Efficient Many-Core Query Execution in Main Memory Column-Stores

Jonathan Dees #1
# SAP AG
69190 Walldorf, Germany
jonathan.dees@sap.com

Peter Sanders #2
* Karlsruhe Institute of Technology
76128 Karlsruhe, Germany
sanders@kit.edu

Abstract — We use the full query set of the TPC-H Benchmark as a case study for the efficient implementation of decision support queries on main memory column-store databases. Instead of splitting a query into separate independent operators, we consider the query as a whole and translate the execution plan into a single function performing the query. This allows highly efficient CPU utilization, minimal materialization and execution in a single pass over the data for most queries. The single pass is performed in parallel and scales near-linearly with the number of cores. The resulting query plans for most of the 22 queries are remarkably simple and are suited for automatic generation and fast compilation. Using a data-parallel, NUMA-aware many-core implementation with block summaries, inverted index data structures, and efficient aggregation algorithms, we achieve one to two orders of magnitude better performance than the current record holders of the TPC-H Benchmark.

I. INTRODUCTION

The starting point for the research presented here are three recent developments that enable significantly improved performance of relational databases:

- Storing each column separately reduces the amount of data that has to be accessed for typical queries that use only a small fraction of the columns present in industrial databases.

- Decreasing RAM prices make it possible to store fairly large databases entirely in main memory. This greatly improves the available bandwidth and reduces latencies by several orders of magnitude. In particular, this nullifies possible disadvantages of column-store databases which have a more fine-grained memory access pattern than row-store representations. This effect is amplified by the fact that columns can often be compressed further decreasing memory cost and saving memory bandwidth.

- Many-core processors have high processing power at relatively low cost. In contrast to cluster based systems, the shared memory of these systems gives us more flexible access to the data.

These developments make it possible to access large amounts of data very fast. To take full advantage, the CPU needs to keep up with processing the data.

However, traditional database systems are usually based on the assumption that the access to the data is the main bottleneck. A standard processing technique is the Volcano iterator model [1]: Each tuple is processed one by one (the tuple can be just a single value). This incurs a significant interpretation overhead for each operation actually performed. For example, to access the next tuple, a function `next()` is called hiding the underlying logic to receive the tuple (e.g., decompression). This results in a simple and easy to use interface but has several drawbacks:

First, we need to do a real function call which cannot be inlined as we use a virtual function or a function pointer, also hindering branch prediction. Second, we need to remember and query the iterator state to provide the correct next tuple. Calling many different `next()` functions in one iteration increases the danger of running out of (L1)-cache for the execution code.

One solution to this problem is to process several tuples at once per step, i.e., one `next()` call returns \( n > 1 \) tuples [2], [3] or even all [4] instead of one. This usually requires to materialize the resulting tuples. Other operators are extended as well such that they process \( n \) tuples at a time. Note that this is not fully possible for all operators (e.g. pipeline-breakers like sort or join). When \( n \) is large enough, the overhead for the call is negligible. For \( n \) not too large, tuples still reside in cache when the next operator processes them. However, we lost the full pipelining power as we need to materialize \( n \) tuples in one step and read it again for the next operator from cache (potentially second or third level cache). This forces additional CPU stalls when waiting for memory and may consume memory bandwidth when data is not in cache any more.

The best of both worlds would therefore be: full pipelining and no overhead for processing single tuples. We achieve this goal by a simple but radical approach: Writing the complete query plan directly in C++ and compiling it to native machine code for execution. This gives us following advantages:

- tuples can be passed in registers
- minimal materialization due to full pipelining
- no overhead from interpretation or function calls
- full optimization knowledge from the compiler

This results in high CPU utilization and memory bandwidth consumption. We wanted to know how big the benefits can be when combined with algorithmic techniques using the full capacity of modern parallel hardware. In this paper we use hand-written code optimized not only for scalability and efficiency but also for simplicity and orthogonality between query plan issues and the details of parallelization and data structures. In ongoing work, we build on this to address...
automatic compilation. First results indicate that using new compiler technologies like LLVM and Clang C++ [5], automatic translation yields similar performance at translation latencies around 10ms.

We use the data and queries of the TPC-H benchmark [6] as a case study. We view TPC-H as a good starting point since it defines a diverse set of nontrivial queries. This means that efficient algorithms, data structures and parallelization strategies can have a significant performance impact. It also allows direct comparison with runtime results from other systems.

Looking at database applications in particular, better performance has several practical implications: On the one hand we need it in order to cope with rapidly growing data sets. Growing efficiency also allows us to save hardware and energy costs on the one hand and to open up new applications on the other hand. For example, in the past classical decision support queries were used by a small number of people to make strategic decisions tolerating long latencies. With two orders of magnitude lower query latencies, we can afford equally expensive queries on a regular basis used by a large number of people expecting immediate answers as in web search engines. One goal of our research is to explore new techniques to further improve the performance of the parallel main memory based SAP HANA Database.

After introducing basic concepts in Section II, we present our algorithmic techniques in Section III. Section IV describes implementation aspects and Section V reports on experiments. Section VI summarizes the results and discusses how they can be transferred to a more general setting with updates and an SQL query compiler. The appendix includes pseudocode describing the implementation of all 22 queries.

Related Work

Using a column-oriented instead of a row-oriented data representation has proven to result in a significant speedup for complex analytical queries both for main memory and disk based systems [7]. Today, there exist several academic column-oriented database systems, e.g., C-Store [8], MonetDB [4] or MonetDB/X100 [3] as well as commercial products like Vertica [9] (based on C-Store), VectorWise [10] (based on MonetDB/X100) or Sybase IQ [11]. All these systems load the accessed data into main memory to process it. Still they are partly designed to handle the data efficiently on disk.

For query execution, C-Store processes tuple by tuple and hands one tuple from one operator to the next (Volcano iterator model). The processing of one tuple in an operator induces a significant performance overhead due to function calls and plan interpretation. MonetDB faces this problem by processing all tuples in one operator step making the performance overhead for the call and plan interpretation negligible. Each operator is available as a precompiled function for each input type combination and can be fine-tuned for efficient CPU usage and high IPC (instructions per cycle). Although these functions are very efficient, we lose the ability to pipeline data, i.e. we need to materialize all resulting tuples in main memory for each operator step, consuming a lot of memory bandwidth limiting the execution speed. MonetDB/X100 reduces this problem by processing only several tuples in an operator step and introducing pipelining again. If the number of tuples is carefully chosen, the next operator in line finds the input tuples still in cache (L2 or even L1 cache), significantly increasing access speed and reducing memory consumption. Note that we still need to materialize the data and pass it via caches instead of registers, losing the full potential of pipelining.

