VectorH: Taking SQL-on-Hadoop to the Next Level

Andrei Costea[‡] Adrian Ionescu[‡] Bogdan Răducanu[‡] Michał Świtakowski[‡] Cristian Bârcă[‡]
Juliusz Sompolski[‡] Alicja Łuszczak[‡] Michał Szafrański[‡]
Giel de Nijs[‡] Actian Corp.[‡]
Peter Boncz[§] CWI[§]

ABSTRACT

Actian Vector in Hadoop (VectorH for short) is a new SQL-on-Hadoop system built on top of the fast Vectorwise analytical database system. VectorH achieves fault tolerance and storage scalability by relying on HDFS, and extends the state-of-the-art in SQL-on-Hadoop systems by instrumenting the HDFS replication policy to optimize read locality. VectorH integrates with YARN for workload management, achieving a high degree of elasticity. Even though HDFS is an append-only filesystem, and VectorH supports (update-averse) ordered tables, trickle updates are possible thanks to Positional Delta Trees (PDTs), a differential update structure that can be queried efficiently. We describe the changes made to single-server Vectorwise to turn it into a Hadoop-based MPP system, encompassing workload management, parallel query optimization and execution, HDFS storage, transaction processing and Spark integration. We evaluate VectorH against HAWQ, Impala, SparkSQL and Hive, showing orders of magnitude better performance.

1. INTRODUCTION

Hadoop, originally an open-source copy of MapReduce, has become the standard software layer for Big Data clusters. Though MapReduce is losing market share as the programming framework of choice to an ever-expanding range of competing options and meta-frameworks, most prominently Spark [26], Hadoop has established itself as the software base for a variety of technologies, thanks to its evolution which decoupled the YARN resource manager from MapReduce and the wide adoption of its distributed file system HDFS, which has become the de-facto standard for cheap scalable storage for on-premise clusters. *SQL-on-Hadoop systems* allow to connect existing SQL-based business applications with the results of "big data" pipelines, adding to their worth, further accelerating the adoption of Hadoop in commercial settings. This paper describes **VectorH**, a new SQL-on-Hadoop system with advanced query execution, updatability, YARN, HDFS and Spark integration.

SIGMOD '16, June 26–July 1, 2016, San Francisco, California, USA.

ISBN 978-1-4503-1247-9/12/05…$15.00

DOI: http://dx.doi.org/10.1145/2882903.2903742

SQL-on-Hadoop=MPP DBMS? A SQL-on-Hadoop system shares certain characteristics with an analytical parallel DBMS, such as Teradata. First of all, these so-called MPP systems are used in *analytical* workloads, which consist of relatively few very heavy queries – compared to transactional workloads – that perform heavy scans, joins, aggregations and analytical SQL, such as SQL'2003 window functions based on PARTITION BY, ROLL UP and GROUPING SETS. That said, these workloads are by no means read-only, and systems must also sustain a continuous stream of voluminous updates under limited concurrency.

In the past decade, analytical database systems have seen the rise of (i) *columnar stores* – which reduce the amount of IO and memory bandwidth needed in query processing further boosted by (ii) integrated *data compression* and (iii) facilities that allow *data skipping* in many queries, e.g. through lightweight index structures like MinMax indexes that exploit natural orders and correlations in the data. Such modern analytical database systems typically (iv) use a query engine that is (re)designed to exploit modern hardware strengths. Two approaches for this are popular. Vectorwise [27] developed the first fully *vectorized* query engine where all operations on data are performed on vectors of values, rather than tuple-at-a-time, reducing query interpretation overhead, increasing data and code CPU cache locality and allowing to use SIMD instructions. The competing approach is to use Just-In-Time (JIT) compilation, such that a SQL query is translated into assembly instructions – see Impala [24].

A commonality between SQL-on-Hadoop and MPP systems is that both run on a cluster of nodes; both systems must distribute data in some way over this cluster and when a single query is run it must ideally parallelize over all nodes – and all their cores – in this cluster. While this shared goal is obvious, it requires a lot of technical excellence in data placement, query optimization and execution to achieve near-linear parallel scalability on complex SQL queries. Both kinds of system, finally, must be prepared for nodes or links between them to go down (un)expectedly, and thus replicate data and take measures to achieve fault tolerance.

SQL-on-Hadoop≠MPP DBMS. There are also significant differences between SQL-on-Hadoop and MPP systems. MPP systems typically run on a dedicated cluster, and sometimes even on specific hardware (database machines), whereas SQL-on-Hadoop systems share a Hadoop cluster with other workloads. The task of dividing the hardware resources between concurrent parallel queries is already hard in MPP, but when other Hadoop jobs also take cores, memory, network and disk bandwidth, this becomes even harder.

In order to counter this problem, all members of the Hadoop ecosystem should coordinate their resource usage through a resource manager such as YARN.

By basing data storage on HDFS, SQL-on-Hadoop systems leverage scalability and fault-tolerance for a large part from their Hadoop infrastructure; whereas MPP systems must introduce specific and proprietary mechanisms for fault tolerance (e.g. for adding and removing hardware, reacting gracefully to failures, monitoring status). MPP systems are typically more expensive, not only in upfront software and hardware cost, but also in terms of administration cost; the required expertise is more rare than that for maintaining a Hadoop cluster, which has become a standard in this space.

In some analysis scenarios, raw data first needs to be processed with methods and algorithms that cannot be expressed in SQL, but better in a programming language. MPP systems have introduced proprietary UDF (User Defined Function) interfaces with differing degrees of expressive power in an attempt to address this, whereas in SQL-on-Hadoop it is more natural to use parallel programming frameworks like Spark for such tasks. The advantage of this approach is that Spark is more powerful and flexible than any UDF API, and is open, so re-use in creating UDFs is more likely. SQL-on-Hadoop systems thus must fit seamlessly into "big data" pipelines, either through integrated APIs or through an ability to read and write large datasets in a number of standard formats (text, binary, ORC, Parquet) stored via HDFS on the cluster.

VectorH Features and Contributions. VectorH is a SQL-on-Hadoop system whose prime contribution is (i) to provide vastly superior query processing *performance*. This higher performance is based on a combination of a truly vectorized query engine, compressed columnar data formats with better lightweight compression methods than those used in common Hadoop formats, and better data skipping statistics than these common formats. Its SQL technology is mature, which is especially visible in the query plan quality produced by its cost-based query optimizer, but is also observed in its robust SQL support including analytical SQL, encryption, authentication, and user role management.

A second system-level contribution is that VectorH (ii) instruments the HDFS block placement policy to determine where its compressed columnar blocks are replicated – by HDFS default on 3 nodes. VectorH uses this locality to always guarantee local IO, even if the cluster composition changes, e.g. due to hardware failures. This tight *integration with HDFS* is a novelty in SQL-on-Hadoop systems.

VectorH also pioneers (iii) *elasticity* features in Hadoop, by tightly integrating with YARN; it is able to scale up and down its memory usage and core footprint, in order to accommodate fluctuating workloads needs on a Hadoop cluster that is shared with many other users and job types.

VectorH further (iv) allows *fine-grained updates* (inserts, deletes, modifications) to its columnar compressed tables by leveraging the Vectorwise's Positional Delta Tree [12] data structure, without updates affecting the high performance of read queries – which still work on the latest up-to-date state. Integrating its column storage formats and PDTs in the append-only HDFS environment is another system-level contribution of the system. Finally, the *Spark integration* of VectorH allows it to read all common Hadoop formats with full HDFS locality and also enables running user code in parallel on VectorH-resident data efficiently.

2. FROM VECTORWISE TO VECTORH

In this section we first summarize the most relevant features of the Vectorwise system, the performance-leading[1] single-server **Actian Vector** product. Then, we provide a short overview of how this technology was used to architect the new shared-nothing **Actian VectorH** SQL-on-Hadoop product, which also serves as roadmap for the paper.

Relevant Vectorwise Features. Vectorized processing performs any operation on data at the granularity of *vectors* (mini-columns) of roughly 1000 elements. This dramatically reduces the overhead typically present in row-based query engines that interpret query plans tuple-at-a-time [4]. Additionally, instruction cache locality increases as the interpreter stays longer in each function, and memory bandwidth consumption is reduced as all vectors reside in the CPU data cache. Vectorized execution exposes possibilities to leverage features of modern CPUs like SIMD instructions in database workloads and increases analytical query performance by an order of magnitude. The vectorized execution model has been adopted by analytical subsystems such as IBM BLU [20] and SQLserver column indexes [17]; and it is also being deployed in new releases of Hive [13].

