Analytic Database Design Choices: Vertica's Experience and Perspectives

Chuck Bear, Shilpa Lawande, Nga Tran,
Ben Vandiver, Stephen Walkauskas
Hewlett Packard (Vertica)
150 Cambridgepark Drive
Cambridge, MA, 02140 USA
+1 617 386 4400
{first.last}@hp.com

ABSTRACT
This paper describes the high level architecture of the Vertica Analytic Database, and some lessons learned during its development. While the system's overall features have been described elsewhere in more breadth[1], the goal here is to present some of the good and bad choices that were made over the last 10 years, and to give a sense of how the fundamental design decisions interact with each other.

Categories and Subject Descriptors
H.2.2 [**Database Management**]: Physical Design. H.2.4 [**Database Management**]: Systems – *Relational Databases, Parallel Databases*. H.3.2 [**Information Storage And Retrieval**]: Information Storage – *File Organization*.

General Terms
Design, Performance.

Keywords
Vertica, Analytic Databases, Columnar Databases, Database Execution Engines, JIT Compilation and Vectorization, Mistakes.

1. INTRODUCTION
Since 2010, the market for SQL-based *analytic* RDBMSs has rapidly consolidated[3][4][5][6][7]. However, there is renewed interest in building SQL engines, particularly within the Hadoop ecosystem [8][9][10][11][12]. While it took Vertica half a decade[1] to build a solid analytic SQL database engine from scratch, we hope that if we share our experience and mistakes, those who are building new engines can learn from us and cut their implementation and productization time down to 5 years.

[1]Vertica was founded in 2005, based on the C-Store paper [2]. While Vertica generated revenue prior[13] to the 2010 release of 4.0[14], it is the opinion of the authors that this was the first production-ready, broadly-deployable version.

7th Biennial Conference on Innovative Data Systems Research (CIDR '15) January 4-7, 2015, Asilomar, California, USA.

2. VERTICA ARCHITECTURE

2.1 Design Motivation
Vertica is a SQL RDBMS designed for analytics. Generally, analytic workloads are dominated by operations over many rows per request, rather than workloads consisting of high numbers of requests per second, each retrieving or altering only a handful of rows. Vertica's original design was inspired by the C-Store[2] academic project, though much elaboration was required during commercialization[1], and progress has been made on broader workloads, with features such as fast key lookup queries and flexible schemas[22].

2.2 Physical Storage Layout
Vertica's data storage scheme is perhaps best presented in the context of an example. Consider, as the logical data set, stock trade data. As trades come in, we could record the symbol, date, time, price, quantity/volume, etc., in "insertion order" (Figure 1). Data will be written as rows; the computer will store the values for each row (a trade) sequentially, followed by values for the next row. Inserting data into this structure will be relatively fast, as it is simply appended.

SYMBOL	DATE	TIME	PRICE	VOLUME	ETC
…	…	…	…	…	…
HPQ	*05/13/11*	*01:02:02 PM*	*40.01*	*100*	…
IBM	05/13/11	01:02:03 PM	171.22	10	…
AAPL	05/13/11	01:02:03 PM	338.02	5	…
GOOG	05/13/11	01:02:04 PM	524.03	150	…
HPQ	*05/13/11*	*01:02:05 PM*	*39.97*	*40*	…
AAPL	05/13/11	01:02:07 PM	338.02	20	…
GOOG	05/13/11	01:02:07 PM	524.02	40	…
…	…	…	…	…	…

Figure 1: Insertion order for stock trade data.

Q1: SELECT SUM(volume) FROM trades WHERE symbol = 'HPQ' AND date = '5/13/2011'

However, if we have a query (Q1) that requests the sum of shares traded (volume) with predicates on symbol and date/time (which we may reasonably expect are common), insertion order is not an optimal storage format with regard to disk I/O or CPU computation. Data may be localized with respect to date and time, as those are highly correlated with when data was loaded, and this may enable some data access optimizations for date predicates. But, with regard to stock symbol, the selected rows (highlighted in Figure 1) are not near each other on disk, nor are they far enough apart to be accessed individually in an efficient

manner even if they were indexed. Hence, our first refinement is to keep the physical data sorted by symbol, date, and time (Figure 2).

SYMBOL	DATE	TIME	PRICE	VOLUME	ETC
…	…	…	…	…	…
AAPL	05/13/11	01:02:07 PM	338.02	20	…
AAPL	05/13/11	01:02:03 PM	338.02	5	…
…	…	…	…	…	…
GOOG	05/13/11	01:02:04 PM	524.03	150	…
GOOG	05/13/11	01:02:07 PM	524.02	40	…
…	…	…	…	…	…
HPQ	*05/13/11*	*01:02:02 PM*	*40.01*	*100*	…
HPQ	*05/13/11*	*01:02:05 PM*	*39.97*	*40*	…
…	…	…	…	…	…
IBM	05/13/11	01:02:03 PM	171.22	10	…
…	…	…	…	…	…

Figure 2: Stock trade data, sorted by symbol, date, and time.

By sorting the data by symbol and date+time, our query can, by any of a variety of mechanisms, skip rows that are not relevant to a query such as Q1, and get to the relevant (highlighted) rows quickly. However, more improvements can be made. Since our query only requires three columns from the table (symbol, date, and quantity), by storing data for each column in a separate file (thus making Vertica a *column store*), we can reduce the I/O of this query to just the table cells that will contribute to the result (Figure 3).