An interesting fact is that one starting point for MonetDB/B/X100 was the good performance of a query written directly in C++ [3]. Even for the most simple TPC-H query (1) which uses only a single table, the C++ implementation clearly dominates other systems. The comparison included only one query with 1 GB data and was single-threaded.

There are some recent approaches for generating code for a given SQL query. HIQUE [12] is a prototype system generating C code for each incoming SQL query. Compilation times of the system are clearly visible and the reported execution times for three tested TPC-H queries are close to MonetDBs: query 1 and 3 are faster, query 10 is slower. The translated operators are still separately visible and not fully pipelined. Also note that the data of a table is stored in classical tuple format on disk.

In HyPer [13], SQL queries are compiled using LLVM [5] and C++. Here, code compilation times are lower and operators are more interlaced. They used a modified TPC-H benchmark version (TPC-CH [14]) which makes direct comparison more difficult. Running times for TPC-H like OLAP queries are below MonetDB and VectorWise by a factor of 1–2 for 5 tested queries.

Both [13] and [12] do not use parallel execution or data compression, they use only scale factor one in the experiments (1 GB data).

DBToaster [15], [16] is another database system which uses C++ code generation and compilation to speed up their execution. However, it does not target the execution of general SQL Queries but produces specific C++ code for efficiently updating materialized database views on insertion or deletion of data.

There has been a lot of research on parallel databases in the late 1980s and early 1990s [17], [18]. Our approach to addressing locality owes a lot to the shared nothing concept developed during that time. Böhm et al. [19] achieve good speedups for TPC-R – the predecessor or TPC-H – on a distributed memory machine. However, this approach is expensive since it is based on replicating all the data.

II. Preliminaries

Figure 1 summarizes the database schema for the TPC-H Benchmark which simulates a large database of orders consisting of 1–7 line items each. Except for two constant size tables NATION and REGION, the other relations are scale factor $SF \times$ times a (large) constant such that the total size of an uncompressed row representation is about $SF \cdot 10^9$ bytes. Attribute values are populated using rules that are realistic


To a certain extent but ultimately contain uniformly random decisions. There are 22 queries each of which also involves a small number of random parameters. In a power test, queries 1–22 are executed one after the other. In a throughput test, a prescribed minimum number of query streams is processed where the order or queries in each stream is prescribed but scheduling between streams is flexible.

We are targeting shared memory machines with multiple sockets each containing a multi-core processor with a significant number of cores and simultaneously executing threads. Each socket has its own locally attached main memory. Accessing memory of other sockets comes with a performance penalty. There is a complex hierarchy of caches. With increasing cache size, latency grows and more threads share the same instantiation of this cache.

III. TECHNIQUES

A. Database Schema

We consider database schemas that can be viewed as a generalization of the snowflake schema [20] which is in turn a generalization of a star schema. The relations form a directed acyclic schema graph $G$ where an edge $(R, U)$ indicates that a tuple of relation $R$ references a tuple of relation $U$. Already the TPC-H benchmark (Figure 1) shows that the graph is not necessarily a tree. Paths in $G$ specify a way to access data via a sequence of key dereferencing operation. From the point of view of a node $R$, DAG $G$ can be viewed as a compact representation of a closure relation $R^*$ that contains an attribute for every path starting at $R$. This view resembles the network model [21].

B. Data Representation

Basic Data Types and Compression: Most attributes can be mapped to integers, i.e., numbers, enumeration types, and dates. The latter are stored as number of days since Jan 1st 1970 in TPC-H. For reasons of efficiency and simplicity, we currently store all scalar values using 1, 2, 4, or 8 bytes. For example dates take 2 bytes. We have also implemented more general bit compression, i.e., only storing exactly the number of bits needed but we are currently not using this for performance reasons. We also have not tried more sophisticated compression schemes like [22], [23] for the same reason. More generally, we do not think that TPC-H is a good basis for experiments on data compression since the randomly generated data is unlikely to yield good prediction for real world scenarios. Strings stemming from a small set of possible values are compressed into an enumeration referencing string values in a dictionary. For example, there are only 150 strings denoting part types. More complex strings are currently not compressed. Note that we can often circumvent actually processing these strings using the inverted text indices introduced in Section III-C.

Block-cyclic allocation: Each column of the database is split into blocks which are assigned to the local memories of each socket of a NUMA machine in a round robin fashion. Blocking ensures cache efficiency while the cyclic distribution ensures that the memory banks are about equally loaded. Note that simply splitting a column into one segment for each local memory would yield very bad performance since many queries do most of their work on only a subrange of a column. For example in TPC-H Query 4, ORDERS and LINEITEMS are selected from a small time interval. We handle the assignment of memory blocks to sockets transparently: The application still uses a contiguous memory range as this range is virtual and further translated to physical memory by the operation system. The assignment of virtual to physical addresses can be modified by a system call on linux systems such that the physical address points to the correct socket [24]. For the size of a memory block we use a multiple of the system page size. We keep track of the assignment of blocks to NUMA nodes as we use this information for optimal memory access when iterating over the data (see Section III-E).

C. Index Data Structures

Index data structures allow us to efficiently navigate the schema graph. We have implemented several types of index data structures. Note that these data structures are very different from the B-tree based indices used in traditional disk-based column-store databases and rather resemble relations of the network data model [21] in which each tuple of a table references one, or an arbitrary set of tuples from another table.

Forward Join Indices: Recall that an edge $R \rightarrow U$ in the schema DAG means that each tuple $r \in R$ contains a primary key logically referencing a unique element of a tuple $u \in U$. A forward index makes this referencing explicit by also storing the actual position of $u$ in $r$.

Indexing Sorted Relations: When a relation $R$ is sorted by the primary key of another relation $U$ reachable from $R$ in the schema graph, a tuple $u \in U$ may simply specify the first tuple in $R$ that references $u$. The desired set can then be found by simply scanning $R$ starting at $u$. We use this type of index to reference LINEITEMs from ORDERs.

Inverted Join Indices: These efficiently go backwards in the schema graph $G$ described in Section III-A. Suppose there is a path $P$ from relation $R$ to relation $U$ in $G$. Then, given a tuple $u \in U$ we want to lookup all tuples of $R$ that reference $u$ via $P$. Inverted indices support this type of lookup by explicitly listing the tuples in $R$ referring to $u$. Note that the path may contain more than a single edge. For
example, in TPC-H Query 11, we use an inverted index listing all PARTSUPPs with SUPPLIERs from a given nation. Note that we do not yet use sophisticated index compression techniques like [25], [26] which could further reduce space usage while keeping high performance. However, in our experiments, inverted indices take less than one third of the total space so that this optimization seems not very pressing.

