Vectorization Based Query Processing:

1. Introduction:

The query execution engine plays an important role in database system performance. The effectiveness of query execution engines in the hardware level is determined by their ability to find enough independent work and exploit the parallel execution capabilities of the modern CPUs [1]. The processing model in database management systems defines how the system executes a query plan in the execution engine. Thus, to gain the maximal throughput from the CPU and to utilize the enormous hardware developments and parallelism in the past decade, most modern database systems have adopted vectorization based query processing in their execution engine. This state-of-the-art query processing paradigm yields faster execution speed by adopting a policy of column-wise execution and operating in a batch of tuples instead of a single tuple. Thus, working on a batch gives the database systems an opportunity to utilize Single Instruction Multiple Data (SIMD) architecture in the CPU where a single operation can be applied to a batch of input and multiple results can be obtained at once. Moreover, executing on SIMD architecture reduces the overhead caused by costly branches and function calls, and stores data in CPU cache, reducing the main-memory traffic (mostly useful in multicore systems).

Due to these advantages, most of the modern database systems including Apache Spark, Apache Hive, Vectorwise [4,5], DB2 BLU, Quickstep, columnar SQL server have adopted vectorization based query processing. There are also vectorized versions of relational databases which adopt the iterator model of query processing by default. Some of the examples include the vectorized MySQL implemented by MonetDB/X100 [1], vectorized model of postgres have been implemented by Citus [15,16,17], the SQL server introduced vector extensions in 2016: Advanced Vector Extensions (AVX 2), Streaming SIMD Extensions 4 (SSE 4). The SIMD vectorization and the vectorized model of query processing is also being used in research relating to query compilation strategies [2, 3]. There is also some ongoing research to speed up the performance of vectorization based query execution through the use of efficient algorithms by Vectorwise and Hyper [4,5,6].

Vectorization based query processing is mostly being used in computer intensive application areas like Online Analytical Applications (OLAP), decision support and multimedia retrieval [1, 2, 3]. This can also be scaled up to multiple machines due to the use of columnar stores of data, which can further increase the speed of query execution in case of large data [1].

2. Background:

Before the publication of MonetDB/X100 paper in 2005 and advent of vectorization model for query processing in 2005, most database systems tend to be designed imprecisely for utilizing modern CPU architectures thereby inhibiting compilers from using their most performance critical

optimization techniques and resulting in low instructions per cycle in compute-intensive application areas [1]. This is because almost every relational and NoSQL database management systems including MySQL, PostgreSQL, SQLite, Oracle and mongoDB adopted an iterator model for processing queries (also called pipeline and volcano model), in which each next() call returns only a single tuple resulting in an interpretation overhead and significant time in evaluating the query plan rather than actually calculating the result [2].

Apart from vectorization and iterator based query processing, materialization model has also been used especially for Online Transaction Processing (OLTP) applications [7,8,9,10]. In this model, a query is evaluated by creating and storing each temporary result and then, passing it as an argument to the next operator [12]. The overall cost of query processing is increased with this model as each temporary materialized result is written to disk. This model is used commonly in OLTP workloads as queries only need to access a small number of tuples at a time and there will be fewer function calls and lower execution/coordination overhead in OLTP using a materialized model [11]. However, as the large intermediate results are produced, this method is not suitable for OLAP queries. Some of the systems using materialized models include Teradata, VoltDB and HyRise.

Modern CPUs are designed in such a way that if enough independent work is provided to them, they can exploit parallelism and produce highest throughput. As there has been tremendous development in the hardware systems, the difference between the CPU executing in its minimal throughput and the full throughput is significant [1]. Based on these findings, researchers at CWI Amsterdam introduced a new database management system called MonetDB/X100. This utilizes vectorwise execution and uses SIMD architecture in the CPU. On the surface, the vectorized model resembles a classical iterator-style engine, however each invocation of next() returns a batch of data [11].

3. Topic Explanation:

This section is organized as follows: Section 3.1 provides brief information about vectorization. In Section 3.2, vectorization and SIMD architecture have been discussed. Section 3.3 provides information about the inner workings of CPU revolving around vectorized execution and in section 3.4, vectorization based query processing models have been discussed.

3.1 Vectorization

Vectorization is the process of converting the scalar implementation of the algorithm working on only one data at a time to a vector implementation processing multiple data at once.