SYMBOL	DATE	TIME	PRICE	VOLUME	ETC
…	…	…	…	…	…
AAPL	05/13/11	01:02:07 PM	338.02	20	…
AAPL	05/13/11	01:02:03 PM	338.02	5	…
…	…	…	…	…	…
GOOG	05/13/11	01:02:04 PM	524.03	150	…
GOOG	05/13/11	01:02:07 PM	524.02	40	…
…	…	…	…	…	…
HPQ	*05/13/11*	*01:02:02 PM*	*40.01*	*100*	…
HPQ	*05/13/11*	*01:02:05 PM*	*39.97*	*40*	…
…	…	…	…	…	…
IBM	05/13/11	01:02:03 PM	171.22	10	…
…	…	…	…	…	…

Figure 3: Stock trade data, sorted and stored by column.

The next thing we can do is apply compression to the columns (Figure 4). If data is sorted by symbol, date, and time, the symbol column would be highly compressible with run length encoding (RLE)[2]. Other columns are also highly compressible, for slightly more complex reasons[3].

Once compression has been applied, it is clear that Q1 will require much less I/O than in the initial storage layout. The RLE'd symbol and date columns become quite tiny, and are probably popular enough to end up in a cache. The only I/O that may be necessary is retrieval of the volume data to be summed.

It is also apparent that Q1 would benefit from parallel processing (MPP). Partial sums can be computed by multiple processors (such as machines in a cluster), and then combined.

In the Vertica (and C-Store) lexicon, the physical structure that stores data is called a *projection*. Projections specify how the logical data is to be segmented across a cluster of machines, sorted, and compressed when it is stored persistently.

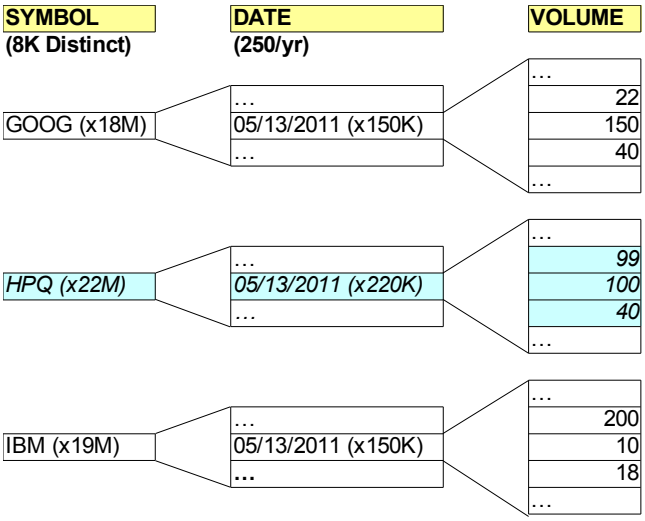

Figure 4: Trade data sorted, columnized, and run length encoded

2.3 Implementation Details

It is a common[4] misconception that Vertica splits data into columns first, and then sorts each column separately. This would make tuple reconstruction exceedingly difficult. Instead, the data is sorted first; hence the *n*th record can be constructed by retrieving the *n*th value from each column file. Retrieving the *n*th value from a column is facilitated by a structure called the *position index*, which is a sparse mapping of positions to file offsets (Figure 5).

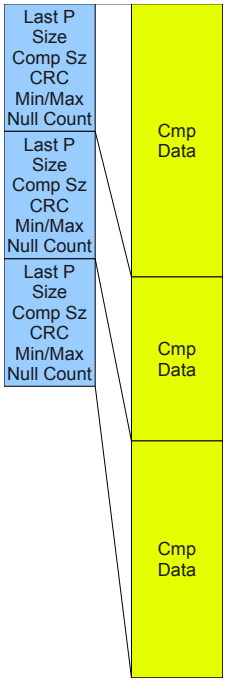

Figure 5: Position Index

[2]In one data set tested, the symbol column had 8,000 distinct values, whereas there were two billion trade rows per year.

[3]For example, while stocks vary in price from pennies to 100K+ USD, once the data is organized by symbol and time, only a few distinct values are seen in a section of the column, hence in practice it is highly compressible using block dictionary techniques.

[4] Among job interview candidates, though perhaps not the readers of this paper

Each *position index* entry records the position number, the file offset for the data, the compressed and uncompressed sizes of the data, and a few statistics, such as a CRC and minimum and maximum values. Thus, retrieving a value at worst requires a binary search of the in-memory[5] position index, followed by a file read and decompression. As this could be expensive for large numbers of tuples, Vertica access paths exclusively use forward-only "skip-scans" of the columns.

2.4 Updates

When considering support for inserts, updates, and deletes, it quickly becomes evident that sorting, optimal compression, and unfragmented storage are fundamentally at odds with efficient implementations of tiny modifications. Leaving "holes" for new data works against compression and scan efficiency. Rebuilding the structure on each modification would also be be prohibitively inefficient on many workloads.