**NUMA Aware Inverted Join Indices:** Our system tries to schedule tasks accessing tuples on socket $S$ to threads also located on $S$ to improve memory access. Tuples referenced by a tuple on socket $S$ are likely to be accessed in the same task and should therefore be located on the same socket. This is not possible in general. Still, for a relation where $R$ is sorted by the primary key of another relation $U$ we can adjust the NUMA memory block distribution of $R$ such that tuples $r \in R$ reside on same socket as the referencing tuple $u$. In TPC-H, we apply this for ORDER and LINEITEM.

For inverted join indices we introduce a new NUMA aware data structure: Suppose we have an inverted index from relation $R$ to relation $U$. For a tuple $r \in R$, we have a set of referenced tuples $u \in U$ for each memory socket. Each set of a socket holds the tuple positions of all tuples located on this socket. When we use this index during query execution, we do not directly access the referenced tuples but start a new task for each socket. The tasks are then executed by a thread located on the corresponding socket having local memory access to the referenced tuples.

As we need to start an extra task for each memory socket, this index should only be used if the list of referenced tuples for one tuple $r \in R$ is long enough on average and the NUMA effects outweigh the additional work.

**Inverted Text Indices:** Consider any text attribute $t$ of a closure relation $R^*$. We can view the texts in $t$ as a collection of documents and use standard techniques from information retrieval to support fast search. In particular, we can store an inverted index storing for each word $w$ occurring in some document the list of tuple IDs in $R^*$ where $t = w$. Unfortunately, this way of searching is not immediately compatible with SQL queries of the form like ‘%pattern%’ because such predicates also match substrings of a word. For example, in the TPC-H there are eight queries with keyword like. They all will actually only match full words but the question is how to rule out other kinds of matches. Fortunately there is a fast way to do this: we build a character based full text index $C$ on the dictionary of words occurring in any text in $t$, for example a suffix array. In many applications and in the TPC-H, this dictionary is very small so that the additional space consumption is negligible. Then when evaluating a like-query for pattern ‘%x%’ we search for all occurrences of $x$ in $C$. If the number of occurrences is small, we can use the inverted index for those words to quickly find all relevant tuples. In particular, if $x$ is actually a word that is not a substring of another dictionary entry then there will be exactly one occurrence.

**D. Block Summaries**

For columns whose values are correlated with the ordering column, we store minima and maxima for each block similar to Netezza’s zone maps described in [27]. For example, in TPC-H, ship dates, commit dates, and receipt dates off LINEITEMs are strongly correlated with order dates which are the primary sorting criterion. This optimization causes only negligible storage overhead since it stores only two additional values per data block. When we select the rows of a column lying in range $[a, b]$ we have five cases for each block:

1) $\min \geq a$ and $\max \leq b$: select all rows
2) $\min > b$ or $\max < a$: no row is selected
3) $\min > a$ and $\max > b$: only check for $\leq b$.
4) $\min < a$ and $\max < b$: only check for $\geq a$.
5) $\min < a$ and $\max > b$: both checks needed

Except for case (5) we gain an advantage. In cases (1) and (2) we do not have to touch the values at all and thus save memory bandwidth.

**E. Scanning**

Many SQL queries that are written down as a complicated combination of joins can actually be viewed as a simple select-from-where query on a closure relation $R^*$ and can be implemented by scanning relation $R$ and accessing attributes corresponding to paths in the schema graph by explicitly following the path using our indices. Doing this efficiently requires a number of optimizations.

**NUMA aware Parallelization using NBB:** Our initial implementation used Intel Threading Building Blocks (Intel TBB) [28]. It splits a given range of a column to be scanned into smaller ranges (blocks) and assigns the blocks as tasks to different threads. For small ranges, these blocks are only a fraction of a memory block in order to allow for enough parallelism and load balancing. Unfortunately, Intel TBB is not NUMA-aware so far – there is no way to reliably assign a thread to a CPU and therefore we cannot ensure that the accessed memory resides on the local socket.

To overcome this limitation, we built our own NUMA aware scheduler, called NBB [29] based on POSIX Threads. NBB manages a pool of threads for each socket. We generate NBB tasks such they scan a piece of a relation located on a single socket and NBB tries to assign this task to a thread on this socket – only using threads from remote sockets when all local threads are busy.

**Ordering Predicate Evaluations:** The where-clause of most queries restricts the set of tuples contributing to the output. Often, this restriction is a conjunction of several predicates. It is important in which order these predicates are tested. Usually, we want to start with the most restrictive predicate first, however, we also need to consider the cost of evaluating the predicate, in particular the access cost for the data. Hence, choosing the order in which to test conditions is an important tuning problem.

**Access Cost:** The cost for accessing an attribute of $R^*$ defined by a path $P$ not only depends on the length of the path
but also on the size of the involved physical columns. Namely, small columns may fit in cache while large columns will cause additional cache faults. For example, in TPC-H it makes little difference whether one accesses an attribute directly stored with a supplier or stored with the nation of the supplier – the 25 entries of the nation dimension table easily fit in cache.

**Loop Unrolling:** In most queries, the main part of the running time is spent in one (most inner) loop iterating over data and processing it. Usually, the instructions within the loop depend on each other and must be executed in a certain order which may cause lost processor cycles. By unrolling the loop we allow the compiler to reorder the instructions potentially allowing a higher number of instructions per cycle. Note that the unroll count must not be too large as the code of the loop should still fit into L1 cache (we use an unroll count of 8).

**SIMD Instructions:** On the first glance, decision support queries are highly data parallel and thus should allow parallel operation on multiple tuples using the single-instruction-multiple-data features of modern processors. Thus we tried this in some cases, however so far without noticeable performance improvements. One reason might be that precompiled tuple by tuple processing does not benefit as much as vectorized execution from SIMD [30].

**F. Aggregation**

Decision support queries frequently reduce large data sets by commutative and associative operations like counting, summing, minimizing, or maximizing over a large number of values. We differentiate between two cases in TPC-H.

1. For the majority of the queries, the number of results produced is so small that we can afford to compute a local result table for each thread participating in the computation. Often, this result will even fit into the L1 cache. Only at the end, the local results are combined into the overall result. This way we avoid ruinous synchronization overhead. To receive the overall result, we combine two local results into a single result for pairs of remaining results (in parallel). Thus, the number of parallel steps to receive the overall result is \(\lceil \log_2(\text{number of threads}) \rceil\) when we use our scheduling framework NBB.