Vectorwise introduced the compression schemes PFOR, PFOR-DELTA and PDICT, designed to achieve good compression ratios even with skewed frequency value distributions, as well as high decompression speed [28]. PDICT is dictionary compression, PFOR is zero prefix compression when representing values as the difference from a block-dependent base (the Frame Of Reference). PFOR-DELTA compresses deltas between subsequent tuples using PFOR, and has been adopted by the Lucene system to store its inverted text index. These schemes represent values as thin fixed-bitwidth codes packed together, but store infrequent values uncompressed as "exceptions" later in the block. The letter P in these stands for "Patched", since decompression first inflates all values from the compressed codes; then in a second phase patches the values that were exceptions by hopping over the decompressed codes starting from the first exception, treating these codes as "next" pointers. Thus, the data-dependent part of decompression is separated in a short second phase while most work (code inflation) is SIMD-friendly and branch-free. These decompression formats were recently re-implemented in Vectorwise in AVX2, yielding a function that decompresses 64 or 128 consecutive values in a typically less than half a CPU cycle per value. The Hadoop formats ORC and Parquet, while clear improvements over predecessor formats such as sequence and RCfile, have not been designed for branch-free SIMD decompression and are much slower to parse [25].

In data warehouses, fact table order is often time-related, so date-time columns typically are correlated with tuple-order. Vectorwise automatically keeps so-called *MinMax* statistics on all columns. These MinMax indexes store simple metadata about the values in a given range of records, and allow quick elimination of ranges of records during scan operations (skipping), saving both IO and CPU decompression cost. Both ORC and Parquet adopted this idea, but Parquet often cannot avoid IO as it stores MinMax information at a block position that can only be determined while parsing the header – forcing the block to be read [25].

[1] See www.tpc.org/downloaded_result_files/tpch_results.txt

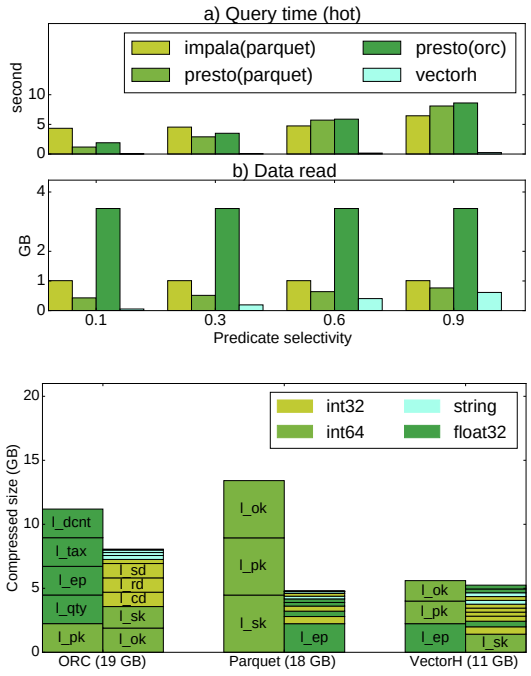


Figure 1: Data Format Micro-Benchmarks.

Storage Format Micro-Benchmarks. We performed micro-benchmarks on the SF=100 TPCH `lineitem` table that was sorted on the `l_shipdate` column, to corroborate our belief that the Vectorwise data storage is more efficient than ORC and Parquet implementations. The *topmost* chart of Figure 1 shows that Vectorwise is faster on: `SELECT max(l_linenumber) FROM lineitem WHERE l_shipdate<X` queries, where `X` is varied to test different selection percentages. Such queries can profit strongly from MinMax skipping, given the table order on `l_shipdate`. While the time differences can be partly explained by the higher efficiency of the query processing engine of VectorH, and specifically its vectorized decompression (whereas the Parquet and ORC readers decode value-at-a-time), the *middle* chart shows that it is also touching much less data. Impala does not do MinMax skipping at all while Presto does, but it reads more data than needed[2] (for example, at 90% selectivity data read is even bigger than the total size of the columns). Parquet/ORC split row groups by row count, so compressible columns may be split into too small chunks [14]. In contrast, VectorH's format allows these thin columns to occupy full 512KB blocks. For example, one VectorH block might contain the whole column while with Parquet/ORC this column could be split into thousands of small pieces depending on the compressability of the other columns.

A small part of the reduced data reading is in fact not due to better skipping, but because the PFOR schemes used in VectorH compress better than both the ORC and Parquet formats do, as shown by the *lowest* chart[3]. Vectorwise data is almost twice smaller, though Parquet could be close were it not for its inefficient handling of 64-bits integers.

Vectorwise Physical Design Options. Vectorwise tables can be stored unordered or as a "clustered index". Rather than creating some kind of B-tree, this means that Vectorwise stores the table sorted on the index key. It also provides unclustered indexes (i.e. real index trees), which can help queries that access a few tuples to avoid a table scan. Vectorwise supports horizontal table partitioning by hash, which can be combined with indexing.

The main benefit of a clustered index is data skipping in scans for queries with range-predicates on the index keys. When a clustered index is declared on a foreign key, treatment is special as the tuple order then gets derived from that of the referenced table, making the tables *co-ordered*. This allows merge-joins between them, and is very similar on the physical storage level to nested Hadoop data formats such as ORC and Parquet (where, say, an `order` object may contain nested `lineitem`s). In the columnar layout used by ORC and Parquet, this means a table consists of columns with the same tuple order, and nested tables are co-ordered and merge-joinable using an also co-ordered "repeat-count" column. Such a tight, co-ordered, layout is difficult to insert/delete in, a problem solved in Vectorwise with "PDTs".

Compressed, columnar, ordered, data formats are read-optimized and direct modifications on these would be costly. Vectorwise allows high-performance updates nevertheless, using a differential update mechanism based on *Positional Delta Trees* (PDT) [11]. Positional Delta Trees are *counting* B+-trees that store updates (insert, delete, modify) in their leaves, where the interior nodes count the amount of inserts minus the amount of deletes in the sub-tree below them. PDTs can quickly translate back-and-forth between the old position of a tuple (the SID - stable ID) and its current position (RID). PDTs allow to update ordered tables (i.e. clustered indexes) as well a co-ordered tables (clustered index on a foreign key) with logarithmic complexity [11]. Their primary goal is *fast merging* of differences in a scan, which happens for each and every query. Merging in differences from a PDT is fast because it identifies tuples by position, rather than by primary key, which saves both key-comparisons (CPU) and scan work on the key (IO).

VectorH & Paper Roadmap. VectorH is a YARN-based cluster version of Vectorwise relying on HDFS for storage and fault-tolerance. HDFS is a global cluster filesystem where all nodes can read all data, but in order to achieve good performance, one must read local data mostly. The VectorH integration with HDFS is discussed in Section 3.

To determine on which nodes VectorH can run, given data locality but also Hadoop resource availability, it integrates with YARN, as discussed in Section 4.

The foundation of shared-nothing parallel Vectorwise was laid by a MSc project [7]. It created a MPP system consisting of Vectorwise processes, adding a MPI-based parallel query processing infrastructure, "exchange" operators, and a parallel rewriter. This prototype was significantly extended in the VectorH product as described in Section 5.

HDFS is an append-only filesystem, but this restriction does not hinder Vectorwise's PDT update structure. Section 6 describes transaction handling in VectorH, including distributed logging and 2PC. Section 7 outlines the VectorH-Spark integration that allows users to run code near their data in parallel, but also to read and write data in Hadoop data formats with high performance. VectorH is evaluated against Impala, Hive, HAWQ and SparkSQL on a 1000GB TPC-H database in Section 8. We then discuss related work in Section 9 before concluding in Section 10.

[2]Profile information showed that the Presto ORC reader does not skip IO, but it does skip decompression CPU work.
[3]We omitted `l_comment` from the comparison as it is not compressible with lightweight schemes.

3. STORAGE AFFINITY WITH HDFS

The storage layer of Vectorwise was modified in VectorH to use the `libhdfs` JNI library instead of a normal filesystem, for its persistent table storage, and its write-ahead log (WAL). HDFS files are split into fixed size blocks (typically 128MB to 1GB) which are automatically replicated across the datanodes in the cluster (by default at R=3 locations) to provide fault-tolerance. VectorH may optionally also use HDFS for temporary storage spilling operators though then it overrides the replication degree to just 1. Vectorwise has a *predictive buffer manager* that not only handles immediate page requests but tries to keep concurrent scan queries busy by prefetching [23]. Whereas on standard filesystems Vectorwise uses asynchronous direct IO, for HDFS a thread pool was added to perform parallel synchronous HDFS block reads. HDFS provides an efficient way to read local data called *short-circuit* reads, through which files can be read directly from disk, bypassing the datanode. This provides reasonable performance (say 30% overhead compared to direct IO). VectorH in general achieves the situation that all table IOs are short-circuited. Its IO scheduler uses HDFS *block location tracking* to decide which thread makes each IO request, with the goal of keeping all local disks busy.

Original Layout. In Vectorwise, the unit of table storage is a block of fixed compressed size (default 512KB). For IO efficiency, writing is done in groups of consecutive blocks (by default 8 groups per block). This makes the effective IO unit bigger (4MB by default) which is better suited for rotational disks. Blocks are stored consecutively in files, where by default one file is used for each column in a table partition. Each block has a number which determines its position in a file (e.g. block 0 will be at position 0, block 1 at position 512KB). In memory, a column is a list of blocks, not necessarily with the same ordering as in the file. This is because the file space of freed blocks can be reused.