Instead, Vertica does not modify its files. Each large load of new data is stored in a new set of sorted, compressed files. (Smaller loads are collected in memory before being sorted and compressed.) Data is also not directly deleted or updated; instead a sorted list of deleted positions (see 2.3) is kept, and the affected rows are skipped at query time.

2.5 Background Data Reorganization

Giving each load a separate set of files and never truly deleting records would eventually lead to large numbers of tiny files.

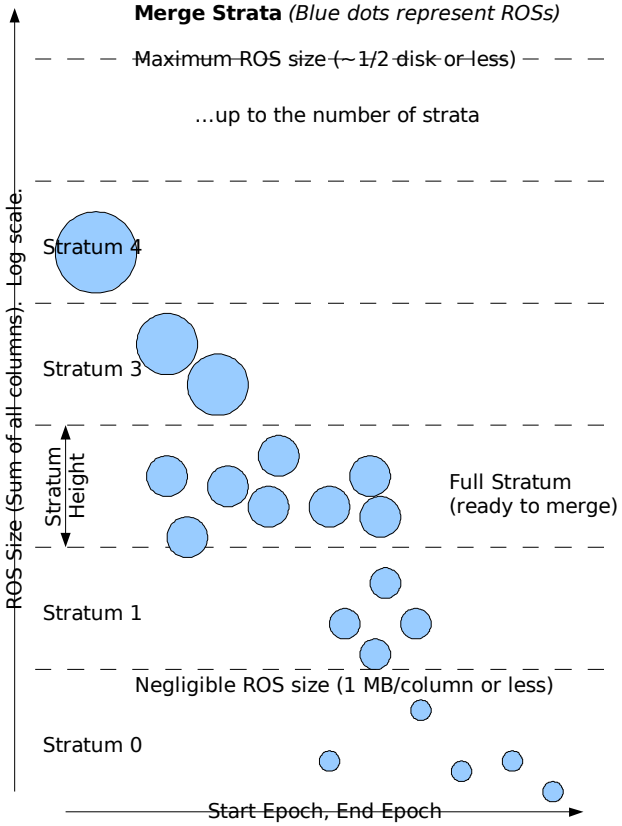

Figure 6: Data files are merged with files of like size, leading to an upper bound on the number of rewrites per tuple.

[5] Position indexes are sparse and are designed to be 1/1000th the size of column data, so as long as there is 1GB of memory for each TB of data stored per node, this is a reasonable assumption.

The solution to this is to reorganize data as a low-priority background operation. Vertica's *tuple mover* handles this, and is tasked with combining small files into larger ones, purging the deleted records, and so forth.

It is important to note that the tuple mover combines files of like size into exponentially larger files (Figure 6). In this way, it is possible to put an upper bound on the number of times a tuple will be rewritten in its lifetime. (This is generally only a handful.)

While not glamorous, the importance of the tuple mover and its maturity cannot be understated. Back in 2010, prior to a major redesign, the tuple mover was a source of significant pain for customers and the support department. Problems were largely alleviated by the idea of combining files of like size (which, unlike the old scheme, requires no tuning), and concurrent tuple mover operations within the same projection. With these enhancements, Vertica accepts most sustained load rates with data batched every second or so, without a significant backlog of maintenance tasks.

2.6 Parallel Processing

Vertica is designed to run on a cluster of commodity machines. Vertica parallelizes computations across nodes for operations that either have no inter-row dependencies, or where data can be partitioned into groups of related rows. There are no assigned "leader nodes"; all nodes are capable of planning queries and distributing work to other nodes for execution. This is possible because the table, projection, segmentation, and other global definitions are replicated and synchronized across all nodes.

In practice, Vertica uses segmentation (distribution of data across nodes, generally via consistent hashing) for more than just uniform data layout. GROUP BY and JOIN operations benefit when data is pre-segmented on their keys, as do analytics with SQL PARTION BY clauses.

Vertica is designed and implemented for a *shared nothing* environment[15]. The basic idea here is that each compute node is connected to its own local storage, and other nodes cannot reach that storage directly. The benefits of this approach are largely in terms of price and performance; there is no expensive interconnect between compute and storage, and there is no potential for a bottleneck in a compute-to-storage switching fabric. On the other hand, this requires that Vertica implement many features from scratch in software, which otherwise could be delegated to the storage layer. Such features include expansion and contraction of compute and storage, rebalancing data across machines, backup/restore, and disaster recovery. At this point, these trade-offs seem to be worthwhile for most customers.

2.7 Transactions and Recovery

The decisions to make Vertica a parallel database and to use an append-only storage organization have deep implications for the transaction and recovery systems.

The use of append-only storage makes implementation of queries in "read committed" isolation trivial. No locks or other concurrency devices are needed; the query simply reference counts the relevant files upon starting, and can be assured that no transactions will update the files or otherwise interfere.

Loads and inserts are also concurrent and transactional; each load builds its own set of sorted files, which it then either attaches to the local catalog if asked to commit, or discards if asked to abort. Transactions demanding serializable isolation are accommodated with table-level locks.

The use of a cluster of machines contributes to high availability. Projections are defined so that rows are each sent to two (or more) machines, such that when a node is unavailable, data can still be accessed on another *buddy* node. Because Vertica's storage is append-only, if a node fails and recovers with its persistent storage intact, it can retrieve all the new data it missed during the outage from its buddies.