If we have 42 cores and each core has a SIMD register of width 8, the overall speedup is the product of speedup due to multiple cores and the speedup due to vectorization, i.e, 336x [11].

There are 3 ways in which CPU can perform vectorization [11]:

i. **Automatic Vectorization**

The compiler identifies when instructions within a loop can be converted to vectorized form. However, it works for simple loops only.

ii. **Compiler Hints**
Provide hints and additional information to the compiler about the code to let it know if it is safe to vectorize.

```
void add (int *restrict A, int *restrict B, int *restrict C)
{
      for (int i =0; i <MAX, i++){
            C[i] = A[i] + B[i];
      }
}
```

In the above code, a restricted keyword has been specified to tell the compiler that arrays are distinct locations in the memory.

iii. **Explicit Vectorization**
Manually assemble data within SIMD registers using CPU intrinsics to perform vectorized instructions.

```
void add(int *X, int *Y, int *Z)
{
      __mm128i *vecX = (__m128i*)X;
      __mm128i *vecY = (__m128i*)Y;
      __mm128i *vecZ = (__m128i*)Z;
      for (int i=0; i<MAX/4; i++) {
            _mm_store_si128(vecZ++,
            _mm_add_epi32(*vecX++, *vecY++));
      }
}
```

3.2 Vectorization and SIMD architecture

One of the many improvements that CPU and hardware systems have incorporated in the past decade is the adoption of wider SIMD registers [1]. Due to this, processing one operation on multiple pairs of operands at once has been possible resulting in faster result processing and improved disk utilization. In case of query execution, the data is divided into vectors and fed to SIMD, which applies a single operand to all the elements inside the vector at once and results in a single output vector. Equation 1 and 2 shows how vectorization applies single operand to all the elements inside the vector at once [11].

$$X + Y = Z \quad \textbf{(1)}$$

$$[x1 \;\; x2 \; ... \;\; xn] \; + \; [y1 \;\; y2 \; ... \; yn] \; = \; [x1 \; + \; y1 \;\; x2 + y2 \; ... \;\; xn \; + \;\; yn] \quad \textbf{(2)}$$

Figure 1 and 2 displays the result of applying vectorization in a 128-bit SIMD register. At first, the first batch of vector is processed. It's result is stored in a 128-bit SIMD register. Then, the second batch of vector is processed and the final result is stored.