Blocks can be freed as a result of updates (delete, drop column, etc.) and subsequently reused. However, this requires writing in the middle of a file, which is not supported by HDFS. In the original layout, space could therefore only freed by rewriting the table fully elsewhere and deleting its files – either triggered manually, or automatically in case when differential updates in PDTs exceed a RAM threshold and *update propagation* is triggered. For clustered (ordered) tables, all updates (delete, insert, modify) go to PDTs, while for unordered tables inserts are handled as appends. Thus, workloads consisting of consecutive deletes (which are compact in the PDT) and inserts can lead to significant wasted disk space. A second problem is that appending to a table requires opening many files. For example, a table with 100 columns and 10 partitions, at a replication degree R=3, requires 3000 open files.

File-per-partition Layout. It is noteworthy that Hadoop formats such as ORC and Parquet store data of all columns in the same file; their column orientation is to be understood as PAX [1] with a huge block size. For instance, ORC uses 1GB chunks that represent a table slice, in which different segments store the various columns – OLAP queries only need to read these files partially and skip over the column segments that are not needed. Even though the HDFS block size is 128MB or larger, (short-circuit) reads occur on the actual granularity of the IO, i.e. more fine-grained.

To better suit HDFS, a *file-per-partition* VectorH layout

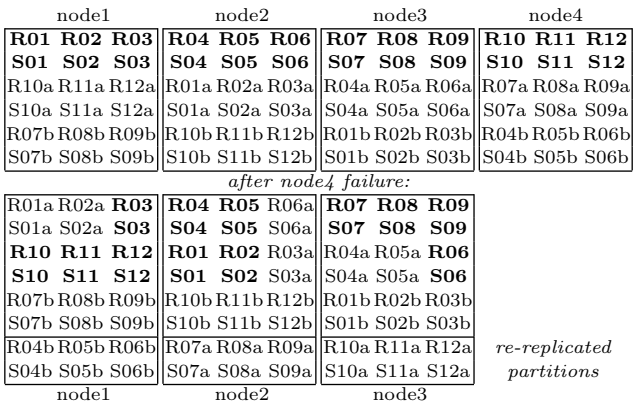


Figure 2: Partition Affinity Mapping for the 12 partitions of table R,S before (top) & after (bottom) node4 failure. Responsible partitions in bold; a/b are the second/third copy (R=3).

was introduced: all columns of a table partition are stored in the same file. So, for example, a table with 100 columns and 10 partitions, at a replication degree R=3 leads to 30 HDFS files.

To reclaim unused space VectorH data files are split horizontally, into fixed-size chunks consisting of 1024 blocks, similar to Parquet RowGroups. Each block chunk is stored in a separate HDFS file. Only one block chunk file is open for writing at a time. Block chunks make it possible to write in the middle of a table partition, although only at block chunk boundaries. Thus it becomes possible to free space by deleting a block chunk file when all (or most) of the blocks in it are unused. At the end of an append, though, the last blocks are typically only partially filled. Since blocks are written to files at fixed offsets, these partially filled blocks would use as much space as full blocks. As a solution to this, partial blocks are written to a partial chunk file. A subsequent append merges these blocks with new data into new blocks and then frees the previous partial chunk file.

Instrumenting HDFS Replication. HDFS by default replicates data in R=3 locations, and the default replication policy is that the first copy is stored on the datanode that issues the write, and the two others are determined by the namenode – if the cluster is large and topology information is present, one replica may be in the same rack and the third in a different rack. The HDFS replication policy determines this placement *per file*, hence all the blocks in one file get spread over the same R=3 datanodes. While all SQL-on-Hadoop systems so far have left block placement to HDFS, VectorH specifically influences this.

In VectorH each table partition at any time has one *responsible node*. This assignment is currently semi-static and is changed when a node fails or when the administrator changes the set of nodes across which VectorH is deployed. VectorH strives for a responsibility assignment where all data files of a partition are local to the HDFS datanode which is responsible for it. VectorH tries to achieve this by issuing appends only from the responsible datanode, as by default the first copy ends up with the writer. However, if this datanode goes down or for some other reason is no longer available to VectorH, we have to access the second or third copy of these chunk files, and given the default HDFS replication policy these could be anywhere. Therefore, node

failures or changes in the node set used by a SQL-on-Hadoop system will degrade data affinity whenever using the default policy.

To get control over data placement, VectorH instruments the HDFS replication policy for its own files. HDFS allows to register a `BlockPlacementPolicy` class for this, whose method `chooseTarget()` gets a filename as a parameter and delivers a list of datanode names where replicas should be stored. It is called when a client initiates a file append, but also during HDFS re-replication and re-balancing initiated in the background by the namenode. The current VectorH replication policy initially establishes the *affinity* between partitions and nodes by assigning table partitions to all datanodes running VectorH in such a way that each partition gets stored evenly at R different datanodes. Each such partition assignment means that all chunk files of the partition are stored at that particular datanode. Figure 2 shows a *partition affinity mapping* example of tables R and S with 12 partitions (R04 in bold is the primary copy and R04a and R04b its two replicas). As can be seen, the initial affinity mapping performed at table creation forms a round-robin pattern – we discuss node failures in the next section.

Another benefit of explicit locality control is the co-location of table partitions that refer to each other over a declared foreign key. This can be done when the partitioning key of one table includes a foreign key, and the partitioning key of the other table includes the primary key that it refers to, and both tables have the same amount of partitions. In such cases, the foreign key join between these two tables can be performed by just joining the matching partitions, and if these matching partitions happen to be stored on the same HDFS datanodes, no network communication is needed. Indeed, in Figure 2 one can see that matching partitions of tables R,S are always co-located such that the `WHERE R.key = S.key` join in SQL queries can be executed by joining the corresponding partitions on their responsible node.

4. ELASTICITY WITH YARN

Workers and Master. VectorH consists of a set of N Vectorwise processes, called the *worker set* that run on the datanodes of a Hadoop cluster. The VectorH administrator maintains a *list of viable machines* in a configuration file to identify these – this may just be a subset of the whole Hadoop cluster so one can use a set of machines which have the same hardware, as this leads to better parallel query load balancing. Before actually starting these processes, VectorH needs to negotiate with YARN's resource manager where to run, and it ends up using a subset of these machines as its worker set. One of the workers becomes the *session-master*, however this node does not possess any specific data structures, so this role can be easily interchanged. The session master has a coordinating role in transactions, and performs parallel query optimization which includes running a query *scheduler* that performs DBMS workload management, determining the amount of cores and memory it can use. Currently, this is done using a policy where table partition scans are performed at their responsible nodes, and the available amount of cores is divided equally among them.

Out-of-band YARN. The YARN resource manager (RM) administers RAM and CPU resources on a Hadoop cluster, for this purpose a NodeManager (NM) daemon runs on each datanode. Clients communicates with the RM, to start a first process as their ApplicationMaster (AM). Typically, the AM starts further containers (processes) on other nodes by communicating with the NMs, following negotiated resource demands in terms of minimum and maximum (desired) amounts of RAM and cores. YARN can use different scheduling policies, namely the `FairScheduler` and `CapacityScheduler` where the latter assigns resources to multiple queues having different priorities. Newly arriving high-priority jobs may cause running jobs to be *pre-empted* by YARN, first by asking their AMs to decrease resource usage and after a timeout by killing their containers.

While YARN is a great step forward for Hadoop and native applications, SQL-on-Hadoop systems still struggle to fit into its concepts because they are usually long-running processes. Equating YARN job submission to query submission, is not viable because a DBMS needs to run a query on threads in the already active server processes, rather than spawn a separate new process for each query, and furthermore the latency of the YARN protocols is high and would eclipse interactive query running times. Also, even if a Vectorwise process would run in a YARN container, the problem would be that YARN does not allow yet to modify the resources of a container. Stopping and re-starting a Vectorwise container just to e.g. increase its RAM budget would interrupt the running application for tens of seconds and lose the contents of its buffer pool to much performance detriments.

Therefore, VectorH runs processes *out-of-band*, i.e. separate from its containers. The actual containers are dummies that sleep almost constantly, monitoring once in a while the co-located VectorH processes to ping back their live status to YARN and implicitly to VectorH, i.e. VectorH runs a dbAgent process that acts as its YARN client. Instead of running a single container on each datanode with all its RAM and core resources, it typically runs multiple AMs with corresponding containers on the worker set, each allocating a slice of its resources. Hereby, it can gradually increase and decrease its resource usage (start or stop new AMs/slices) over time. YARN pre-emption can even kill containers in which case dbAgent notices this event and then instructs the session master to adjusts its own resource usage accordingly, by changing the configuration of the DBMS workload management (e.g. making queries use less cores and memory, possibly leading to spilling operators).

