primitive that adds corresponding row values of the two input vectors *col1 and col2* and writes the sum of them into a *result* vector. A selection vector *sel_vec* is an optional parameter, and it can come from other selection vectors. A selection vector is an array containing offsets of the relevant rows. If *sel_vec* is given, primitives evaluate only rows corresponding to rows positions of the selection vector. If *sel_vec* is *null*, primitives evaluate all rows. Also, *vec_num* contains a vector size. This is a typical interface of primitives in the vectorized query processing.

Figure 12 shows a select primitive that evaluates a boolean condition on the two input vectors and writes relevant row positions into an output selection vector (*res_sel_vec*). Like *add* primitive, select primitives also can process input vectors with or without an input selection vector. If a select primitive takes a selection vector, it works as if the primitive evaluates a boolean condition as well as ANDing together. This is because with a selection vector, a select primitive evaluates only relevant rows already selected from other select primitives.

Implementations in Open Source Projects

So far, I discussed different query execution engine models. Now, let's take a look at some real examples for the models.

Apache Impala is a distributed SQL engine on Hadoop. Impala was firstly introduced by Cloudera in 2008. As it was recently designed, Impala reflected the recent studies of the database community. It chose the block-iteration model and Just-in-time (JIT) query compilation using LLVM. With regard to their design choice, Impala looks similar to [4], but Impala uses push-based tuple iteration rather than pull-based iteration. You can find iteration interface and row block implementation at Impala source code as shown Figure 13 and 14. These source files also contain valuable comments to help us to figure out a real example of the block-oriented model. I recommend you read them.

```
class ExecNode {
 ...
  virtual Status Init(const TPlanNode& tnode, RuntimeState* state);
  virtual Status Prepare(RuntimeState* state);
  virtual Status Open(RuntimeState* state);
  virtual Status GetNext(RuntimeState* state, RowBatch* row_batch, bool* eos) = 0;
  virtual void Close(RuntimeState* state);
 ...
}
```

exec-node.h hosted with ❤ by GitHub view raw

<Figure 13. An execution Iteration interface in Impala>

```
class RowBatch {
  ...
  TupleRow* ALWAYS_INLINE GetRow(int row_idx);
  ...
}

```

```
class Iterator {
  public:
    Iterator(RowBatch* parent, int row_idx, int limit = -1) :
    TupleRow* IR_ALWAYS_INLINE Get();
    TupleRow* IR_ALWAYS_INLINE Next();
  ...
}

```

row-batch.h hosted with ❤ by GitHub view raw

<Figure 14. Row block implementation in Impala>

As I mentioned earlier, the block iteration model has some disadvantages such as data cache misses and less optimization opportunities by modern compilers, but Impala choose JIT query compilation techniques to mitigate them. At runtime, Impala generates codes for expressions, hash function, and others by using LLVM JIT. This approach can eliminate many branches causing misprediction and use intermediate data kept in CPU registers [4], gaining performance benefits. Cloudera also provides how Impala's code generation improves the performance [8]. It's worth to read it.

Apache Hive is a data warehouse solution in Hadoop. Before Stinger project, Hive used the tuple-at-a-time model. Since Hive adopted the vectorized query processing in 2014, Hive has provided two execution engines based on both query processing models respectively. Users can set a specific configuration to choose one of both engines. Hive has a highly abstract iteration interface. As you can see in Figure 15, *Operator* class is a typical Volcano-style iteration interface, and *process()*corresponds *next()* of Volcano-style model.

```
/**
 * Base operator implementation.
 **/
public abstract class Operator<T extends OperatorDesc> implements ... {
  public void initialize(Configuration hconf, ...) ...
```

```
6
7     public abstract void processOp(Object row, int tag) throws ...;
8
9     public void close(boolean abort) throws HiveException { ... }
10  }
11
12
```