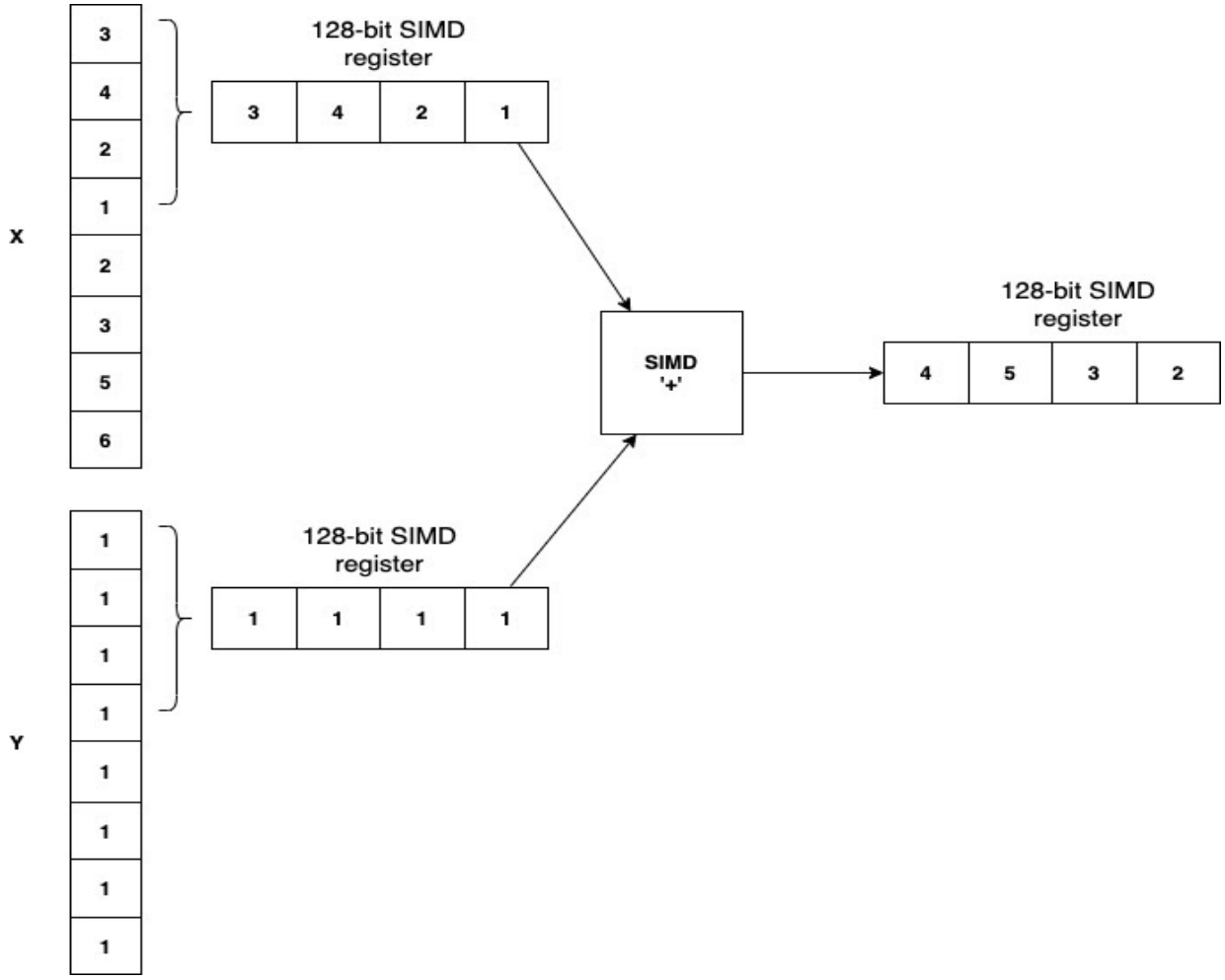

Figure 1 Applying operator '+' on the first batch at once

3.3 CPU in improving query execution speed

According to Moore's law, the number of transistors on a microchip doubles every two years though the cost of computers is halved, i.e, we can obtain an increase in performance with less price. The performance has even been increased than what Moore's law suggested as the CPU organizes instructions into pipeline stages [1]. Due to this, the work of a single CPU instruction has been divided among multiple processors and the delays from instructions that cannot complete in a single cycle are blocked and made to wait until the processor becomes free again. Moreover, multiple instructions can also run concurrently in multiple processors if each one is independent of the other [1].

Though pipelining increases speed, there are two problems associated with it [1, 11]:

(i) The instruction which is dependent upon another instruction cannot be pushed immediately into the same pipeline.

(ii) In case of IF-a-THEN-b-ELSE-c branches, the CPU must predict the result of the IF condition and decide to put *c* into the pipeline even before the condition has been executed. If *a* turns out to

be false after the CPU had predicted the opposite and placed *c* in the pipeline, the entire pipeline is flushed. Therefore, in case of long branches, this can result in a significant overhead.