Min-cost Flow Network Algorithms. When VectorH initially starts, or when it recovers from node failures[4] its dbAgent must select those N machines with most data locality (stored blocks), out of all nodes which are on the viable machine list and which have enough free memory and core resources. To get cluster resource information, dbAgent asks YARN for the *cluster node reports*, and for finding VectorH table partition locality it queries the HDFS Namenode to retrieve *block locations* for the corresponding chunk files. It may happen that there are no such nodes available, in which case the worker set shrinks, which is also the case in the below part of Figure 2.

After establishing the worker set, dbAgent checks which partitions are already local to them. Not all partitions may

[4]Typical VectorH deployments are maximally a few co-located racks, so node failures are rare and recovery is kept simple: the system silently restarts, though users will notice at most a minute of non-responsiveness plus transaction aborts.

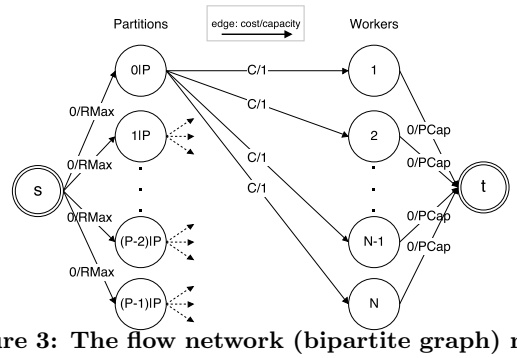


Figure 3: The flow network (bipartite graph) model used to determine the responsibility assignment.

be represented R times already, so the question is which partitions should go where – the resulting *partition affinity map* steers the behavior of the instrumented HDFS `BlockPlacementPolicy` class (and hence re-replication). Figure 2 shows an outcome where partitions 10-12 that were on node4 are re-replicated to node3, 7-9 to node2 and 4-6 to node1.

This mapping can be computed by solving a min-cost matching problem on a network flow where the left side represents the table partitions and the right side the workers, as illustrated in Figure 3 (a related approach was proposed in [15]). From the source (s) to each partition, we create edges with cost 0 and capacity equal to the HDFS replication degree (RMax=R), and we create edges from each partition to each worker with capacity 1 and cost C=0 for local partitions, otherwise C=1. We then create edges from the workers to the destination node (t), assigning a cost 0 and a capacity equal to the amount of partitions each node should store (PCap = #partitions/N).

A final step for dbAgent is to compute the responsibility assignment: which worker is responsible for each partition (i.e. which partitions should be in bold in Figure 2). To assign responsibilities to worker nodes, it runs the same min-cost flow algorithm, with the only difference that the edges from source (s) to each partition have cost 0 and capacity 1. The lower part of Figure 2 shows an example result of this, were node1 becomes responsible for partitions 3,10-12; node2 for 1-2,4-5 and node3 for 6-9 (in bold).

More details about the min-cost algorithms used for (i) worker set selection, (ii) creating the data affinity mapping, (iii) responsibility assignment, and even (iv) run-time query resource scheduling can be found in [5].

Dynamic Resource Management. While VectorH runs, the amount of cores and memory it uses in the query scheduler may change over time – though the worker set currently must stay the same. At startup, VectorH will negotiate with YARN to get to its configured *target* of resources, if this is not possible, it will start nevertheless as long as it gets above a configured minimum. If there are higher-priority jobs, YARN will pre-empt VectorH and reduce its resources, but VectorH will periodically negotiate with YARN to go back to its target resource footprint. Using the *automatic footprint* option, VectorH can also self-regulate its desired core/memory footprint depending on the query workload. In a future release, we intend to also allow to grow and shrink the worker set (not only cores/RAM) dynamically.

In Figure 2 one can see that after the failure of node4, and assuming these nodes have 24 cores, a maximal resource query plan can run on 3x24 cores processing a join query between R and S[5]. If in a future release VectorH would support dynamic shrinking of the worker set, in an idle workload VectorH could shrink to a minimum resource footprint of $\lceil \#nworkers/R \rceil$ and *still* have all table partitions local at the active workers. In the scenario of Figure 2 we could imagine that the partition responsibility assignment is changed such that node1 becomes responsible for all its partitions (all in bold), and node2 and 3 for none of theirs (all a/b versions), as they become inactive workers. A minimal-resource query plan could just use *one thread* at node1 – 72 times fewer resources than the maximal plan.

5. PARALLELISM WITH MPI

The shared-nothing parallelism in VectorH builds on the infrastructure for multi-core parallelism in Vectorwise [2], which is based on *Exchange* (`Xchg`) operators [10]. An `Xchg` operator does not modify the data that streams in and out of it, but only redistributes these streams. It thus encapsulates parallelism, allowing other query operators (scan, select, join, aggregation) to be unaware of it. The different `Xchg` operator flavors are characterized by specific constraints on the numbers of producer/consumer streams and the type of data redistribution performed. For example, `XchgHashSplit` hash-partitions the data from n producer streams to m consumer streams, while `XchgUnion` has a single consumer and there is no data partitioning involved. We also implement `XchgMergeUnion`, `XchgBroadcast` and `XchgRangeSplit`. In our system a stream corresponds to a separate thread that executes some part of an operator pipeline. Consequently, an `Xchg` operator acts as a synchronization point among a number of producer and consumer threads.

This "Volcano" model with `Xchg` operators was adopted in the 1990s by relational DBMS products, which in those days added multi-CPU parallel capabilities to their existing designs, allowing all existing query processing operators to be used as-is. In recent years, new single-server multi-core systems have been presented [20, 19] that have advocated an approach where all relational operators are redesigned to be explicitly aware of parallelism. Ingredients to such designs are fine-grained shared-memory work-queues, NUMA affinity, and lock-avoidance with atomic instructions or -elision with hardware transactional memory. However, when considering MPP parallelism, these approaches do not apply and `Xchg` is still a solid basis for MPP parallelism.

Distributed Exchange. Figure 4 depicts the implementation of a generic Distributed Exchange (`DXchg`) operator with two producer nodes (`n1` and `n2`) and two consumer nodes (`n3` and `n4`). On `n4` there are two threads. For network communication we use MPI (Message Passing Interface), which is extensively used in the high performance computing community and offers a rich and well-defined API for collective and point-to-point communication. It provides a good balance between efficiency (low latency, high throughput), portability across various network fabrics found in Hadoop clusters (e.g. Ethernet, Infiniband) and ease of development and maintenance. Specifically, we use the Intel MPI library, which provides good multi-threading support and in our experience outperforms all alternatives, particularly

[5]Even though each node has 4 primary matching join partitions, these joins can be parallelized 6-way each, over multiple cores to keep all 24 threads busy.

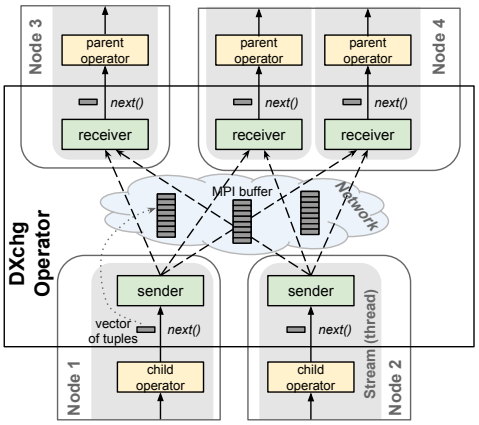


Figure 4: DXchg Operator Implementation

on high-speed Infiniband networks (where MPI provides a native binding, avoiding slow TCP/IP emulation).

`DXchg` producers send messages of a fixed size (>= 256KB for good MPI throughput) to consumers and use double-buffering, such that communication overlaps with processing (i.e. partitioning + serialization). Tuples are serialized into MPI message buffers in a PAX-like layout, such that Receivers can return vectors directly out of these buffers with minimal processing and no extra copying.

It is often the case that the sets of producer and consumer nodes are overlapping, or even the same. As an optimization, for intra-node communication we only send pointers to sender-side buffers in order to avoid the full `memcpy()` that MPI would otherwise need to do behind the scenes.

For the two partitioning `DXchg` variants, `DXchgHashSplit` (DXHS) and `DXchgRangeSplit` (DXRS), the original implementations, described in [7], used a *thread-to-thread* approach. Each `DXchg` sender partitions its own data with a fanout that is equal to the total number of receiver threads across all nodes. For senders to operate independently, they each require $fanout$ private buffers. Typically, the nodes in the cluster are homogeneous (same num_cores). Hence, the partitioning fanout equates to $num\_nodes * num\_cores$, and considering double-buffering and the fact that each node has num_cores `DXchg` threads active, this results in a per-node memory requirement of $2 * num\_nodes * num\_cores^2$. As VectorH gets deployed on larger systems with typically tens and sometimes more than 100 nodes, each having often around 20 cores, partitioning into 100*20=2000 buffers becomes more expensive due to TLB misses. More urgently, this could lead to buffering many GBs ($2 * 100 * 20^2 * 256$KB = 20GB). Not only is this excessive, but the fact that the buffers are only sent when full or when the input is exhausted (the latter being likely in case of 20GB buffer space per node, leading also to small MPI message sizes) can make `DXchg` a materializing operator, reducing parallelism.