2.8 Nothing Novel Except the Combination

One interesting observation is that Vertica was not the first product to bring sorting[16], columnar databases[17], MPP clusters[18], or compression[19] to the database market. Instead, its strength is a combination of complementary design choices, from storage format to transaction model, that emphasize analytic workloads over OLTP[20].

2.9 Live Aggregate Projections

As a result of being a standard SQL database with a unique architecture, many features found in other databases have Vertica counterparts, though with significantly different implementations. For example, many databases have some form of materialized views, one use of which is to preaggregate data to speed up queries. Study has been given to the efficient maintenance of such views, with early or deferred maintenance as primary choices[21].

Vertica's *Live Aggregate Projections(LAP)* feature[23] also preaggregates data to speed up queries, but the implementation is generally unlike that of databases that do in-place updates.

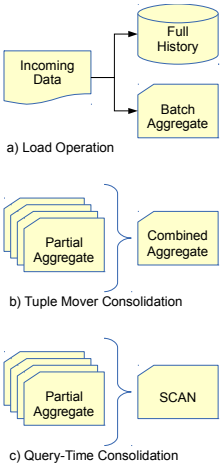

Figure 7: Basic Design of Live Aggregate Projections

Figure 7 illustrates the high-level design of the LAP feature. Data from each load transaction is copied and stored in multiple projections (a). For example, a copy of the full data batch may be placed in a base projection, while a second copy undergoes aggregation (summation, counting, min/max, most recent, etc.) per partitioning key. Loaded data is committed or rolled back as an atomic unit, ensuring transactional consistency. Background tuple mover tasks (b) combine small partial aggregates into larger ones (by summing the sums, taking the maximum of the maximums, etc.), further aggregating the data. Lastly, if multiple partial aggregates remain, they are combined at query time (c). Stored data for LAPs is sorted by the aggregation key, so combining partial aggregates is an efficient "merge" operation applied to sorted streams, and data for individual keys can be found by searching, rather than scanning. This design supports contention-free bulkloads, while keeping the answers returned by the aggregate projection completely consistent with those found in the logical table.

3. DESIGN MISTAKES

3.1 Early vs. Late Materialization in Joins

In the case of column stores like Vertica, it is a compelling idea to scan some columns and apply filtering to eliminate tuples before retrieving values from other columns. Implementation of this concept is fairly straightforward for predicates, but, from even early versions of Vertica, this was also applied to queries with joins.

Q2: SELECT SUM(sv) FROM fact WHERE fk IN (SELECT pk FROM d)

For purposes of illustration, let us consider the query Q2[6]. Figure 8 illustrates several approaches for executing this query.

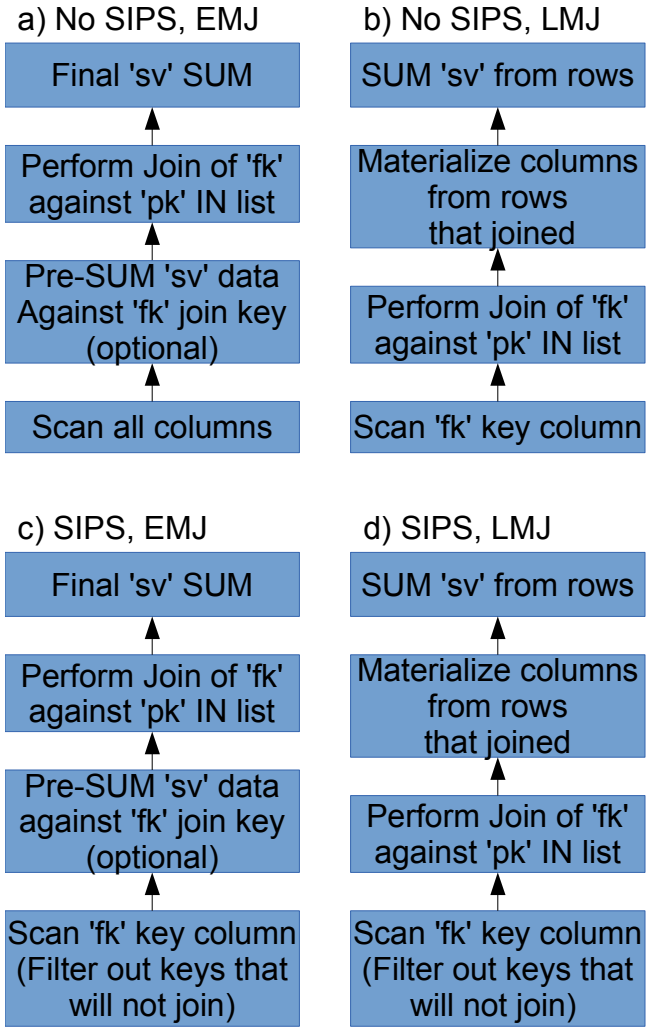

Figure 8: Four of many execution possible plans for Q2

The baseline "unoptimized" execution strategy is shown in (a). First, all data rows and columns are scanned. Optionally, data can

[6]We have intentionally organized the underlying table projection such that the join key is much easier to retrieve than the value to be aggregated.

be preaggregated on the join key to (hopefully) reduce data processed by the join. After executing the join, final aggregation is required. We call this join strategy *early materialization* (EMJ), as all data is materialized before the join.