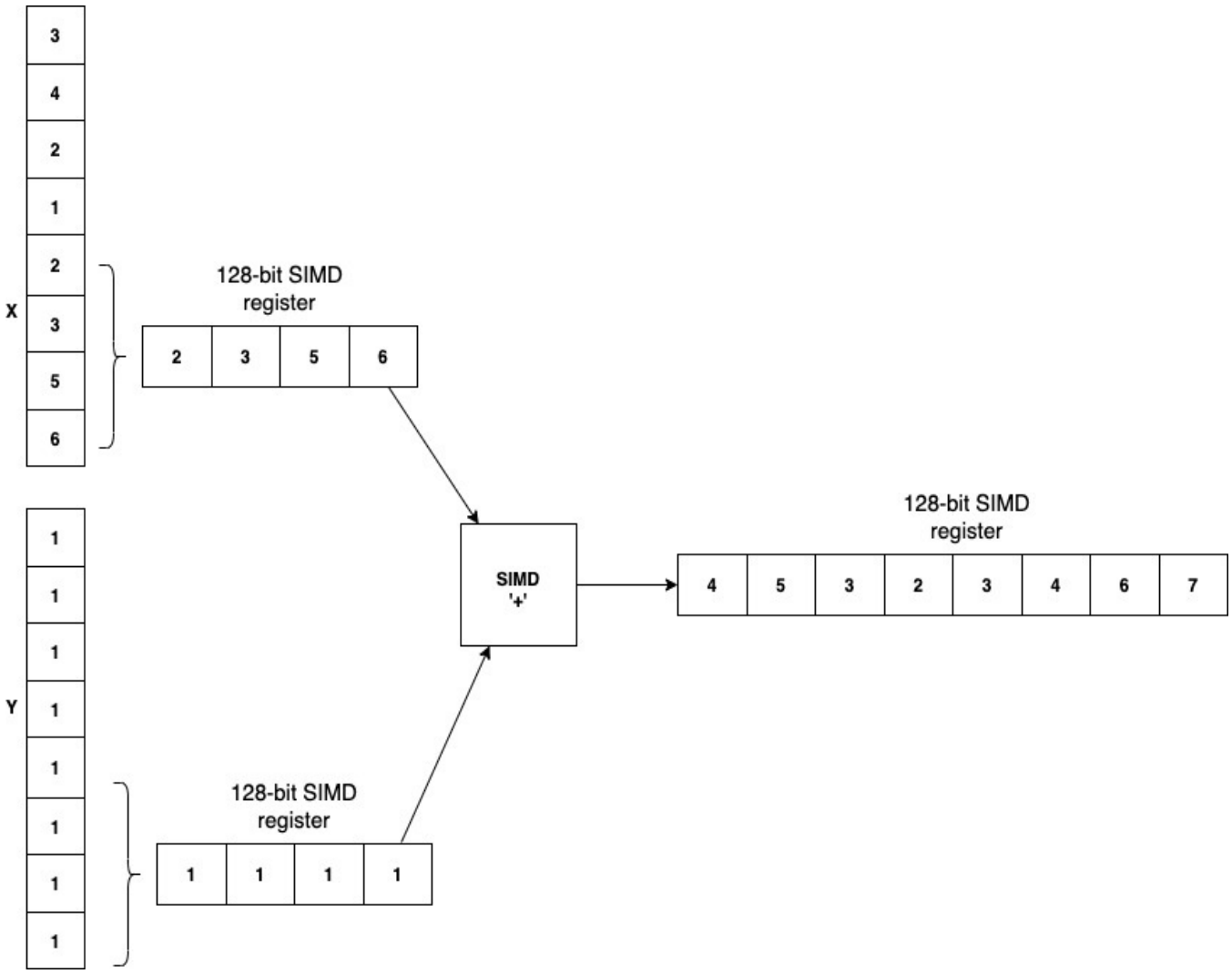

Figure 2 Applying operator '+' on the second batch and storing total result

The most widely executed branching code in DBMS is the filter operation in sequential Scan as in Algorithm 1 for Q1, however, it is impossible to predict accurately [13]. Thus, branching should be minimized as much as possible. Algorithm 2 shows how the filter operation in sequential scan can be made branchless [13].

Q1: SELECT * FROM table WHERE key >= k_{lower} AND key <= k_{upper};

Algorithm 1 Selection Scan (Scalar - Branching)

```
j ← 0                                                    ▷ output index
for i ← 0 to |T_keys_in| − 1 do
    k ← T_keys_in[i]                                     ▷ access key columns
    if (k ≥ k_lower) && (k ≤ k_upper) then               ▷ short circuit and
        T_payloads_out[j] ← T_payloads_in[i]             ▷ copy all columns
        T_keys_out[j] ← k
        j ← j + 1
    end if
end for
```

Algorithm 2 Selection Scan (Scalar - Branchless)

```
j ← 0                                                    ▷ output index
for i ← 0 to |T_keys_in| − 1 do
    k ← T_keys_in[i]                                     ▷ copy all columns
    T_payloads_out[j] ← T_payloads_in[i]
    T_keys_out[j] ← k
    m ←(k ≥ k_lower ? 1 : 0) & (k ≤ k_upper ? 1 : 0)
    j ← j + m        ▷ if-then-else expressions use conditional ...
end for              ▷ ... flags to update the index without branching
```

Figure 3 shows the performance comparison between applying branching and no branching on the code for the same queries [11]. The CPU cycles/tuple of the code with branching is smaller at the beginning due to the expensive copying of all columns performed by branchless code. The algorithm 1 is benefited with lower data initially, however, as the data in the selection space increases, it becomes slower than the branchless algorithm due to redundant prediction overhead, and gets benefited at the end again when selection space gets reduced.