Therefore, we implemented a *thread-to-node* approach, where rather than sending MPI messages to remote threads directly, these get sent to a node, reducing the fanout to num_nodes and the buffering requirements to $2*num\_nodes*num\_cores$. The sender includes a one-byte column that identifies the receiving thread for each tuple. The exchange receiver lets consumer threads selectively consume data from incoming buffers using the one-byte-column. The *thread-to-node* `DXchg` is more scalable than the default implementation, yet on low core counts and small clusters the *thread-to-thread* implementation is still used as it has a small performance advantage there.

Query Optimization. Currently it is the session-master which parallelizes incoming queries. In the future, it will be relatively easy to distribute this work over all worker nodes at least for read-queries, but given that updates need some form of central coordination, and the fact that analytical workloads have only a limited query throughput, we settled for this centralized approach for now. Most query optimization changes were made in the Parallel Rewriter that already existed in Vectorwise for single-server query parallelization [2]. It performs cost-based optimization using a dynamic programming algorithm, and a simple cost model based on the cardinality estimates taken from the serial plan. A *state* in this search space is a tuple consisting of a physical operator label x, some structural properties sp and a current level of parallelism pl: $s = (x, sp, pl)$. For each such state, we define as $c(s)$ the best (estimated) cost of a parallel query plan for the sub-tree rooted at x that satisfies the structural data properties sp such that exactly pl[6] streams consume the output of x. The structural properties used in VectorH are partitioning (guarantees that two tuples with the same value on a set of keys will be output by the same stream) and sorting (guarantees that tuples from each stream are sorted on some other keys). The aim is thus to find $c(s = (root, \emptyset, 1))$ where $root$ is the full query tree.

The Parallel Rewriter contains a collection of rewrite rules that transform a state in a different state. We give three examples for *MergeJoin*. The first adds an *XchgHashSplit* on top and from a state $s_1 = (mj, partitioned[keys], pl)$ recurses into $s_1' = (mj, \emptyset, pl)$, relaxing the requirement for partitioning since *XchgHashSplit* (on $[keys]$) takes care of it. The second adds an *XchgUnion*[pl] when at a state $s_2 = (mj, \emptyset, 1)$ and therefore recurses into $s_2' = (mj, \emptyset, pl)$. A final transformation just recurses into mj's children from $s_3 = (mj, \emptyset, 1)$ into $s_3' = (child_1, sorted[keys], 1)$ and $s_3'' = (child_2, sorted[keys], 1)$. More details on the dynamic programming algorithm to traverse this search space is in [2].

The initial MPP prototype of VectorH [7] modified the Parallel Rewriter to (i) make pl a list (instead of an integer) that contains the current parallelism level per node, (ii) adapting transformations to introduce *distributed* `Dxchg` rather than local `Xchg` operators and (iii) adapting the cost model for these operators. This was sufficient since the initial prototype assumed the presence of a global distributed file system without table partitioning, where everybody can read everything at equal IO cost. This was a simplifying assumption only, and is invalid in HDFS even though that *is* a global distributed filesystem, since tables have affinities to datanodes. Moreover, with the introduction of hash-partitioned tables in VectorH there appeared a number of new optimization opportunities, e.g. certain group by/join queries on the partition key can be executed without `Dxchg` operations. The Parallel Rewriter aims to avoid communication at all cost, and appropriately adds a high cost for `Dxchg` operators in its model. Finally, with the addition of trickle updates on partitioned tables, ensuring data locality for update operators becomes a requirement rather than an optimization opportunity as will be discussed in Section 6.

Detecting locality. We will illustrate some of the VectorH-

[6]This number is always smaller than the maximum level of parallelism granted to a query by the scheduler.

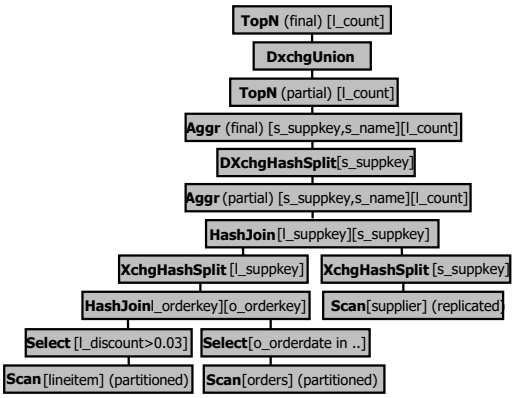


Figure 5: Example Distributed Query Plan

specific transformations through a query on the TPC-H schema, on tables `lineitem` partitioned by `l_orderkey`, `orders` partitioned by `o_orderkey` and `supplier` non-partitioned (i.e. replicated):

```
SELECT FIRST 10 s_suppkey, s_name, count(*) as l_count
FROM   lineitem, orders, supplier
WHERE l_orderkey=o_orderkey AND l_suppkey=s_suppkey AND
      l_discount>0.03 AND
      o_orderdate BETWEEN '1995-03-05' AND '1997-03-05'
GROUP BY s_suppkey, s_name
ORDER BY l_count
```

The distributed plan that VectorH generates is shown in Figure 5. The only network communication is high up in the query tree (the `DXchg` operator), namely splitting *s_suppkey*s that match all the predicates and communicating partial results before selecting the top 10 suppliers. The Parallel Rewriter finds this plan using transformations like:

- **local join**: the opportunity to `HashJoin` the matching partitions of `lineitem` and `orders` is detected by tracking the origins the join keys, detecting that the tables are partitioned on the join key.
- **replicate build side of a join**: in the `HashJoin` with `supplier`, we detect that the build side of the join operates entirely on replicated tables, and thus splits both sides of the join only *locally* (between threads on the same node). A variant of this would be to forgo splitting and build a shared hash table – the decision between these is made based on cost.
- **partial aggregation**: before the `DXchgHashSplit` required by the `Aggr` on `s_suppkey` (and `s_name` but Vectorwise knows it is functionally determined), it is often better to aggregate locally first. Rather than letting each thread aggregate its stream locally, on medium-cardinality aggregations, inserting a local `Xchg` has the effect of bringing together more duplicate keys, and thus further reducing the amount of data that needs to be sent. This strategy is detected here, but VectorH also detects that a `XchgHashSplit` does not need to be inserted below the `Aggr` as the stream is already partitioned on `s_suppkey` at that point.

With all three rules applied, on a 6-node cluster with Infiniband 40Gbit/s for TPC-H SF-500 VectorH runs the example query in 5.02s. Without partial aggregation this is 5.64s; and without replication in the build 5.67s. The biggest effect is from local joins: without this, performance deteriorates to 25.51s; whereas without any of these rewrites it is 26.14s.

Some of the changes that were made to accommodate these new transformations augmented the state structure by (i) adding a *replicated* boolean field to mark that the entire sub-tree rooted at the state's operator will be replicated on all nodes, by (ii)extending the partitioning structural property to include also a mapping of each partition to streams/nodes[7] and (iii) marking whether the partitioning property is partial or not.

6. TRANSACTIONS IN HADOOP

Vectorwise transaction management is described in [12], and consists of the following elements:

- as described in Section 3, updates (insert,modify,delete) are administered as differences, stored in a Positional Delta Tree (PDT) that is kept for each table (partition) in RAM. Not all updates go to PDTs, large inserts to unordered tables are appended directly on disk.
- PDTs can be stacked (differences on differences on .. on a table). Isolation is provided by sharing among all queries a slow-moving large *Read-PDT* that contains direct differences to a persistent table and stacked on that a smaller *Write-PDT* that contains differences to the Read-PDT. When it starts, a transaction creates an empty private *Trans-PDT* stacked on top it all.
- while a transaction runs, it gathers changes w.r.t. the transaction start in the Trans-PDT. This provides *snapshot isolation*, since commits by concurrent transactions make a copy-on-write of the master Write-PDT. If such a situation occurs, the old master Write-PDT is freed as soon as all transactions referencing it finish.
- when a transaction commits, its Trans-PDT is *serialized* to the global database state, which may have advanced in the meantime. PDT serialization not only transforms the Trans-PDT contents but implements *optimistic concurrency control* as it will detect write-write conflicts at the tuple granularity. Such conflicts, if present, force the transaction to abort.
- during commit, which happens atomically (globally locked), the serialized Trans-PDT is written into a Write Ahead Log (WAL) for persistence, and its changes are *propagated* into the master Write-PDT; this makes the system reach the new database state that new incoming transactions will see. The changes from Write-PDT are propagated to the Read-PDT when the size of the Write-PDT reaches a certain threshold.

Vectorwise uses a single, global, WAL that is also used to log other meta-data, such as DDL commands, the creation and deletion of blocks in column files, and changes to the contents of MinMax indexes.