Strategy (a) is obviously suboptimal, because data that will not join is still scanned. Vertica's first improvement, implemented in the early days, is illustrated in (b). Here, only the key column is scanned before the join, and the remaining columns are materialized later, only for rows that joined. This data is then summed. We refer to strategy (b) as *late materialization*(LMJ).

Later in Vertica's evolution, a new strategy for joins was developed[24]. As shown in Figure 9, the join first loads the smaller relation. Second, information about the keys present is passed to the scan operation for the larger relation, Third, the scan for the larger relation proceeds. This scan need not emit rows that will be eliminated by the join (though it may emit them anyway, without affecting correctness).

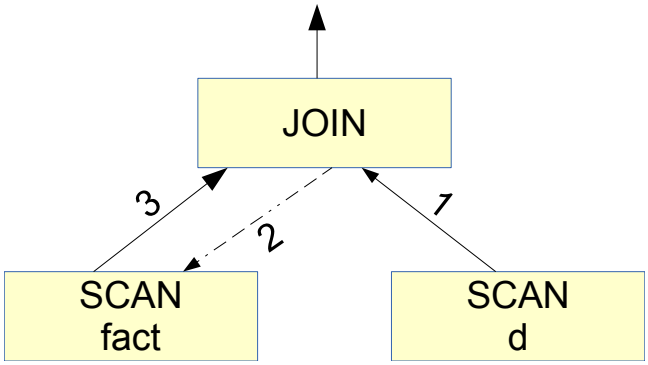

Figure 9: Sideways Information Passing Scheme (SIPS)

Figure 8 (c) and (d) show execution strategies involving SIPS. These are similar to (a) and (b) respectively, except that the scan does not emit most tuples that will ultimately be filtered out by the join. In (c), the scan loads and filters the fk column first, and then loads the remaining columns, essentially achieving the benefits of LMJ.

Table 1 shows performance results for query Q2, with varying selectivity ratios. As expected, strategy (a) shows constant run-time across the selectivity range, while LMJ and SIPS provide much more efficient execution when most rows are eliminated by the join.

Selectivity	Neither Feature	LMJ only	SIPS only	SIPS+LMJ
0.00%	1206	39	23	27
1.00%	1202	63	33	39
2.00%	1200	75	50	57
3.00%	1208	121	75	79
5.00%	1207	151	93	116
10.00%	1200	195	141	191
20.00%	1202	362	405	360
50.00%	1202	1050	1086	1047
100.00%	1204	1720	1222	1724

Table 1: Performance of execution strategies for Q2, with varying join selectivity

There are several notable points here:

1. SIPS is never significantly harmful. This is because SIPS turns itself off, avoiding runtime overhead, if it detects that it is not being helpful.
2. While LMJ is harmful in one case, it it is because of how it changes the application of the grouping operation optimization.
3. While not illustrated here, the LMJ optimization goes over a performance cliff if the join has a large number of keys to apply to the outer 'fact' table. If we cannot handle the inner join keys in main memory, we would have to do a lot of reorganization of the outer to make late materialization efficient. In fact, Vertica bails out in this case and retries the query with SIPS only.
4. SIPS is much more graceful in its performance degradation with large key sets. It can pass information across that admits “false positives” while still filtering rows, or it can be disabled completely, leading to performance equal to the normal ordering of operations.

It is notable that SIPS alone is generally the most efficient option. However, LMJ was far more difficult to implement than SIPS, due to the difficulties of keeping tuple reconstruction context, etc. Ultimately, while Vertica engineering was initially quite proud of late materialized joins, this was not the best course, and while we have not thrown them out, we wouldn't bother to implement them if we didn't have them already.

3.2 Execution Engine Design

It is easy to start with an execution engine design based on tuple iterators[25], and materialization of intermediates (a la M/R) avoids many implementation difficultines. When performance is considered however, both design elements can be dismissed as “too slow”, so the first versions of Vertica opted for a directed acyclic graph (DAG) [26] data flow model, to provide block execution efficiency without any overhead associated with obligatory materialized intermediates.

The pre-3.0 Vertica executor fed all the files related to a query's table projections into a DAG executor, which would apply relevant flow control, and finish when all results reached the root operator, which returned data to the client. Within the executor, all nodes in the DAG which did not require input from other operators (such as scans) would start running. When their outputs were ready for consumption by downstream operators, data would start flowing, and eventually all data would stream through the query execution plan.

This “push” DAG executor was eventually mostly scrapped in 2007-2008, for a handful of reasons, largely revolving around data relevance and parameterization. (The rest were “just bugs”.) In a DAG execution strategy, many leaf nodes execute initially, only to eventually be halted by full output buffers, as the downstream operations (such as a join) are not ready for input. More importantly, some operations have important data to share with their inputs, such as join key data (see 3.1), and it is not possible to do true subquery execution with a DAG executor, as this certainly requires a cyclical data flow.

Instead, Vertica largely uses a “pull” tree for execution, but with data arranged in block groups for efficiency (see 4.1). Essentially, this is just the iterator model, but with each invocation returning a block of rows. The exceptions are INSERT, COPY, DELETE, MERGE DML statements, which cause 'X'-shaped execution plans; the input is a typical execution tree, but the output splits to several downstream targets using DAG dataflow principles.