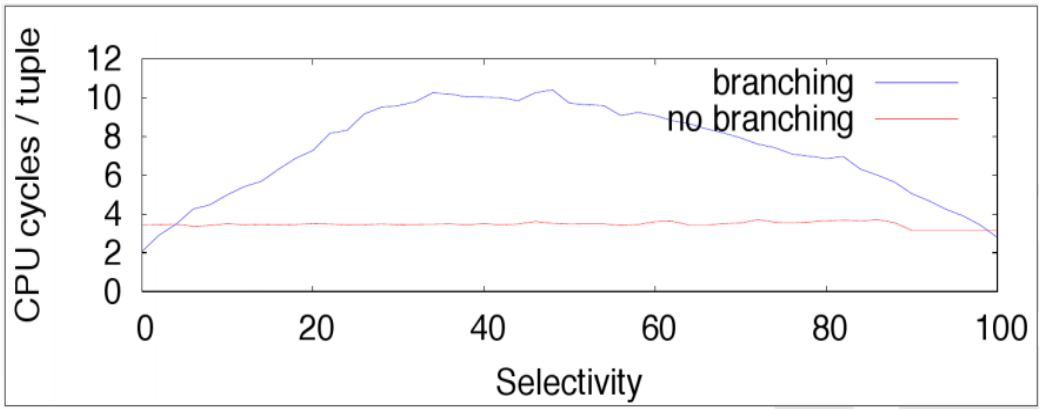

Figure 3. Performance comparison between the CPU cycles/tuple executed by branching and no branching [11]

3.4 Vectorization based query processing model with branchless code

As in the case of an iterator model, each operator implements an API that consists of open(), next() and close() methods. Each invocation of next() produces a batch of tuples, and a "pull" model approach of query evaluation is followed in which next() is called recursively to traverse the operator tree from the root downwards, with the result tuples being pulled upwards [3]. Figure 4 displays the query graph of Q2 and Figure 5 displays the corresponding branchless vectorized pseudocode at each node.

Q2: SELECT R.id, S.cdate FROM R JOIN S ON R.id = S.id WHERE S.value > 10;

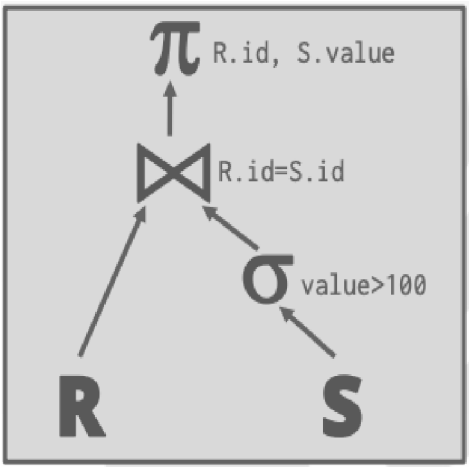

Figure 4. Query Tree of Q2

At first, the code executes in a sequence 1, 2, 3 in Figure 5. The output of 3 is returned back to the invocation of left.Next() in batch to node 2. Then, the execution transfers to 2, 4 and 5. The output of 5 is returned to the invocation of child.Next() in 4, and the output of 4 is returned to right.Next() invocation in line no. 4 in 2. Thus, the vectorized execution model follows a pull based approach where traversal takes place from up to down and the results are pulled from child to parent [3].

4. Testing:

For testing, I compared the execution speed of vectorized MonetDB/X100 and MySQL (Section 4.1). MonetDB/X100 implements both vectorized and columnar search [14]. I also observed the difference between implementing only vectorization in query execution and implementing both vectorized and columnar search (Section 4.2). This was provided by Citus which has vectorized PostgreSQL [16]. Both 4.1 and 4.2 were run on a single computer with a single processor of 2 GHz Quad-Core Intel Core i5 specification and no GPU. The concurrent execution was possible only on the 4 cores available on the processor and the 8 logical cores it creates.