Distributed Transactions in VectorH. We now summarize how VectorH builds on this for distributed transaction management. Since the PDTs store all committed updates in RAM and grow over time, holding them in memory on all nodes would not scale. Rather than a single WAL, VectorH uses table partition-specific WALs, and only the node that is *responsible* for a table partition (see Sections 3 and 4) reads

[7]Was needed to ensure correctness in situations where responsibilities change over time or where the number of threads allocated to a query does not match the number of table partitions. This also has the added benefit of being flexible in concurrency scenarios, i.e. #partitions is not strongly tied to parallelism level

this WAL at startup, keeps the respective PDTs in memory, and writes to it as part of the commit sequence. Update queries get a distributed query plan that ensures that each table partition is updated at its responsible node, and hence modifies PDTs on the right node, or if applicable, performs HDFS appends to the right datanode.

VectorH introduces 2PC (2-Phase Commit) to ensure ACID properties for distributed transactions, where a much-reduced global WAL is written to by the session-master, who coordinates transaction processing. The fact that HDFS is a distributed filesystem, and any worker can read (and potentially write) this global WAL means that the role of session-master can be taken over by any other worker in case of session-master failure.

Log Shipping. Not all VectorH tables are partitioned: typically, small tables will not use partitioning and these can be scanned and cached in the buffer pool by all workers (therefore we consider them as "replicated"). All workers need to read their PDTs from their WALs on startup (replicated PDTs) and keep them in RAM in order to ensure that table scans on these see the latest image. The commit protocol was extended with *log-shipping* for these replicated tables, such that all changes to them are broadcast during transaction processing to all workers, who then can apply the changes (i.e., to the replicated PDTs). We could alternatively have forced a query plan that executes updates on replicated tables to replicate this work on all nodes. However, log-shipping consumes less resources and, in the future, we plan to allow nodes that have a table partition HDFS-local but are *not* responsible for it, to also participate in query processing – for increased parallelism and better load-balancing. For this functionality, the responsible node will have to ship its log actions to the (R-1) nodes that also store the table-partition. Hence, log-shipping is useful. The log actions sent over the network use the same format as in the on-disk transaction log, and are applied in a manner similar to the replay of transaction log on startup, allowing reuse of existing code and the testing infrastructure.

Update Propagation. The in-memory PDT structures cannot grow forever. Vectorwise implements a background process called *update propagation* that is responsible for flushing PDTs to the compressed column store. Update propagation is triggered based on the size of PDTs as well as the fraction of tuples that reside in memory. The latter policy cares for high performance of table scans, which are affected by merging of the on-disk data with PDT data. The internal storage of PDTs is column-wise, as this improves memory access locality and saves bandwidth in table scans. We found that in practice inserts account for most of the PDT volume. To make update propagation more efficient, VectorH introduces an algorithm that is able to separate tail inserts from other types of updates. It also provides a query option whereby inserts to unordered tables, which normally would be direct appends, get buffered as PDT inserts, as for very small inserts this provides better performance (no IO). In a future release, this decision should become automatic. Deletes are stored efficiently in PDTs, especially for contiguous ranges of deleted tuples, while modifies often concern only one column. Flushing tail inserts only creates new data blocks and does not modify existing ones. Other kinds of updates require re-compression of existing blocks and can be flushed with a lower frequency. In future releases we will exploit the horizontally fragmented table partitions, in block chunk files (see Section 3); making the decision for each chunk-file to either (i) re-write it in a new file with the PDT changes applied and delete the old one or (ii) to avoid rewriting it if there are few updates on it – moving these updates to the new PDTs, rather than fully emptying them.

Referential Integrity. The ability to perform concurrent updates is provided for schemas with an enforced unique key or foreign key constraint as well. It must not be allowed for two transactions to insert the same key if an unique key is defined. If the table is partitioned and the partition key is a subset of the unique key, VectorH verifies such constraints by performing node-local verification only. Foreign key constraints also prohibit inserting keys that do not exist in the referenced table or have been concurrently deleted from the referenced table. If one of the involved tables is non-partitioned (i.e. replicated), typically the smaller referenced side, all nodes have a replica of committed PDTs and it is possible to find possible conflicts if all nodes execute the verification. If both tables are partitioned, and co-located at their responsible nodes, VectorH again performs quick node-local verification. The default policy for integrity constraints that cannot be checked without communication is to reject concurrent updates – though this policy may be refined in a future release. However, this policy covers all typical cases, as it is the most natural to partition data on the primary key and the foreign key referencing it.

MinMax Indexes. MinMax indexes are small table summaries kept for each table partition; it conceptually divides a table in a limited number of large tuple ranges, where for each range it keeps the Min and Max value of each column. Whereas ORC and Parquet store this information inside their file format, VectorH stores this information separately as part of the WAL. The rationale is that MinMax information is intended to help prevent data accesses, therefore it is better to store it separately from that data. Because on VectorH only the responsible nodes have this information, yet during query optimization this information is consulted (e.g. by the Parallel Rewriter), a MPI network interface was developed to allow MinMax indexes to be consulted remotely. This interface was engineered such that it is possible to resolve all MinMax information needed for an entire query (which may involve multiple selection predicates on multiple tables) in a single network interaction. MinMax information is relatively easy to maintain: deletes are ignored, and for inserts and modifies the Min and Max extremes can just be widened using the new values, without need to scan the old values in the table. MinMax indexes are rebuilt from scratch as part of update propagation, which happens in distributed fashion in VectorH already.

7. CONNECTIVITY WITH SPARK

Vectorwise has a `vwload` loader utility for bulk-loading. It provides a range of features enabling users to load from a variety of input files and handle errors in a flexible way. For example, it allows to specify custom delimiters, load only a subset of columns from the input file, perform character set conversion, use custom date formats, skip a number of errors, log rejected tuples to a file, etc. VectorH extends `vwload` with a capability to load data from HDFS in parallel. To achieve high efficiency its functionality was moved into a *vwload* operator that runs directly inside the VectorH server.

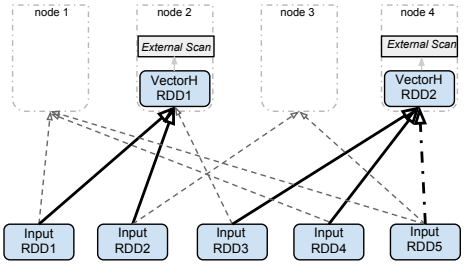


Figure 6: Mapping Spark RDDs to VectorH RDDs and ExternalScan operators

Spark-VectorH Connector. SparkSQL [3] is a module built on top of Apache Spark that enhances it with relational processing capabilities. It has seen increasing popularity among data scientists because of its versatility and tight integration with various systems from the Hadoop ecosystem, Parquet, Avro or Hive to name a few. In early 2015, its Data Source API was released, providing users with the ability to develop their own data source that can export data into SparkSQL, but that can also ingest data from SparkSQL into another database system.

For this connector some new operators were needed in VectorH: `ExternalScan` is an operator that is able to process binary data coming from multiple network sockets (in parallel) and `ExternalDump` that is able to output binary data in parallel through network sockets. Using these operators, we were able to implement (i) the `insert()` and the (i) `buildScan()` methods of the `BaseRelation` interface exposed by the Data Source API. A prerequisite for improved performance was to implement our own `VectorH RDD` which extended Spark's RDD (Resilient Distributed Dataset) interface. This new RDD has exactly as many partitions as `ExternalScan`/`ExternalDump` operators and overrides the `getPreferredLocations()` method in order to specify the affinity of each RDD partition to the location (hostname) of its corresponding `ExternalScan`/`ExternalDump` operator. Consequently, we instruct Spark's scheduler where to process those `VectorH RDD` partitions to get local Spark-VectorH transfers. Finally, we add a `NarrowDependency` between the `VectorH RDD` and its parent RDD that defines the mapping between parent RDD partitions and `VectorH RDD` partitions, using an algorithm similar to Hopcroft-Karp's matching in bipartite graphs. Figure 6 shows an Input RDD with 5 partitions mapped to 2 VectorH `ExternalScan` operators on 4 nodes. Each input partition has an affinity to exactly two nodes (e.g. HDFS blocks in a scenario where replication is set to R=2). The dashed arrows represent these affinities, the solid arrows represent assignments that respect affinity, while the dot-dash arrow represent assignments that are forced to ignore affinity information (and therefore possibly incur network communication).

We are currently able to ingest data into VectorH coming from arbitrary Spark sources: files stored in formats like Parquet, Avro, ORC, etc, but also perform complex computation in Spark and only load the transformed data into VectorH. Furthermore, a SparkSQL user can perform computations in Spark on data read (and possibly filtered) from VectorH. A benefit of loading data into VectorH from Spark is that Spark(SQL) creates one RDD partition per input HDFS block. In scenarios where VectorH and Spark run on the same set of nodes, we can instrument assignments in such a way that all blocks are read locally with short-circuit HDFS reads. This is in contrast with the standard `vwload` utility, which typically reads remote blocks over HDFS.

Performance. We tested loading 650GB in 72 CSV input files with 10 uniformly distributed integer columns into VectorH on the same 6-node cluster used for the experiment in Figure 5. Using the `vwload` utility this takes 1237 seconds. However, by distributing the CSV input files in HDFS such that each node has 12 of them local and tweaking with the parameter order in `vwload` to make the VectorH load operators only read local files, this is reduced to 850 seconds. Using the Spark-VectorH Connector, this works out-of-the-box in 892 seconds, which is impressive given that the data is read and parsed in a different process.