2. If the results do not fit into L2 cache, we use a different method (Queries 10 and 15 in TPC-H). We divide the final result table into different parts such that each of the parts fits into L2 cache. In our implementation we use 256 parts and divide the parts based on the most significant bits of the (hash) key. The aggregation is done in two passes.

   First, each thread splits its part of the input into the 256 parts. Memory mangement for the required arrays of tuples uses thread-local memory pools. In the second pass, each part is entirely assigned to one thread which can produce the corresponding part of the result table by local computations. The approach eliminates the need for inter-processor synchronization and greatly improves cache locality. See also the aggregation from Cieslewize et al. [31] or Manegold et al. [32] using a similar technique to speed up joins.

**G. Using Inverted Indices**

Suppose we are performing computations on tuples from a closure relation \(R^*\) which involves a highly selective predicate \(P\) depending only on a closure relation \(U^*\) and we have an index inverting the path from \(R\) to \(U\). Then rather than scanning \(R\) and performing expensive indirect accesses to the attributes from \(U^*\), we can scan \(U\) and use the index to retrieve only those tuples of \(R\) for which the predicate \(P\) is fulfilled. Using inverted indices exhibits parallelism at two levels of granularity – when scanning \(U\), and when scanning an inverted list for a particular tuple \(u \in U\).

Beyond this most common standard use of indices we can also combine several index results: When indices are available for several sufficiently selective predicates, we can compute the intersection of the index results directly. For inverted index data structures this is a standard operation known from information retrieval. In particular, a pairwise intersection can be computed in time proportional to the smaller set if the inverted lists are represented appropriately [33]. We can also efficiently intersect the sets of intervals we get when we have sorted relations and the mixed case.

TPC-H Query 13 is a (somewhat nonstandard) example where intersecting lists is useful: This query processes orders whose comment text does not match two words specified in the query. By intersecting the inverted indices for these two words, we can efficiently preprocess a bit pattern specifying which tuples can be ignored thus saving complex string matching operations.

**H. Preprocessing**

Suppose we are performing computations on tuples from a closure relation \(R^*\) which involves computing some predicate or value \(x\) that only depends on data in a closure relation \(U^*\) which contains far less tuples than \(R\). Then we can precompute these values cheaply storing them in a temporary column \(X\). This makes sense if evaluating \(x\) is considerably more expensive than accessing \(X\). This can be the case even for very simple computations. For example, if \(x\) is a predicate then \(X\) is just a bit array that is more likely to fit into cache. In TPC-H, this technique is particularly useful if the computation involves complicated string operations (for example in TPC-H Query 13 described above).

**I. Postprocessing**

**Top \(k\) Queries:** Many TPC-H queries output only the \(k\) largest or smallest results in sorted order. To save memory and improve cache locality, each thread periodically eliminates all but the top \(k\) of its local results. This can done in linear time. In the end we use partial sorting in order to extract the final output, i.e., the globally top \(k\) elements are identified and only those are actually sorted.

**J. Miscellaneous Functions**

We are using a fast implementation of the Boyer-Moore algorithm for string matching problems where we do not have an index [34], [35].
Most predicates on dates can be reduced to comparisons between integers. As a byproduct of our studies we found a very fast way to extract the year \( y \) out of a date represented as number of days \( d \) since Jan 1st 1970 (or any other starting date) without storing a full lookup table:

\[
y = YT[d \geq 8].\text{base} + (d\&255) > YT[d \geq 8].\text{offset}
\]

where the precomputed table \( YT[i] \) stores the year of day \( 256i \) and the number of days remaining of this year at day \( 256i \). Note that this table will fit into L1 cache. Months or days within a month can be extracted in similar ways.

IV. IMPLEMENTATION

A. Software Engineering Aspects

Our strategy is to encapsulate low level aspects of the machines like NUMA aware data layout, SIMD instructions, parallelization, thread scheduling, (de)compression, data layout etc. in easy to use generic C++ libraries. Thus a query compiler or an application programmer can generate queries without knowing about the details of the machine or about many-core programming. By recompiling one can also adapt these query codes to different platforms. Of course, for optimal performance, aspects like the order of testing predicates or the decision whether an index should be used, depends both on the machine and on the data so that some interactions are unavoidable. On the long run this problem should be solved using some auto-tuning mechanism.

B. Implementing TPC-H

We generate data using the generators defined by the TPC-H benchmark and also check the resulting rows for correctness. The ORDER table is sorted by order date. LINEITEMS are sorted by ORDER and thus also by order date. The relation PARTSUPP is sorted lexicographically first by PART and then by SUPPLIER. We have forward indices and backward indices for all edges of the schema graph shown in Figure 1. The backward index is usually implemented as an inverted index except when sorting allows a more compact representation which is the case for the bold edges in Figure 1. There is one multihop inverted index indicated by the backward arrow in Figure 1.

To allow fair comparison with other systems we comply to the official TPC-H implementation rules as far as possible. In particular, we follow the rules for data types, join indices and sorting relations (see 1.3.1 and 1.5.7 in [6]). Still, our implementation is not a full database system and misses several features like update support, ACID support and executing arbitrary SQL statements. However, we believe that it is in principle possible to add this to our system. See Section VI for further discussion.