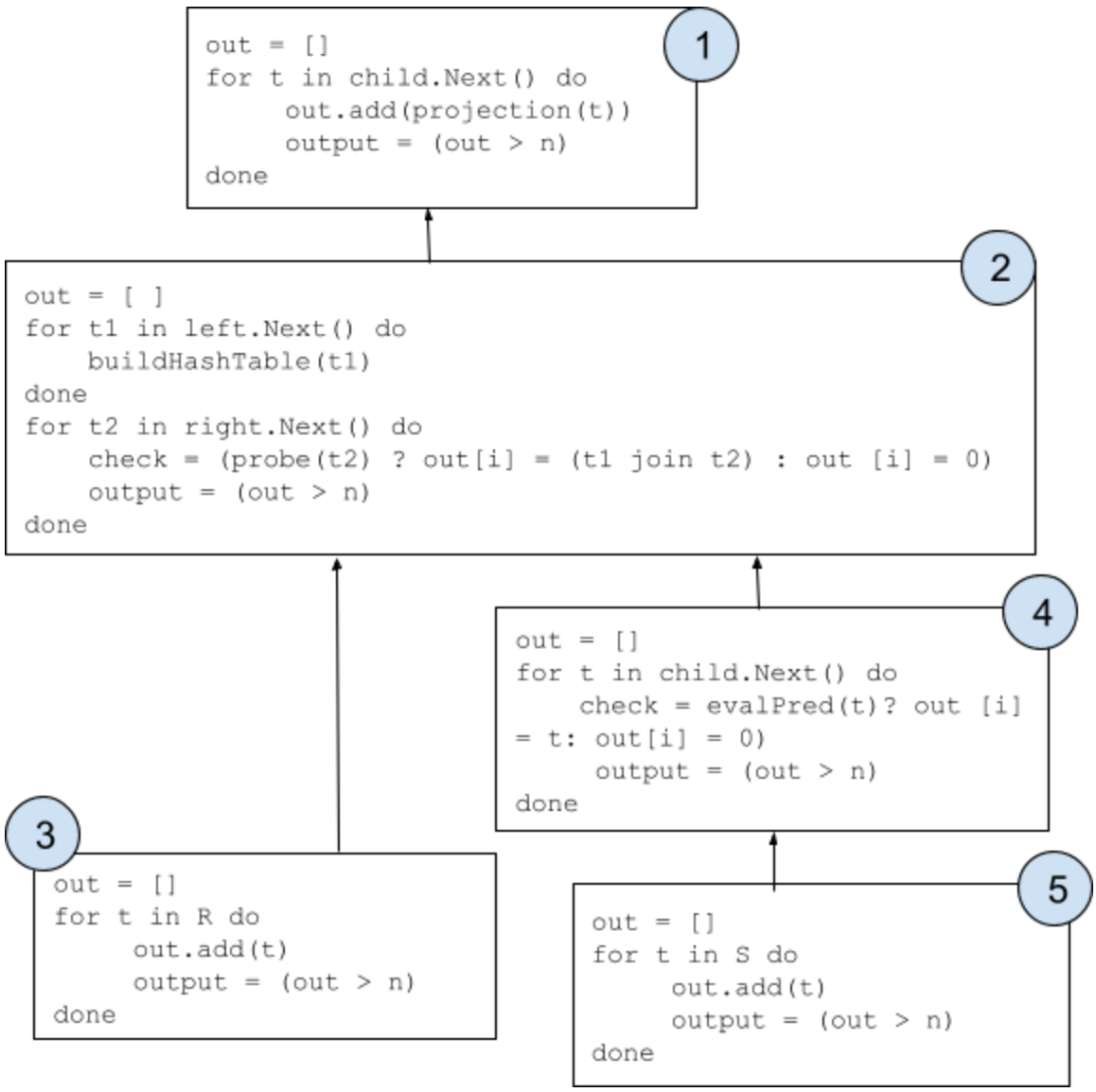

Figure 5. Branchless vectorized pseudocode for Q2

4.1 Comparison between MonetDB/X100 and MySQL

4.1.1 Setup:

For comparison between MonetDB/X100 and MySQL, firstly, I installed MonetDB using [14]. The database is based on the material published in the book J.R. Bruijin, F.S. Gaastra and I. Schaar, Dutch-Asiatic Shipping in the 17th and 18th Centuries, which gives an account of the trips made to the East and ships that returned safely (or wrecked on the way) [14]. It consists of a total of 8000 records with 8 tables and 2 views. The tables are related to craftsmen, impotenten, invoices, passengers, seafarers, soldiers, total and voyages. The voyages table is the main table to which all the other tables refer to. Similarly, the onboard_people view combines the records from all the

tables into one big table, to make records easier to access [14]. Similarly, the extended_onboard view only contains the narrowed onboard information from onboard_people.