There are other problems with DAGs, and Vertica encountered several manifestations of one of them, the *diamond problem*. If, within the DAG, there is a producer that sends data to multiple outputs, to then be combined by a consumer with multiple inputs, there is peril that the control flow between the producer and

consumer could be overconstrained. Within the first 8 months, we had observed scan operations providing positions faster than materialization operations could accept them, and it took years before segment+merge combinations were free of deadlock bugs.

3.3 Partitioned Hash Join

Early versions of Vertica did not support joins unless the smaller relation fit in cluster main memory. The first external join algorithm, available in Vertica 3, was a variant of partitioned hash join[27]. When the execution engine was redesigned in Vertica 4, sort merge join was implemented instead for external joins. At the time, this was done for several reasons:

1. It was simple to implement. The sort code was already available in the new executor, but the partitioning code was not.
2. The implementation of PHJ had become notorious for its flakiness, and was known to completely wedge the Linux CFQ scheduler (circa 2008).
3. Vertica was adding support for fast inequality and range predicates, which depend on sorting anyway.

This turned out to be a good choice for other reasons, which became clear later:

4. SMJ was found to be faster, more efficient, and more scalable than PHJ (Figure 10). Since then, as Vertica's sort has become more efficient with the addition of JIT compilation, the gap has probably widened.
5. Some sorting in SMJ can be skipped if either input is already sorted.
6. Generating useful filtering data for SIPS is easier in SMJ.

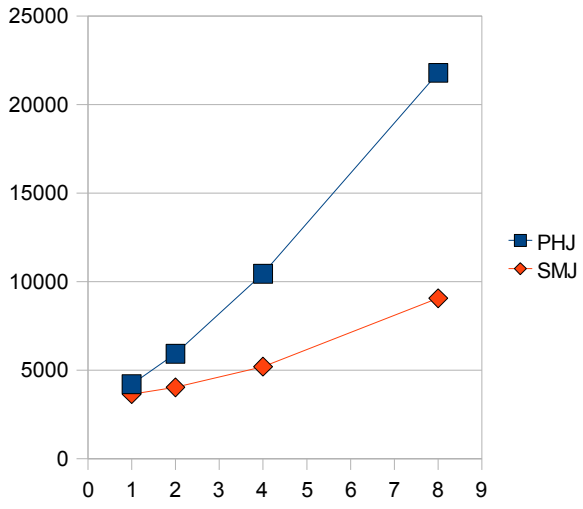

Figure 10: Join run time (vertical) vs number of concurrent joins (horizontal)

3.4 Buddy Projection Sort Order

As discussed above (2.7), Vertica achieves high availability by "k-safety", keeping multiple copies of the data, with the copies shifted around the cluster. Vertica initially claimed that these *buddy projections* were superior to straightforward replication in that the buddy projections need not be sorted the same way, hence the extra disk space could be used to better optimize for the query workload. However, this suggestion has several drawbacks, some of which are obvious in retrospect:

1. While the queries are in fact sped up when all nodes are available, when a node is down all queries will get slower because they will have to use data from a non-optimized projection. In a synthetic benchmark, pretty much any speedup/slowdown factor can be achieved, but in one real example, ~3x speedup with all nodes up resulted in a 42x slowdown in the case of node-down operation.
2. The query optimizer, which takes the sort order into account at a very deep level, was encumbered by significant complexity.
3. Rebuilding a node from scratch was slow. Data had to be decompressed, sent across the network, sorted to the new order, and compressed again, whereas with replicas, the compressed files are simply copied across the network.
4. Rebuilding a node from scratch was not even always possible if the disk utilization was higher than recommended, because there was insufficient temporary space for the external sort algorithm.

Needless to say, all large production customers of Vertica currently use replica buddy projections.

4. GOOD IDEAS

4.1 JIT Compilation and Vectorization

Processing data via sequences of operations that are only known at run-time can lead to a variety of CPU inefficiencies, such as indirect branches, conditional logic, and so on. As JIT compilation and vectorization are standard techniques for spending less CPU cycles on figuring out what to do vs. actually doing it, use of these techniques in databases has, of course, been explored[28].

The value of running batches of tuples through an operation before proceeding to the next operation, thus amortizing program control flow costs, has been a key consideration in Vertica's execution engine design since long before the first customer paid us money. The approach provides significant rewards with only minor implementation cost; in the first Vertica experiment a simple aggregation query went from a CPU-bound 5 minutes to a significantly I/O-bound 2 minutes in just an afternoon of adjustments to the expression evaluator. Since that early validation, we have batched everything. Such batching also provides for the possibility of vectorization, though this is currently left up to the compiler.

The most complex use of batching is in joins, where either data input block can become empty, the data output block can become full, or other things may need attention. In this case, coroutines were applied (despite the slight complexity of getting these to work in C++), to preserve the performance benefits of batching, while keeping the code sane.

JIT compilation is significantly more complex to implement, and somewhat trickier to debug. However, some operations, like comparing, merging, and sorting, are best handled in a row-by-row manner. For this reason, JIT compilation of low-level tuple operations was added in Vertica 4.1 (2009).