Table I summarizes the used optimizations for each query. We also give a detailed description of the implementation of all 22 queries in the appendix, where we describe each query as a procedure in pseudocode.

```
TABLE I

<table>
<thead>
<tr>
<th>Q</th>
<th>NUMA</th>
<th>BS</th>
<th>Agg</th>
<th>Top K</th>
<th>NUMA Indexes</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.49</td>
<td>1.21</td>
<td>*</td>
<td>*</td>
<td>2FI</td>
<td>34.6</td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>FI</td>
<td>26.3</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>FI</td>
<td>23.0</td>
</tr>
<tr>
<td>4</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>I</td>
<td>12.8</td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>FI</td>
<td>40.5</td>
</tr>
<tr>
<td>6</td>
<td>1.43</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>2FI</td>
<td>37.0</td>
</tr>
<tr>
<td>7</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>2I</td>
<td>38.9</td>
</tr>
<tr>
<td>8</td>
<td>1.39</td>
<td>1.50</td>
<td>1.21</td>
<td>3FI</td>
<td>32.7</td>
<td>29</td>
</tr>
<tr>
<td>9</td>
<td>1.47</td>
<td>1.09</td>
<td>2FI</td>
<td>F</td>
<td>48.3</td>
<td>28</td>
</tr>
<tr>
<td>10</td>
<td>*</td>
<td>*</td>
<td>4.2</td>
<td>3FI</td>
<td>9.6</td>
<td>8</td>
</tr>
<tr>
<td>11</td>
<td>*</td>
<td>*</td>
<td>4.2</td>
<td>2FI</td>
<td>9.6</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>1.22</td>
<td>1.04</td>
<td>MH 2.8</td>
<td>F</td>
<td>13.2</td>
<td>30</td>
</tr>
<tr>
<td>13</td>
<td>1.10</td>
<td>*</td>
<td>I</td>
<td>F</td>
<td>37.3</td>
<td>43</td>
</tr>
<tr>
<td>14</td>
<td>1.90</td>
<td>*</td>
<td>I</td>
<td>LW 5.3</td>
<td>9.6</td>
<td>11</td>
</tr>
<tr>
<td>15</td>
<td>1.44</td>
<td>*</td>
<td>10.7</td>
<td>F</td>
<td>35.7</td>
<td>34</td>
</tr>
<tr>
<td>16</td>
<td>*</td>
<td>*</td>
<td>10.7</td>
<td>F</td>
<td>23.8</td>
<td>38</td>
</tr>
<tr>
<td>17</td>
<td>*</td>
<td>*</td>
<td>10.7</td>
<td>F</td>
<td>8.9</td>
<td>6</td>
</tr>
<tr>
<td>18</td>
<td>*</td>
<td>*</td>
<td>10.7</td>
<td>F</td>
<td>23.4</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>*</td>
<td>*</td>
<td>10.7</td>
<td>F</td>
<td>42.3</td>
<td>17</td>
</tr>
<tr>
<td>20</td>
<td>*</td>
<td>*</td>
<td>10.7</td>
<td>F</td>
<td>31.5</td>
<td>22</td>
</tr>
<tr>
<td>21</td>
<td>1.31</td>
<td>*</td>
<td>*</td>
<td>F</td>
<td>26.3</td>
<td>12</td>
</tr>
<tr>
<td>22</td>
<td>1.80</td>
<td>*</td>
<td>*</td>
<td>F</td>
<td>21.6</td>
<td>33</td>
</tr>
</tbody>
</table>

NUMA aware scheduling. Block Summaries, advanced Aggregation. Forward join index. Indexes=(Inverted join index, Multi-Hop index, Word Index).
```

V. EXPERIMENTS

When comparing our running times across different scale factors, we see that our implementation scales close to linear with the scale factors, i.e. we scale not only with the number of cores but also with the amount of data.

We now report results of our implementation using g++ version 4.5 with optimization level -O3. The machine used has 4 sockets each equipped with an Intel Xeon X7560 eight-core processor running at 2.27 GHz. Total RAM size is 256 GB. Peak memory bandwidth with the stream benchmark [36] is 50 GB/s. To make a fair comparison, we use a machine which has almost equal processing power as the machines from the published benchmark results. We provide scaled numbers by
We see 10–90% improvement on average.

There are two queries that need aggregations with a large result set, query 10 and 15. Here our advanced aggregation algorithm yields a factor of 4.2 and 10 speedup in comparison to the simple version where each thread holds its local hashmap which is merged into one big hashmap in the end.

Table II compares our running times with the best published results for TPC-H for scale factors 100 and 300 on a single machine. The current record holder for the TPC-H Benchmark for scale factor 100 and 300 are both recent results using VectorWise 2.0.1 [10] as DBMS, running on two Intel Xeon E5-2690 processors at 2.9 GHz. According to SPECint2006, the used hardware by VectorWise is almost equal to ours (670 to 710). The next best published result for SF300, not listed in the table, uses VectorWise 1.6 on an Intel Xeon E7-8837 (2.67 GHz) having 32 core system with a SPECint2006 of 748, which is even closer to our machine which can be used as our competitor benchmark.

In fact, using the official results, we are sure that the competing DBMS is professionally tuned and uses suited hardware.

On average we are a factor of 34 faster than the record holder on SF300. At SF100 we are 26 times faster than the best other system. Note that the competing machines have sufficient memory to keep all the processed data in main memory. Time spent parsing SQL and generating an execution plan is less important on such data sizes, so this can only explain a small part of the performance difference. Also note that the current next best TPC-H result with a different DBMS has factor 3-4 lower performance numbers.

Figure 3 reports on a crude approximation of the TPC-H running times with the best published results for TPC-H for scale factors 100 and 300 on a single machine. The current record holder for the TPC-H Benchmark for scale factor 100 and 300 are both recent results using VectorWise 2.0.1 [10] as DBMS, running on two Intel Xeon E5-2690 processors at 2.9 GHz. According to SPECint2006, the used hardware by VectorWise is almost equal to ours (670 to 710). The next best published result for SF300, not listed in the table, uses VectorWise 1.6 on an Intel Xeon E7-8837 (2.67 GHz) having 32 core system with a SPECint2006 of 748, which is even closer to our machine which can be used as our competitor benchmark.

In fact, using the official results, we are sure that the competing DBMS is professionally tuned and uses suited hardware.

On average we are a factor of 34 faster than the record holder on SF300. At SF100 we are 26 times faster than the best other system. Note that the competing machines have sufficient memory to keep all the processed data in main memory. Time spent parsing SQL and generating an execution plan is less important on such data sizes, so this can only explain a small part of the performance difference. Also note that the current next best TPC-H result with a different DBMS has factor 3-4 lower performance numbers.

Figure 3 reports on a crude approximation of the TPC-H running times with the best published results for TPC-H for scale factors 100 and 300 on a single machine. The current record holder for the TPC-H Benchmark for scale factor 100 and 300 are both recent results using VectorWise 2.0.1 [10] as DBMS, running on two Intel Xeon E5-2690 processors at 2.9 GHz. According to SPECint2006, the used hardware by VectorWise is almost equal to ours (670 to 710). The next best published result for SF300, not listed in the table, uses VectorWise 1.6 on an Intel Xeon E7-8837 (2.67 GHz) having 32 core system with a SPECint2006 of 748, which is even closer to our machine which can be used as our competitor benchmark.
near linearly both with S and SF indicating efficient use of the system. For SF1, running times remain almost constant until \( S > 10 \) which indicates that only with a sufficient number of streams we get enough work to saturate our machine.

Compared to the current record holder VectorWise 2.0.1 which needs 240 s for SF100 and \( S = 11 \) streams, we need only 8.8 s. This factor 27 improvement can only partially be attributed to the missing update operations in our measurements.