4.1.2 Results

I executed queries (Q3 to Q8) on both MonetDB and MySQL. Most commands in MonetDB are quite similar to MySQL, however, some MySQL commands don't exist in MonetDB. As shown in Table 1, MonetDB was quite faster in executing all the queries than MySQL even in my local computer with limited parallelism.

Q3: SELECT count(*) FROM voyages;
Q4: SELECT chamber, AVG(invoice) AS average FROM invoices WHERE invoice IS NOT NULL GROUP BY chamber;
Q5: SELECT type, COUNT(*) AS total FROM onboard_people GROUP BY type ORDER BY type;
Q6: SELECT count(*) FROM impotenten;
Q7: SELECT COUNT(*) FROM voyages WHERE particulars LIKE '%_recked%';
Q8: SELECT type, COUNT(*) AS total FROM onboard_people GROUP BY type ORDER BY type;

Table 1. Comparison of execution time of MonetDB/100 and MySQL

Query	Execution Time (ms)	
	MonetDB/X100	MySQL
Q3	3248	99836
Q4	7126	22510
Q5	19106	123385
Q6	880	5537
Q7	9300	10711
Q8	7794	35060

4.2 Comparison between vectorized execution and execution incorporating both vectorization and columnar-storage in Citus and Postgres

4.2.1 Setup

Citus has provided an extension named cstore_fdw for vectorized execution, and they have provided a separate extension citus for columnar storage and compression [15]. I executed a query (Q9 to Q14) with only cstore_fdw as in [15] to check how vectorization improves the cost of the query. Then, I used both the cstore_fdw and citus extension [16] to get the combined benefit of vectorization and parallelization. For the comparison, I used the same dataset provided by citus github repository [17]. The dataset is related to customer reviews with a total of 1762504 records. However, there is only a single table and there are 12 attributes in the table (customer_id, review_date, review_rating, review_votes, review_helpful_votes, product_id, product_title, product_sales_rank, product_group, product_category, product_subcategory and similar_product_ids).

4.2.2 Results

As there are only 4 cores in my local computer and the dataset was quite small, the query execution time (Q9 to Q14) in both vectorized citus and vectorized and columnar citus was almost the same as shown in Table 3. However, there was quite a significant difference in the cost of the query plan shown in Table 2. The query plan implemented by each type is shown in Table 4. If the dataset would have been bigger and if the query was executed in multiple processors, probably the execution speed would have been drastically different as well.

Q9: SELECT count(*) FROM customer_reviews;

Q10: SELECT avg(review_votes), sum(review_votes) FROM customer_reviews;

Q11: SELECT product_group, sum(review_helpful_votes) AS total_helpful_votes FROM customer_reviews GROUP BY product_group;

Q12: SELECT review_date, count(review_date) AS review_count FROM customer_reviews GROUP BY review_date;

Q13: SELECT customer_id, review_date, review_rating, product_id, product_title FROM customer_reviews WHERE customer_id ='A27T7HVDXA3K2A' AND product_title LIKE '%Dune%' AND review_date >= '1998-01-01' AND review_date <= '1998-12-31';

Q14: SELECT width_bucket(length(product_title),1, 50, 5) title_length_bucket, round(avg(review_rating), 2) AS review_average, count(*) FROM customer_reviews WHERE product_group = 'Book' GROUP BY title_length_bucket ORDER BY Title_length_bucket;

Table 2. Cost comparison of Citus and PostgreSQL

Query	Cost		
	Citus		PostgreSQL
	Vectorized	Vectorized+Columnar Storage	
Q9	22031.31	4406.27	62835.62
Q10	27509.57	9488.88	64671.51
Q11	28583.56	10167.21	64672.39
Q12	27511.56	9490.87	65061.24
Q13	40610.08	3381.73	68343.13
Q14	117275.43	2173.88	108764.83

Table 3. Execution time comparison between Citus and PostgreSQL

Query	Execution Time (ms)		
	Citus		PostgreSQL
	Vectorized	Vectorized+Columnar Storage	
Q9	396.986	364.796	3039.372
Q10	441.452	412.275	3076.071
Q11	751.012	761.722	3246.775
Q12	576.157	568.878	3093.151
Q13	241.751	212.513	2874.039
Q14	1071.718	942.433	3225.083