Operator.java hosted with ❤ by GitHub view raw

<Figure 15. Iterator interface in Hive>

Interestingly, this single interface supports both tuple-at-a-time and vectorized query processing. Depending on the engine mode, the first argument *Object* can be a row or *VectorizedRowBatch*, which contains a selection vector and a batch of rows organized with each column as a vector. As shown Figure 17, *ColumnVector* is an abstract implementation of a single vector, and it has concrete subclasses for various data types. Each concrete class uses a specific type array to represent a value vector.

```
1   public class VectorizedRowBatch implements Writable {
2     public int numCols;           // number of columns
3     public ColumnVector[] cols;   // a vector for each column
4     public int size;              // number of rows that qualify (i.e. haven't been filter
5     public int[] selected;        // array of positions of selected values
6     public int[] projectedColumns;
7     public int projectionSize;
8
9     ...
10  }
```

VectorizedRowBatch.java hosted with ❤ by GitHub view raw

<Figure 16. Vectorized rows in Hive>

```
1   public abstract class ColumnVector {
2      public boolean[] isNull;
3      public boolean noNulls;
```

```
    public boolean isRepeating;
    ...
}

public class LongColumnVector extends ColumnVector {
  public long[] vector;
  public static final long NULL_VALUE = 1;
  ...
}
```

ColumnVector.java hosted with ❤ by GitHub view raw

<Figure 17. Column vector interface and long vector implementation in Hive>

Note that Hive is implemented in Java that does not provide SIMD primitives. Its vectorized query processing engine is not complete. Java's superword optimization (SIMD) works implicitly when a sequence of routines satisfies with some specific conditions (e.g., aligned memory, direct memory access, and simple operation). In other words, developers cannot manually control using SIMD instructions, and they just let some routine satisfy with the conditions to trigger the superword optimization. Currently, Hive community is working on more superword optimization opportunities [7].

Columnar Storage vs. Columnar Execution

Lastly, I'd like to point out one more thing. In my experience, people are often confused into thinking that columnar storage and columnar execution are the same thing. Furthermore, people misunderstand that some system uses a columnar execution engine if it supports columnar file formats, such as Parquet and ORC. Actually, that's not the case. Columnar storage is about how data are organized in a columnar fashion for I/O efficiency. Columnar execution mainly focuses on efficient computation. This confusion may be because many literatures assume that columnar executions implicitly work on columnar storages. In literatures, that assumption is reasonable because in

database systems storage structures like block or pages are directly used as in-memory row structures. However, it is different in Hadoop ecosystem. In Hadoop ecosystem, analytical systems use their own in-memory row structures, and file formats (e.g., Avro, SequenceFile, Parquet, and ORC) are independently standardized. Thus, the systems read and serialize the storage representation into their own in-memory row structures. That is main difference between analytic systems in Hadoop ecosystem and studies in literatures.

Conclusion

I presented different query execution engine models and their pros and cons. After you read this article, it would be more fun to understand existing analytic processing engines. Besides, if you want to learn columnar execution and storage, I recommend reading Column-Oriented Database Systems (VLDB 2009 Tutorial). It would be helpful to figure out the overall studies and find good papers worth reading further.

References

[1] A decomposition storage model, ACM SIGMOD Conf., 1985.
[2] Balancing Vectorized Query Execution with Bandwidth-Optimized Storage
[3] Block Oriented Processing of Relational Database Operations in Modern Computer Architectures, IEEE ICDE Conf., 2001.
[4] Efficiently Compiling Efficient Query Plans for Modern Hardware, VLDB Conf., 2011.
[5] http://www.7-cpu.com/cpu/Skylake.html
[6] https://en.wikipedia.org/wiki/Skylake_(microarchitecture)
[7] https://issues.apache.org/jira/browse/HIVE-10179
[8] Inside Cloudera Impala: Runtime Code Generation

[9] Latency Numbers Every Programmer Should Know
[10] Monet: A Next-generation DBMS Kernel for Query-Intensive Applications
[11] MonetDB/X100: A DBMS In The CPU Cache, IEEE Data Engineering Bulletin, 2005.
[12] MonetDB/X100: Hyper-Pipelining Query Execution, CIDR 2005.
[13] The Design and Implementation of Modern Column-Oriented Database Systems
[14] Volcano-An Extensible and Parallel Query Evaluation System, IEEE TKDE, 1994.

Author: Hyunsik Choi
ASF Member / Data Engineer View all posts by Hyunsik Choi

Hyunsik Choi / August 29, 2016 / big data, columnar execution, database, execution engine, query processing, vectorized processing