**Small Machine Experiments**

For smaller scale factors (1 GB and 10 GB of data) there is too little work to warrant use of our large 32 core machine. We therefore present experiments with regular desktop sized machines for these cases, mostly with quad-core processors. This also allows a more realistic comparison with other academic implementations which usually use both small inputs and small machines. Table III shows running times for all 22 TPC-H Queries for several systems. Our results are given in ms and the results of other systems are given in multiples of our running time. For VectorWise and MonetDB we were able to exclude hardware variations by running them on our own machine. Running times of HIQUE [12], HyPer [13] and MonetDB/X100 [3] are taken directly from the respective publications. PostgreSQL and MySQL running times originate from [http://monetdb-xquery.org/SQL/Benchmark/TPCH](http://monetdb-xquery.org/SQL/Benchmark/TPCH) (2008). As the used hardware varies we provide SPECInt_rate2006 (all CPUs) and SPECint2006 (single CPU) performance numbers for the used machines (details in section below). Note that HyPer (*) uses a modified schema and modified queries (TPC-CH [14]).

The absolute running times of our implementation are in the range of milliseconds, for SF1 2.5 ms and for SF10 25 ms on average. Other systems have a factor of 100 to 1000 longer running times, PostgreSQL and MySQL even higher. For small scale factors, SQL parsing and query interpretation overhead becomes more relevant which is missing in our implementation, but also in MonetDB/X100 (hand crafted internal query object), HyPer and HIQUE (compile times excluded), so our results look also strong when comparing directly with systems executing precompiled query specific code.

Our average speedup with 4 cores is 2.8 and 3.2 for SF1 and SF10, respectively. As parallelization overhead becomes significant for such small runtimes and our peak memory bandwidth is only 11 GB/s according to the stream benchmark [36], speedups look decent. When omitting parallelization and running single threaded, our implementation has still significantly lower runtimes than the other systems, also after scaling with SPECint2006 (single CPU) numbers.

1 We an Intel i5-750 quad-core CPU running at 2.66 GHz with 8 GB RAM for our system, VectorWise and MonetDB. MonetDB/X100 uses a single-core Itanium 2 with 12 GB RAM, the SPECint2006 number is approximated from recent Itanium 2 processors. PostgreSQL and MySQL use an Intel Core 2 Q6600 quad-core CPU with 8 GB RAM. HIQUE uses an Intel Core 2 Duo 6300 dual-core CPU with 2 GB RAM. HyPer uses an Intel Xeon X5570 quad-core CPU with 64 GB RAM.

**VI. Conclusions and Future Work**

We have demonstrated how the actual computations involved in decision support queries such as in TPC-H can be implemented one to two orders of magnitude more efficiently than in previous systems. Single pass execution, column-based in-memory storage, modern many-core processors with huge main memories and highly tuned algorithms give an overall performance that basically allows instantaneous results on everything that fits in memory. This opens up the possibility to use decision support systems very flexibly without restricting to prearranged aggregations like data-cubes. These queries can now also be used routinely in a wide range of applications without producing unbearable computational costs and energy consumption.

What we describe here would be directly useful for a static database queried with a software library that is considerably lower level than SQL. To make the results more generally useful, we are currently working on a query compiler from SQL to C++. Initial results for 10 out of the 22 TPC-H queries look very promising: Using the Clang C++ Compiler [5] and just-in-time compilation of the generated byte code (LLVM IR), we need only 10–20 ms compile time and achieve similar performance as the hand-written code if all the algorithmic measures are implemented. In particular, parallelization issues are completely handled by query-independent precompiled libraries. Query plan selection is easier than in classical DBMS since we basically have to decide which indices should be used and the remaining query plan can be derived from that.

Updates on a database, i.e., insertions and deletions on a database can be buffered in a small dynamic \( \Delta \)-database, see e.g. update handling in VectorWise [37] supporting also transactions. The \( \Delta \)-database and the main database can then be combined to one view including all data for a query, but for many queries it might be more efficient to compose them into separate queries to the main and \( \Delta \)-database, respectively. From time to time the main database and the \( \Delta \)-database have to be merged to keep the \( \Delta \)-database small (see Krueger et al. [38]).

To allow interactive latencies on even larger databases, clusters of many-core servers will be needed for the time being. We expect that this comes at relatively low overheads as long as we can afford to replicate those smaller relations that are accessed randomly. We can then use the same approach as in our NUMA approach and distribute the large relation evenly over the cluster nodes which are then scanned in parallel. For allowing larger databases at low cost, we may be able to use a hybrid system where less frequently used columns are stored on (solid-state) disk.

It is a tempting idea to achieve even higher throughput using graphics processing units. While we believe that this should be possible in principle, it is currently severely limited by memory sizes of GPUs. TPC-H type SQL-queries involve too little work to warrant moving data between main memory and GPU memory (which is expensive). Since even small PCs would be fast enough to achieve interactive speed on such small inputs,
it seems that GPUs are most interesting for small databases that either require a very large throughput of complex queries or have latency requirements in the millisecond range for some reason.

Acknowledgments

We would like to thank Anja Bog, Roman Dementiev, Franz Färber, Jens Krüger, Wolfgang Lehner, Hasso Plattner, Johannes Singler and Frederik Transier for fruitful discussions and suggestions.

References


Here we give a detailed description of the implementation of all 22 TPC-H queries. Each of the queries is described by a procedure in pseudocode (Q1 - Q22). The legend in Table IV gives a brief introduction to the syntax.

// use block summaries of Lineitem
Procedure Q1 ([date])

<table>
<thead>
<tr>
<th>Procedure name</th>
<th>The first letters of a column name are an abbreviation of the table name. For example, columns staring with l are located in lineitem (cf. TPC-H specification [6])</th>
</tr>
</thead>
<tbody>
<tr>
<td>l</td>
<td>name()</td>
</tr>
<tr>
<td>l</td>
<td>name.list()</td>
</tr>
<tr>
<td>l</td>
<td>name.range()</td>
</tr>
<tr>
<td>PForeach</td>
<td>Parallel loop iterating over the given set. Usually, the result is computed locally in each thread and reduced to one result in the end. Numa PForeach takes advantage of NUMA indexes (see Section III-C).</td>
</tr>
</tbody>
</table>

// use inverted index of p.size
Procedure Q2 ([size],[type])

// use block summaries of Lineitem. A range in the lineitem table corresponds to a range in the order table.
Procedure Q3 ([date],[segment])

// order is sorted by o.orderdate
Procedure Q4 ([date])

APPENDIX


APPENDIX

Here we give a detailed description of the implementation of all 22 TPC-H queries. Each of the queries is described by a procedure in pseudocode (Q1 - Q22). The legend in Table IV gives a brief introduction to the syntax.