8. EVALUATION IN TPC-H SF1000

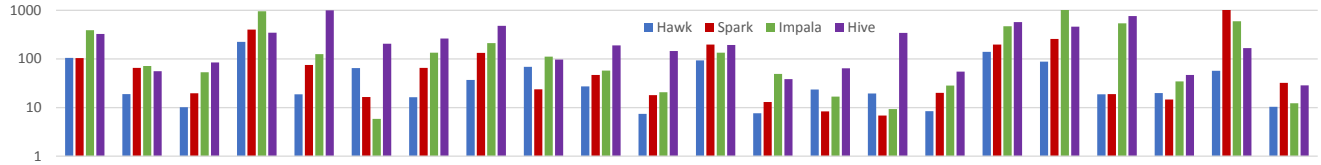
We evaluated the performance of VectorH on a Hadoop cluster on the TPC-H dataset benchmark scale-factor 1000[8] and compare it with the latest version of Impala, Apache Hive, HAWQ and SparkSQL. The cluster consists of 10 nodes with Hadoop 2.6.0 (Cloudera Express 5.5.0) installed. One node runs runs the Hadoop namenode and other services (Cloudera Manager, etc), so we used the remaining 9 machines for the SQL-on-Hadoop experiments. The nodes all have two Intel Xeon E5-2690 v2 CPUs, at 3.00GHz, totaling 20 real cores (40 hyper-threading) and have 256GB RAM (1866 MT/s). Each node has a 10Gb Ethernet card, 24 magnetic disks of 600GB (2.5inch 10K RPM) one holding the OS (CentOS 6.6) and the others for data/HDFS.

VectorH. We tested a pre-release version of VectorH 5.0, built with Intel MPI 5.1.1. Our DDL resembles that used in previous audited TPC-H runs of Vectorwise (Actian Vector): it declares all primary and foreign keys, except for a PK on `lineitem`. Clustered indexes are defined for `region` and `part` on their primary keys; `orders` is clustered on `o_orderdate`, and `lineitem`, `partsupp` and `nation` are clustered on their foreign keys `l_orderkey`, `ps_partkey` and `n_regionkey`, respectively. We also partition `lineitem` and `orders` on `l_orderkey` and `o_orderkey` respectively, as well as `part` and `partsupp` on `p_partkey` and `ps_partkey` respectively, as well as `customer` on `c_custkey` This partitioning scheme, which in all cases uses 180 partitions, means that joins between `lineitem` and `orders` as well as `part` and `partsupp` are co-located mergejoins. The clustered indexes cause selections on date (of any kind) to enable data skipping thanks to MinMax indexes. Total data size was 327GB (in between Parquet at 347GB and ORC at 327GB – both using snappy compression), and thus I/O plays no role. VectorH is CPU-bound on TPC-H, though in Q16 we measured 45% network cost, in Q2,5,10,14,15 around 30% and in Q13,20,22 around 20%, caused by exchange operators with all-to-all communication.

Impala. We tested Impala 2.3 with the official Cloudera TPC-H queries[9], which are more recent than those used in [8]. `lineitem` and `orders` were partitioned by `l_shipdate` and `o_orderdate`, respectively, to trigger Impala's partition

[8]This evaluation does not represent an official score, is not audited, and we omit overall TPC-H metrics. All DDLs and SQL queries are at: `https://github.com/ActianCorp/VectorH-sigmod2016`

[9]github.com/cloudera/Impala/blob/34dc5bd210ddb7af23735f831033d769b0134821/testdata/workloads/tpch



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
VectorH	**1.5**	**1.14**	**3.16**	**0.17**	**1.94**	**0.31**	**2.75**	**1.31**	**11.11**	**1.21**	**1.69**	**0.34**	**3.66**	**0.83**	**1.63**	**1.68**	**1.24**	**0.99**	**1.32**	**2.15**	**1.48**	**2.84**
HAWQ	158.2	21.46	32.06	38.21	36.38	20.19	44.74	48.38	766.4	32.97	12.48	31.75	27.97	19.47	31.58	14.17	173.2	87.08	24.82	42.84	84.7	29.44
SparkSQL	155.4	74.98	62.38	68.27	146.5	5.1	180.2	174.6	264.0	56.62	30.28	66.97	47.65	6.92	11.16	33.81	244.9	254.7	24.89	31.56	1614	91.18
Impala	585.4	81.81	167.7	163.18	242.5	1.81	369.0	276.2	1242.9	69.97	35.04	45.67	180.8	13.95	15.19	47.52	581.53	1234	714.7	74.25	880.8	34.81
Hive	490.1	63.57	266.6	59.08	DNF	63.63	721.8	625.6	1077	230.5	246.1	65.78	140.7	53.23	556.5	92.51	711.7	454.5	1010	100.5	247.7	81.11
after executing updates:																						
	Hive: RF1=34s RF2=112s **GeoDiff=138.2%** - VectorH: RF1=17.8s RF2=8.4s **GeoDiff=102.8%**																					
VectorH	1.68	0.94	3.21	0.23	1.9	0.27	2.74	1.4	11.62	1.21	1.44	0.37	3.9	0.81	1.57	1.64	1.27	0.95	1.5	2.25	1.78	2.82
Hive	608.4	80.8	335.7	205.4	DNF	128.0	690.7	719.8	1150	334.4	218.7	170.5	143.8	130.7	596.7	101.4	891.2	594.6	1167	153.3	275.6	67.85

Figure 7: Table: TPC-H SF1000 results (seconds). Chart: How many times faster is VectorH?

pruning. In Q19 predicate pushdown is not working, which appears to be a known issue.

Hive. We tested Apache Hive 1.2.1 using all available performance optimizations such as the Tez execution engine, vectorization, predicate pushdown, cost-based optimizer and correlation optimizer [13]. We used ORC file with Snappy compression. Partitioning with *bucketing* allows Hive to use local joins and improves performance of a number of queries. The tables are bucketed by the same keys as in the VectorH schema, which leads to the same co-locations. We tuned Tez and YARN, so the number of containers is equal to the number of cores and all RAM is used. Also, we tuned the split size: one mapper is scheduled per container, so the map phase ends in a single wave. The reduce phase uses Hive's auto reducer parallelism feature.

We found some problems: Q5 did not finish in 2 hours, and in Q17 the correlation optimization did not trigger, so lineitem gets scanned twice. Finally, we found that decimal types in columns such as `l_extendedprice`, `l_discount` and `l_tax` are much slower than 32-bit floats; however we used decimals to be fair with the other systems and because the rounding errors inherent to floating point arithmetic are not acceptable in business queries on monetary values.

HAWQ. We tested HAWQ 1.3.1, with 10 segments per node, each with 21GB vmem limit, 12GB for work memory, 6GB for query statements, 256MB shared buffers; merge join optimizations were also enabled. The table schemas were taken from Pivotal[10], in which Parquet storage with snappy compression for the `LINEITEM` and `ORDERS` tables is used. We tried to further improve the performance by partitioning `LINEITEM` and `ORDERS` on dates with 1 month intervals, but this just slowed down many queries.

SparkSQL. We tested SparkSQL 1.5.2 with Parquet storage and snappy compression, and partitioned on `o_orderdate` and `l_shipdate`. SparkSQL was configured to cache tables in RM when possible. Most queries had to be re-written to avoid IN/EXISTS/NOT EXISTS subqueries. Also, some of the queries generated Cartesian Products and had to be manually rewritten to explicitly define the join order.

Results. The total database sizes achieved are 340GB (ORC), 347GB (Parquet) and 364GB (VectorH); showing that applying general-purpose compression to all data (ORC and Parquet use Snappy), which adds significant decompression cost to all table scans, does not provide much extra space savings over doing this only for non-dictionary compressed string columns (VectorH uses LZ4 in this case). VectorH provides interactive performance on TPC-H and is at least one order of magnitude faster than the competition, but sometimes two or even three. HAWQ has the richest physical design options, and is a bit faster than the other competitors (nearest is SparkSQL). We attribute the large difference to VectorH to its PostgreSQL-based query engine, which cannot compete with a modern vectorized engine in terms of CPU efficiency (e.g. see Q1). Impala performance mostly suffers from its single-core join and aggregation processing.

Impact of Updates. Apart from VectorH only Hive allows updates; this results behind-the-scenes in additional delta tables for deletions, insertions, and modifications that must be merged transparently during *all subsequent queries*. This merging can have a cost, which we measure by comparing the geometric mean of the 22 queries before and after ("GeoDiff") the TPC-H refresh functions RF1 (inserts) and RF2 (deletes). Hive query performance after these updates deteriorates to be 38% slower than before. In VectorH, the GeoDiff is 2.8%, which is in range of noise. Therefore, thanks to PDTs, query performance remains unaffected by updates.