The first attempt at JIT compilation in Vertica was built using LLVM[29], however the tool exhibited high compilation time (several milliseconds per query), resulting in unacceptable query

setup overhead[7]. The AsmJIT[30] tool was used instead, despite the resulting optimization only applying to x86 architectures.

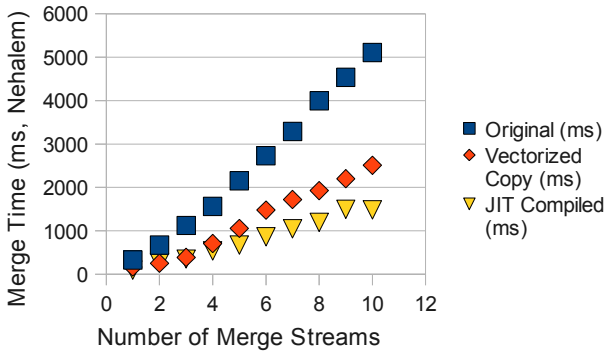

Figure 11: Run time for a merge, with various execution strategies.

Figure 11 shows the performance achieved in a simple test dominated by the cost of merging streams of tuples. (Note that as streams are added, tuples are added, so a line at constant slope up and to the right would mean that performance was not impacted by merge degree.) It is clear that vectorization of the merge's output is helpful, but that JIT compiling tuple comparison and movement provides an additional performance boost.

4.2 Collect Data

The foundation of "Big Data" analytics is that organizations should collect data from the outset, and not "throw anything away". This enables questions to be answered correctly and promptly, with such benefits as tight feedback cycles for product and process improvement[31]. Organizations that don't place a cultural emphasis on data collection and don't value empirical evidence will suffer competitive disadvantage. Hence, an analytic database that does not provide insights into its own operation, to its users and to its creators, would be severely hypocritical.

As of 4.1, Vertica provides SQL-queryable logs of many things. Among the data logged are system statistics (CPU, network protocols, disk, memory, etc.), user requests, errors, query optimizer choices, table and projection usage, individual executor row counts and run times, tuple mover and other background tasks, file lineage, upgrades, configuration changes, and backup history. This data is available to database administrators and users via direct SQL, useful views joining the logs together, and the Vertica Management Console, which provides visualizations of the underlying data.

Vertica also collects this data from customers, particularly when incidents are reported to the support department. At the most recent Vertica Customer Advisory Board, all companies in attendance (with the predictable exception of two banks and an insurance company) were not merely supportive, but enthusiastic about automatic collection, as they understand the value of data as an input into continuous product improvement.

The result is that when developers or product managers ask, for example, "does anyone actually use this feature", there is no need to guess. Figure 12, for example, shows which Vertica features were used to design the physical data layout projections for a variety of customer databases[8].

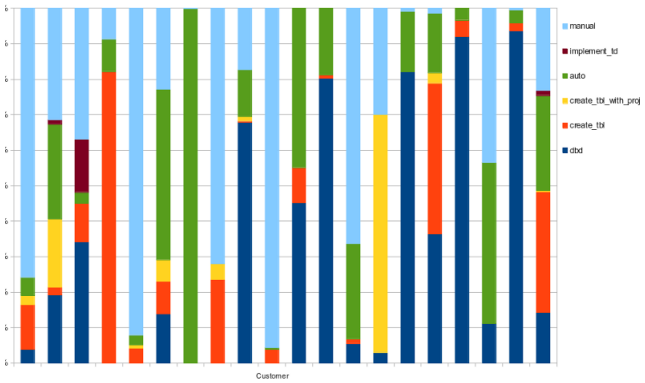

Figure 12: Source of physical database design (vertical), for a selection of customers (horizontal)

5. ACKNOWLEDGMENTS

The Vertica Database is the product of the hard work of many engineers who took their fair share of wrong turns. Without the guidance of academics and other practitioners (the references section below is but an incomplete list), the number of such wrong turns would have undoubtedly been larger. Having the BSD-licensed PostgreSQL[32] as a starting point allowed engineers to immediately start working on the aspects that make Vertica unique, without spending much time on items such as a SQL parser or testing harness. Vertica also owes its timing to AMD64, as commodity 64-bit technology was an essential ingredient to a cost-effective the business model. Further acknowledgment goes to our investors and early adopter customers, without which development would not have been started, or would have been prematurely aborted.

6. REFERENCES

[1] Lamb, A., Fuller, M., Varadarajan, R., Tran, N., Vandiver, B., Doshi, L., and Bear, C. 2012. The Vertica Analytic Database: C-Store 7 Years Later. In *Proc. VLDB '12*, 1790-1801. http://vldb.org/pvldb/vol5/p1790_andrewlamb_vldb2012.pdf

[2] Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S. 2005. C-store: a column-oriented DBMS. In *VLDB '05,* 553-564. http://db.lcs.mit.edu/projects/cstore/.

[3] Kanaracus, C. 2010. Teradata Buys Analytics Vendor Kickfire. *InfoWorld.* http://www.infoworld.com/article/2625533/m-a/teradata-buys-analytics-vendor-kickfire.html.