Procedure Q1 ([date])

// use block summaries of Lineitem

// use inverted index of p.size

// use block summaries of Lineitem. A range in the lineitem table corresponds to a range in the order table.

// order is sorted by o.orderdate

Procedure Q4 ([date])
Array $t$ ← lookup table, true for nationkeys in region [region]
$[a, b] ← range in which $\forall x \in [a, b]:$
$\exists (\text{orderdate}(x) < \text{date}) + 1$ year

**For** each customer $i$ do
    $n ← \text{nationkey}(i)$ // current nationkey
    if $t[n]$ then
        foreach $j$ in $\text{custkey}.\text{list}(i)$ do
            if $j$ in $[a, b]$ then
                for $l$ in $\text{orderdate}.\text{range}(j)$ do
                    if $\text{nationkey}(L) = n$ then
                        res[$n] +=
                        $L \cdot \text{extendedprice}(j) \cdot (1 - L \cdot \text{discount}(j))$

    Procedure Q5 ([date], [region])

**For** each lineitem $i$ with $\exists (\text{date} \leq \text{shipdate}(i) < \text{date}) + 1$ year
    // start with highest selectivity
    if $\text{quantity}(i) < \text{quantity}$ then
        if $\text{discount}(i) = 0.01 \leq \text{discount}(i) \leq \text{discount}(i) + 0.01$ then
            res += $\text{extendedprice}(i) \cdot (1 - \text{discount}(i))$

    Procedure Q6 ([date])

**For** each customer $i$ do
    $key ← i$ // key for partner nation
    if $\text{nationkey}(i) = \text{nation1}$ then
        $key ← \text{nation2}$
    else
        $key ← \text{nation1}$
    continue with next customer
    foreach $j$ in $\text{partsupp}.\text{list}(i)$ do
        for $l$ in $\text{orderdate}.\text{range}(j)$ do
            if $\text{shipdate}(l)$ in $[1995/01/01, 1996/31/12]$ then
                res[$key, \text{extract_year}(\text{shipdate}(l))$] +=
                $\text{extendedprice}(l) \cdot (1 - \text{discount}(l))$

    Procedure Q7 ([nation1], [nation2])

Array $t$ ← lookup table, true for nationkeys in region [region]
$[l, r] ←$ lineitem range with orderdate year 1995 or 1996
$\forall x \in [l, r]:$ $\exists (\text{orderdate}(x) \geq [1995/01/01, 1996/31/12])$

**Numa**
**For** each part $i$ with $\exists \text{type}(i)$
    foreach $j$ in $\text{partsupp}.\text{list}(i)$ do
        if $j$ in $[l, r]$ then
            $\alpha ← \text{orderkey}(j)$ // order index
            if $\text{nationkey}(o) = \text{true then}$
                $\exists (\text{year}(\text{orderdate}(o)) =$
                $\text{nationkey}(\text{year}) +=$
                $\text{extendedprice}(i) \cdot (1 - \text{discount}(i))$
            else
                $\text{resall} \text{year} +=$
                $\text{extendedprice}(i) \cdot (1 - \text{discount}(i))$

    res[nation/resall] (per year) is the result mkShare.
    Procedure Q8 ([type], [region])

$\text{res} ← \text{word}$ in $\text{p}.\text{name}.\text{wordlist}$ matching $\%[\text{color}|\% \text{Numa}
**For** each part $i$ with $\exists \text{name}(i)$ = $\text{word}$
    foreach $j$ in $\text{partsupp}.\text{list}(i)$ do
        $y ← \text{extract_year}(\text{orderdate}(\text{partsupp}(j)))$
        $n ← \text{nationkey}(\text{partsupp}(j))$
        $\text{res}[n, y] += \text{extendedprice}(j) \cdot (1 - \text{discount}(j))$ $\text{ps}.$
        $\text{suppcost}(\text{partsupp}(j)) \cdot \text{quantity}(j)$

    Procedure Q9 ([color])

**PFor** each order $i$ with $\exists \text{orderdate}(i) < \text{date} + 3$ months do
    $c ← \text{custkey}(i)$ // custkey index
    for $j$ in $\text{orderkey}.\text{range}(i)$ do
        if $\text{returnflag}(j) = R$ then
            Aggregate ($\text{extendedprice}(j) \cdot (1 - l \cdot \text{discount}(j))$)
            by $\text{custkey}(j)$. Use parallel aggregation algorithm as (we have many result rows).
        Find the top 20 customers from aggregation (in parallel).
        An alternative implementation starting at customer table does not require an aggregation but is a factor of 4.2 slower.

    Procedure Q10 ([date])

**Numa**
**For** each partsupp $i$ with $\exists \text{nation} = \text{nation}$ do
    all sum $\text{ps}.\text{suppcost}(i) \cdot \text{ps}.\text{quality}(i)$
    frac ← (reduced all sum) / [fraction] // threshold

**Numa**
**For** each partsupp $i$ with $\exists \text{nation} = \text{nation}$ do
    Sum $\text{ps}.\text{suppcost}(i) \cdot \text{ps}.\text{quality}(i)$ to $\text{sumwart}$ for all
    consecutive $i$ with the same $\text{partkey}(i)$. After the last such $i$:
    if $\text{sumwart} > \text{frac}$
        then add $\text{ps}.\text{partkey}(i), \text{sumwart}$ to result

Above, we use a multi hop index to find the partsupp rows for which the corresponding supplier has the given nationkey. The index results in a speedup of 2.8.

    Procedure Q11 ([nation], [fraction])

**PFor** each lineitem $i$ with $\exists \text{receiptdate}(i) < \text{date} + 1$ year do
    if $(\text{numcommittime}(i) < \text{receiptdate}(i)) \wedge$
    $(\text{shipdate}(i) < \text{commitdate}(i)) \wedge$
    $(\text{shipmode}(i) \in \{\text{shipmode1}, \text{shipmode2}\})$ then
        important ← 0 // has priority
        if $(\text{orderpriority}(i) \text{orderkey}(i) \in$
        $(1 - \text{URGENT}) / 2 - \text{HIGH})$ then
            important ← 1
            res[$\text{shipmode}(i), \text{important}] += 1

    Procedure Q12 ([date], [shipmode1], [shipmode2])

$v ←$ Bitvector with length of customer table, initialized with 0
$l_1 ← \text{comment word index list with w like [word1]}$%
$l_2 ← \text{comment word index list with w like [word2]}$

**PFor**
**i** in $[l_1 \cap l_2]$ do
    if $\text{word1}(i)$ matches $\% \text{word1}$ $\% \text{word2}$ then
        $[i] ← 1$