Table 4. Query Plan selected by Citus and PostgreSQL

Query	Query Plan		
	Citus		
	Vectorized	Vectorized+Columnar Storage	PostgreSQL
Q9	1. Foreign scan on table 2. Aggregate	1. ColumnarScan on table 2. Aggregate	1. Parallel Seq Scan 2. Partial Aggregate 3. Gather 4. Finalize Aggregate
Q10	1. Foreign scan on table 2. Aggregate	1. ColumnarScan on table 2. Aggregate	1. Parallel Seq Scan 2. Partial Aggregate 3. Gather 4. Finalize Aggregate
Q11	1. Foreign scan on table 2. Hash Aggregate	1. ColumnarScan on table 2. Hash Aggregate	1. Parallel Seq Scan 2. Partial Hash Aggregate 3. Sort 4. Gather Merge 5. Finalize Group Aggregate
Q12	1. Foreign scan on table 2. Hash Aggregate	1. ColumnarScan on table 2. Hash Aggregate	1. Parallel Seq Scan 2. Partial Hash Aggregate 3. Sort 4. Gather Merge 5. Finalize Group Aggregate
Q13	1. Foreign scan on table	1. ColumnarScan on table	1. Parallel Seq Scan 2. Gather
Q14	1. Foreign scan on table 2. Hash Aggregate 3. Sort	1. ColumnarScan on table 2. Hash Aggregate 3. Sort	1. Parallel Seq Scan 2. Partial Hash Aggregate 3. Sort 4. Gather Merge 5. Finalize Group Aggregate

5. Conclusion

From this project, I learned how CPU handles if-then-else branches and the prediction overhead involved with it. I also learned how vectorization and SIMD architecture are related, the increase in speed if columnar storage is used instead of row storage and the impact in speed and cost that vectorization can have on modern CPUs. Vectorization of the relational database systems which have been adopting an iterator style can be very beneficial as this will increase the speed of execution with the reduction of cost.

References:

[1] Boncz, P. A., Zukowski, M., & Nes, N. (2005, January). MonetDB/X100: Hyper-Pipelining Query Execution. In *Cidr* (Vol. 5, pp. 225-237).

[2] Lang, H., Mühlbauer, T., Funke, F., Boncz, P. A., Neumann, T., & Kemper, A. (2016, June). Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data* (pp. 311-326).

[3] Sompolski, J., Zukowski, M., & Boncz, P. (2011, June). Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware* (pp. 33-40).

[4] Zukowski, M., & Boncz, P. A. (2012). Vectorwise: Beyond column stores. *IEEE Data Engineering Bulletin*, *35*(1), 21-27.

[5] Zukowski, M., Van de Wiel, M., & Boncz, P. (2012, April). Vectorwise: A vectorized analytical DBMS. In *2012 IEEE 28th International Conference on Data Engineering* (pp. 1349-1350). IEEE.

[6] Kemper, A., & Neumann, T. (2011, April). HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering* (pp. 195-206). IEEE.

[7] Karde, P., & Thakare, V. (2010). Selection of materialized views using query optimization in database management: An efficient methodology. *International Journal of Management Systems*, *2*(4), 116-130.

[8] Bello, R. G., Dias, K., Downing, A., Feenan, J., Finnerty, J., Norcott, W. D., ... & Ziauddin, M. (1998, August). Materialized views in Oracle. In *VLDB* (Vol. 98, pp. 24-27).

[9] Gosain, A., & Sachdeva, K. (2017). A systematic review on materialized view selection. In *Proceedings of the 5th International Conference on Frontiers in Intelligent Computing: Theory and Applications* (pp. 663-671). Springer, Singapore.

[10] Chirkova, R., & Yang, J. (2011). Materialized views. *Foundations and Trends in Databases*, *4*(4), 295-405.

[11] Pavlo A. (2020), Advanced Database Systems: Query Execution and Processing, 15-721.

[12] Elmasri, R., Navathe, S. B., Elmasri, R., & Navathe, S. B. (2000). *Fundamentals of Database Systems</Title*. Addison-Wesley</publisher.

[13] Polychroniou, O., Raghavan, A., & Ross, K. A. (2015, May). Rethinking SIMD vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (pp. 1493-1508).

[14] Monetdb. (n.d.). MonetDB Documentation. https://www.monetdb.org/Documentation

[15] Citus Data. (2020). Cstore_fdw. https://github.com/citusdata/cstore_fdw

[16] Citus Data. (2021). Citus. https://github.com/citusdata/citus

[17] Citus Data. (2015). Postgres vectorization test. https://github.com/citusdata/postgres_vectorization_test