[4] EMC Corporation. 2010. EMC To Acquire Greenplum. http://www.emc.com/about/news/press/2010/20100706-01.htm

[5] IBM Corporation. 2010. IBM To Acquire Netezza. http://www-03.ibm.com/press/us/en/pressrelease/32514.wss

[6] HP. 2011. HP To Acquire Vertica. http://h30261.www3.hp.com/phoenix.zhtml?c=71087&p=irol-newsarticle&ID=1528593

[7]Run-time of our unit test suite, which is a blend of small queries, roughly doubled.

[8]This particular illustration was created by a developer who wanted to know if "Implement Temp Design" was ever used; the answer is apparently "rarely".

[7] Teradata. 2011. Teradata To Acquire Aster Data. http://www.teradata.com/News-Releases/2011/Teradata-to-Acquire-Aster-Data

[8] Kornacker, M. and Erickson, J. 2012. Cloudera Impala: Real-Time Queries in Apache Hadoop, For Real. http://blog.cloudera.com/blog/2012/10/cloudera-impala-real-time-queries-in-apache-hadoop-for-real/

[9] Sears, J. 2014. Discover HDP 2.2: Even Faster SQL Queries with Apache Hive and Stinger.next. http://hortonworks.com/blog/discover-hdp-2-2-even-faster-sql-queries-apache-hive-stinger-next/

[10] Hausenblas, M. and Nadeau, J. 2013. APACHE DRILL: Interactive Ad-Hoc Analysis at Scale. In *Big Data* 1, 2 (June 2013). https://www.mapr.com/sites/default/files/apache_drill_interactive_ad-hoc_query_at_scale-hausenblas_nadeau1.pdf

[11] Harris, D. 2012. How one startup wants to inject Hadoop into your SQL. Gigaom. https://gigaom.com/2012/07/24/how-one-startup-wants-to-inject-hadoop-into-your-sql/

[12] Xin, R. 2014. Shark, Spark SQL, Hive on Spark, and the future of SQL on Spark. https://databricks.com/blog/2014/07/01/shark-spark-sql-hive-on-spark-and-the-future-of-sql-on-spark.html

[13] http://www.vertica.com/wp-content/uploads/2011/05/ZyngaSocialGraphing.pdf

[14] http://www.vertica.com/content/vertica-broadens-reach-of-its-industry-leading-analytic-database-with-version-4-0/

[15] Dewitt, D. and Gray, J. 1992. Parallel Database Systems: The Future of High Performance Database Processing. In *Comm. ACM* 35, 6 (June 1992), 85-98. DOI= http://doi.acm.org/10.1145/129888.129894

[16] Knuth, D. 1998. *The Art of Computer Programming: Sorting and Searching*, 2 ed., vol. 3. Addison-Wesley.

[17] http://www.sap.com/pc/tech/database/software/sybase-iq-big-data-management/index.html

[18] Tandem Database Group. 1989. NonStop SQL, A Distributed High Performance, High Availability Implementation of SQL. In *Proceedings of HPTPS*, 1989.

[19] Graefe, G. and Shapiro, L. D. 1991. Data Compression and Database Performance. In *Proceedings of the Symposium on Applied Computing*.

[20] Stonebraker, M. and Cetintemel, U. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st ICDE* (ICDE '05). IEEE Computer Society, Washington, DC, USA, 2-11. http://dx.doi.org/10.1109/ICDE.2005.1

[21] Zhou, J., Larson, P., and Elmongui, H. 2007. Lazy Maintenance of Materialized Views. In *Proc VLDB '07*. 231-242.

[22] HP. 2014. HP Vertica Flex Tables Quick Start http://my.vertica.com/docs/7.1.x/PDF/HP_Vertica_7.1.x_FlextablesQuickstart.pdf

[23] HP. 2014. Live Aggregate Projections. http://my.vertica.com/docs/7.1.x/HTML/Content/Authoring/AdministratorsGuide/Projections/AggregateProjections.htm

[24] Shrinivas, L., Bodagala, S., Varadarajan, R., Cary, A., Bharathan, V., and Bear, C. 2013. Materialization Strategies in the Vertica Analyti Database: Lessons Learned. *ICDE '13*. 1196-1207.

[25] Graefe., G. 1994. Volcano— An Extensible and Parallel Query Evaluation System. *IEEE Trans. on Knowl. and Data Eng.* 6, 1 (February 1994), 120-135. http://dx.doi.org/10.1109/69.273032

[26] MapR. 2012. Anatomy of a Spark Job. https://www.mapr.com/products/product-overview/apache-spark

[27] Dewitt, D. and Naughton, J. 1995. Dynamic Memory Hybrid Hash Join. http://diaswww.epfl.ch/courses/adms07/papers/HybridHashJoin.pdf. University of Wisconson, Madison.

[28] Sompolski, J., Zukowski, M., and Boncz, P. 2011. Vectorization vs. Compilation in Query Execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware* (DaMoN '11). ACM, New York, NY, USA, 33-40. http://doi.acm.org/10.1145/1995441.1995446

[29] LLVM. http://llvm.org/

[30] AsmJit. https://github.com/kobalicek/asmjit

[31] Kiron, D., Ferguson, R. B., and Prentice, P. K. 2013. From Value to Vision: Reimagining the Possible with Data Analytics. MIT Sloan Management Review. http://sloanreview.mit.edu/reports/analytics-innovation/

[32] PostgreSQL. http://www.postgresql.org