**PFor** each customer $i$ do
    count ← 0 // per customer
    foreach $j$ in $\text{custkey}.\text{list}(i)$ do
        if $j[4] = 0$ then count += 1
        res[count] += 1

    Procedure Q13 ([word1], [word2])

**PFor** each lineitem $i$ with $\exists \text{shipdate}(i) < \text{date} + 1$ year do
    sumall += $\text{extendedprice}(i) \cdot (1 - l \cdot \text{discount}(i))$
    // match check is only a simple range check (sorted dictionary)
    if $\text{p}-\text{type}(i) \text{partkey}(j)$ matches PROMO% then
        sum_match += $\text{extendedprice}(i) \cdot (1 - l \cdot \text{discount}(i))$

    Procedure Q14 ([date])

**PFor** each lineitem $i$ with $\exists \text{shipdate}(i) < \text{date} + 3$ months do
    Aggregate ($\text{extendedprice}(i) \cdot (1 - l \cdot \text{discount}(i))$)
    by $\text{supppkey}(i)$. Use parallel aggregation algorithm (many result rows).
    Find supplier(s) with highest revenue in parallel.

    Procedure Q15 ([date])
\[
t \rightarrow \text{Array with length of part table. For a row } i:
\]
\[
t[i] = \begin{cases} 
\text{if } \{ \text{parttype}(i), \text{brand}(i), \text{size}(i) \} \text{ invalid } \rightarrow -1 \\
\text{else: index for } \{\text{type, brand, size}\} \text{ combination}
\end{cases}
\]

Compuite \( t \) in parallel.

PForeach supplier \( i \) do
\[
\begin{cases} 
\text{if } \text{s_comment}(i) \text{ does not match }
\end{cases}
\]
\[
\% \text{Customer Complaints}\% \text{ then}
\]
\[
\text{foreach } j \text{ in } \text{s_lo_p.suppprkey(list)(j)} \text{ do}
\]
\[
\quad \text{res}[j]++ = 1
\]

Procedure Q16 \([\text{[brand],[type],[size[1..8]]]}\)

// intersect index lists

PForeach part \( i \) with \( \text{p_container(i)} = \) [container] and \( \text{p_brand(i)} = \) [brand] do
\[
q \leftarrow 0 \quad \text{if quantity threshold}
\]
\[
\text{foreach } j \text{ in } \text{p_lo_p.partkey.list(i)} \text{ do}
\]
\[
\quad q += \text{l_quantity}(j)
\]
\[
q \leftarrow 0.2 \cdot q / \text{p_lo_p.partkey.list(i)}
\]
\[
\text{foreach } j \text{ in } \text{p_lo_p.partkey.list(i)} \text{ do}
\]
\[
\quad \text{if } \text{l_quantity}(j) < q \text{ then}
\]
\[
\quad \text{res} += \text{l_extendedprice}(j)
\]

res = res / 7.6

Procedure Q17 \([\text{[brand],[container]}]\)

PForeach order \( i \) do
\[
q \leftarrow 0 \quad \text{if quantity of order } i
\]
\[
\text{foreach } j \text{ in } \text{o_orderkey.range(i)} \text{ do}
\]
\[
\quad q += \text{l_quantity}(j)
\]
\[
\text{if } q > \text{[quantity]} \text{ then}
\]
\[
\text{add order } i \text{ and sum } q \text{ to res}
\]
\[
\text{if } \text{size(res)} > 1024 \text{ then keep only top 100}
\]

Procedure Q18 \([\text{[quantity]}]\)

PForeach part \( i \) do
\[
\text{if } \{\text{brand}(i), \text{container}(i), \text{size}(i)\} \text{ valid then}
\]
\[
\text{foreach } j \text{ in } \text{p_lo_p.orderkey.set(i)} \text{ do}
\]
\[
\qquad \text{if } \text{l_quantity}(j) \in \{1,11\} \text{ or } [10,20] \text{ or } [20,30] \text{ depending on } \text{p_brand(i)} \land \text{shipmode}(j) = \text{'AIR'} \land \text{'}} \text{ AIR REG'} \text{ does not exist}
\]
\[
\qquad \text{hiprunstruct}(j) = \text{'}DELIVERINPERSON' \text{ then}
\]
\[
\qquad \text{res} += \text{l_extendedprice}(j) \cdot (1 - \text{discount}(j))
\]

Procedure Q19 \([\text{[quantity[1..3]], [brand[1..3]]]}\)

PForeach part \( i \) do
\[
\text{if } \{\text{brand}(i), \text{container}(i), \text{size}(i)\} \text{ valid then}
\]
\[
\text{foreach } j \text{ in } \text{p_lo_p.orderkey.range(i)} \text{ do}
\]
\[
\quad \text{if } \text{s_nation}[\text{p_lo_p.suppprkey}(j)] = \text{s_nation}[\text{p_lo_p.suppprkey}(i)] \text{ then}
\]
\[
\quad \text{q} \leftarrow 0 \quad \text{if threshold for part supp}
\]
\[
\quad \text{foreach } k \text{ in } \text{p_lo_p.partkey.range(j)} \text{ do}
\]
\[
\qquad \text{if } \text{[date]} \leq \text{lshipdate}(k) < \text{[date]} + \text{1 year}
\]
\[
\qquad \text{then}
\]
\[
\qquad \quad q += \text{l_quantity}(k)
\]
\[
\text{if } q > 0 \land \text{ps_availqty}(j) > q/2 \text{ then}
\]
\[
\text{add supplier } \text{p_lo_p.suppprkey}(j) \text{ to result}
\]

Procedure Q20 \([\text{[color],[nation],[date]}]\)

sum, count += 0 \quad \text{if help to compute average}

PForeach customer \( i \) do
\[
\text{if } \text{c_acctbal}(i) > \text{0 then}
\]
\[
\text{if } \text{c_currencycodes}(i)[0..1] \in \{[11], [12], ... , [17]\} \text{ then}
\]
\[
\text{sum} += \text{c_acctbal}(i)
\]
\[
\text{count} += 1
\]
\[
\text{avg} = \frac{\text{sum}}{\text{count}} \quad \text{if average account balance}
\]

PForeach customer \( i \) do
\[
\text{if } \text{c_currencycodes}(i)[0..1] \in \{[11], [12], ... , [17]\} \text{ then}
\]
\[
\text{res}sum[currencycodes] += \text{c_acctbal}(i)
\]
\[
\text{res}count[currencycodes] += 1
\]

Procedure Q22 \([\text{[I[1..7]]]}\)

\text{Compute } t \text{ in parallel.}