9. RELATED WORK

VectorH builds on previous work in high-performance vectorized query engine [27], parallel query optimization using `Xchg` operators [2], the preceding Vectorwise MPP project [7], and a study into creating an elastic DBMS in YARN [5].

The Hadoop formats ORC and Parquet, borrow some ideas from Vectorwise's PFOR, PFOR-DELTA and PDICT compression schemes [28], but are also influenced by Google's Dremel [18] which popularized a nested object format – nested columnar formats as these allow fast mergejoin-like navigation, and putting multiple tables in one file makes co-location easy in distributed environments. An interesting future direction is to use the join-index update algorithms described in the Positional Delta Trees thesis [11] to support updates on such nested columnar formats efficiently. Our micro-benchmarks as well as source-code inspection of the ORC and Parquet readers across the tested systems found these to be unnecessarily inefficient, in their implementation of skipping as well as decompression (e.g., routine use of expensive general-purpose compression such as Snappy as well as tuple-at-a-time bitwise value decoding instead of doing so in vectorized manner exploiting SIMD) [25].

VectorH is the first SQL-on-Hadoop system that instruments HDFS block placement to ensure co-located table par-

[10] github.com/pivotalguru/demos/tree/master/TPC-H%20Benchmark

titions even as HDFS block distribution evolves over time – though the idea of instrumenting HDFS was proposed to co-locate separate column files in [9]. HAWQ [6] has in common with VectorH that it is a SQL-on-Hadoop system that evolved from an analytical database product (i.e. Greenplum). It provides richer SQL feature support and a more mature optimizer [22] than the "from-scratch" efforts SparkSQL[3], Impala [24] and Hive [13]. However, HAWQ cannot do deletes and modifications to HDFS-resident data as it lacks a differential update mechanism like PDTs, and while all other tested systems can deal with failing nodes quite transparently, manual segment recovery is needed in HAWQ. Our performance tests with these competing systems conform with those in [8], though the newer versions of Hive and Impala we tested need fewer reformulated SQL queries to run TPC-H, showing optimizer progress.

Impala shares certain virtues of VectorH, such as out-of-band YARN integration through its Llama component [16], and a modern query engine – in its case using JIT compilation [24]. Here, each operator is compiled in isolation and passes *tuple buffers* from its child-, and to its parent-operators in the pipeline, on each `next()` call. Similar to vectorization, these `next()` calls are thus amortized over many tuples, and intermediate data is stored in memory – possibly CPU cache resident. Impala's *operator-template-based* JIT compilation is much easier to write, test and maintain than the more efficient *data centric* compilation proposed in HyPer [19] that fuses all operators in a pipeline into a single tight loop, avoiding in-memory materialization of intermediates – these stay in CPU registers. Further, Impala does not have co-located table partitioning, or control over HDFS placement, regularly suffers from low-quality query optimization, and lacks multi-core parallelism.

Recently, an early version of VectorH [11] was benchmarked on TPC-H SF100 against a distributed version of HyPer [21], that proposes full communication multiplexing, where all threads funnel their outgoing data through a single multiplexer, to spread the data according to destination and push it onto a high-speed RDMA network. The experimental results differ from our measurements here due to the small dataset and early demo version of VectorH used, yet such work in the direction of network shuffling optimization is of immediate interest for VectorH, whose scalability sweet-spot we soon intend to improve from tens to hundreds of machines. We did not benchmark against distributed HyPer here because it does not work in Hadoop – where high-end RDMA network hardware is seldomly available.

10. CONCLUSION

VectorH is a new SQL-on-Hadoop system built on the fast analytical Vectorwise technology. It raises the bar in SQL-on-Hadoop systems in terms of mature SQL support on many dimensions, elasticity in full cooperation with YARN, updatability powered by Positional Delta Trees (PDTs) and absolute performance. In our TPC-H evaluation against Impala, Hive, HAWQ and SparkSQL on TPC-H, VectorH proved to be 1-3 orders of magnitude faster, thanks to a combination of lightweight compression, effective MinMax skipping, vectorized execution, data partitioning and clustered indexing, control over HDFS block locality, a well-tuned query optimizer and efficient multi-core parallel execution.

In the future, we aim to integrate elements of the VectorH data layout in existing open-source Hadoop formats. For the integration of VectorH in the Hadoop ecosystem beyond HDFS and YARN, the focus is on extending the Spark-VectorH connector into a seamless external tables, UDF and query pushdown facility integrated with Spark.

[11] The development code-name of VectorH was "Vortex" – this name is used to identify the system in [21].

11. REFERENCES

[1] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving relations for cache performance. In *PVLDB*, 2001.

[2] K. Anikiej. Multi-core parallelization of vectorized query execution. *MSc thesis, VU University*, 2010.

[3] M. Armbrust, R. Xin, et al. Spark SQL: Relational data processing in Spark. In *SIGMOD*, 2015.

[4] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: hyper-pipelining query execution. In *CIDR*, volume 5, 2005.

[5] C. Bârcă. Dynamic Resource Management in Vectorwise on Hadoop. *MSc thesis, VU University Amsterdam*, 2014.

[6] L. Chang, Z. Wang, T. Ma, L. Jian, L. Ma, A. Goldshuv, L. Lonergan, et al. HAWQ: a massively parallel processing SQL engine in hadoop. In *SIGMOD*, 2014.

[7] A. Costea and A. Ionescu. Query optimization and execution in Vectorwise MPP. *MSc thesis, VU University*, 2012.

[8] A. Floratou, U. F. Minhas, and F. Özcan. SQL-on-Hadoop: Full circle back to shared-nothing database architectures. *PVLDB*, 7(12), 2014.

[9] A. Floratou, J. Patel, E. Shekita, and S. Tata. Column-oriented storage techniques for mapreduce. *PVLDB*, 4(7), 2011.

[10] G. Graefe. *Encapsulation of parallelism in the Volcano query processing system*, volume 19. 1990.

[11] S. Héman. *Updating Compressed Column Stores*. PhD thesis, VU University, 2015.

[12] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. Boncz. Positional update handling in column stores. In *SIGMOD*, 2010.

[13] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. Hanson, et al. Major technical advancements in Apache Hive. In *SIGMOD*, 2014.

[14] Y. Huai, S. Ma, R. Lee, O. O'Malley, and X. Zhang. .. table placement methods in clusters. *PVLDB*, 6(14), 2013.

[15] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[16] M. Kornacker et al. Impala: A modern, open-source sql engine for hadoop. In *CIDR*, 2015.

[17] P.-Å. Larson, C. Clinciu, E. Hanson, A. Oks, S. Price, S. Rangarajan, A. Surna, and Q. Zhou. SQL server column store indexes. In *SIGMOD*, 2011.

[18] S. Melnik, A. Gubarev, J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *PVLDB*, 3(1-2), 2010.

[19] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9), 2011.

[20] V. Raman et al. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11), 2013.

[21] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. *PVLDB*, 9(4), 2015.

[22] M. A. Soliman et al. Orca: a modular query optimizer architecture for big data. In *SIGMOD*, 2014.

[23] M. Świtakowski, P. Boncz, and M. Zukowski. From cooperative scans to predictive buffer management. *PVLDB*, 5(12), 2012.

[24] S. Wanderman-Milne and N. Li. Runtime code generation in cloudera impala. *DEBULL*, 37(1), 2014.

[25] S. Whoerl. Efficient relational main-memory query processing for Hadoop Parquet Nested Columnar storage with HyPer and Vectorwise. *MSc thesis, CWI/LMU/TUM/U. Augsburg*, 2014.

[26] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *USENIX*, volume 10, 2010.

[27] M. Zukowski. *Balancing Vectorized Query Execution with Bandwidth-Optimized Storage*. PhD thesis, 2009.

[28] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, 2006.

Appendix

For the perusal of the reader, we provide a graphical performance profile for the SF1000 evaluation by VectorH of TPC-H Q1. This query has a simple plan: `Scan` of `lineitem` followed by `Select` that evaluates a date constraint (99% of tuples qualify) then performs computations in a `Project` on `l_tax`, `l_discount`, `l_extendedprice` etc. followed by GROUP BY aggregation in `Aggr` which produces just 4 groups.
This query is parallelized by putting a `DXchgUnion` (DXU 180:1) on top and then aggregating again.
Between the operators we see arrows with the amount of produced tuples plotted. In the detail-boxes for each operator we also see the amount of tuples going in and out of the operator, as well as the time spent evaluating the operator itself (time) and all time spent in the operator and its children together (cum_time).
Below the DXU there are 180 threads, above just one. The parallel operations in the profile show for each thread the time, cum_time and number of produced tuples, in a graph where on the X-axis every point is a thread (notation `Nxx@yy` means: node `xx`, thread `yy`). Here we see that the query spends most time in the lower `Aggr`, `Project` and `MScan`, which is expected, and there is a slight load balancing problem: cum_time in the parallel `Aggr` varies between 2.95G cycles and 3.64G cycles (20%). Still, the overall performance penalty for this is less than 15%.