



Survey of Vectorized Database Techniques

CS 541

Group: VectDB2

Shivam Bhat (PUID : 33760929)

Venkat Sai (PUID : 34779773)

Akshay Ramakrishnan (PUID : 33660675)



Content

1. What and Why - Vectorization?
2. SIMD/SISD
3. Literature Review/Existing Work
4. Challenges
5. Future Work



What is Vectorization?

The process of converting an algorithm's scalar implementation processing a single pair of operands at a time, to one with a vector implementation processing one operation on multiple pairs of operands at once.

This can significantly enhance the performance of certain database operations, such as querying and filtering large datasets.



Vectorization in Databases

Vectorization can be used to speed up a variety of operations during query processing in database systems, including filtering, sorting, aggregating, and merging huge datasets.

By breaking down the data into vectors and processing them simultaneously, vectorized operations can be orders of magnitude faster than their scalar counterparts.



Vectorization in Databases

In a **traditional scalar-based query processing system**, filtering large amounts of data would **require iterating over each row of the dataset** and applying the filter condition one row at a time.

However, **in a vectorized system**, the **filter** condition can be **applied to entire vectors of data at once**, resulting in much faster processing.

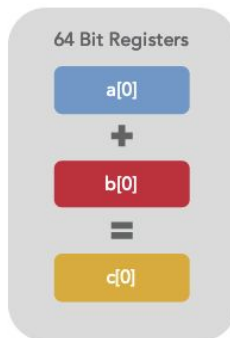
Vectorization can also **improve memory usage and cache efficiency**, as **vectorized operations typically require fewer memory accesses** and can be more easily optimized for cache locality.

SIMD - Single Instruction, Multiple Data

A set of CPU instructions that allow the processor to **operate simultaneously on many data points**

```
for (i = 0; i < count; i++)  
    c[i] = a[i] + b[i];
```

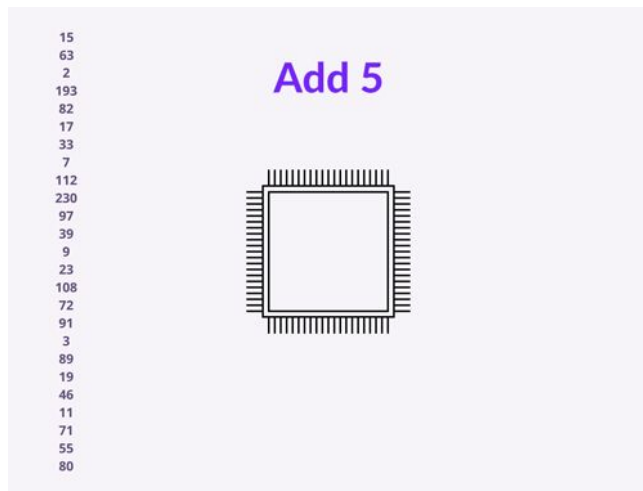
Scalar Mode



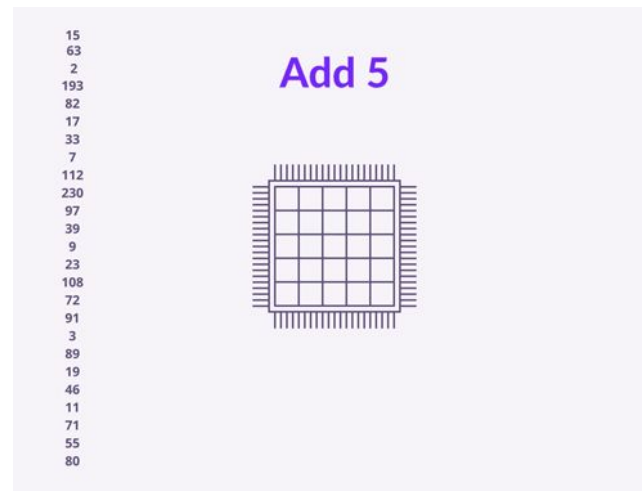
SIMD Vector Mode



SIMD in action



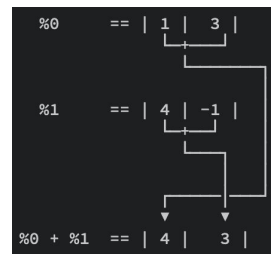
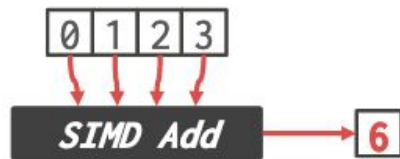
SISD



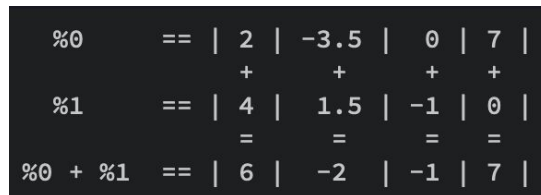
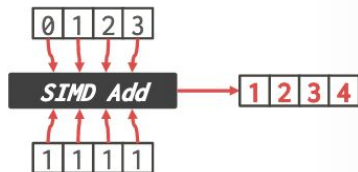
SIMD

Types of Vectorization-Directions

Horizontal : Operates on all elements together within a single vector.



Vertical : Operates in an element-wise manner on elements of each vector





Parallelism

Hardware provides performance through three sources of parallelism:

1. Thread parallelism
2. Instruction level parallelism
3. Data parallelism

Databases have evolved to take advantage of all sources of parallelism.



Thread parallelism

Thread parallelism is achieved, for individual operators, by splitting the input equally among threads.

In the case of queries that combine multiple operators, by using the pipeline breaking points of the query plan to split the materialized data in chunks that are distributed to threads dynamically.



Instruction level parallelism

Instruction level parallelism is achieved by applying the same operation to a block of tuples and by compiling into tight machine code

```
int a=0;
```

```
int b=1;
```

Data parallelism

Data parallelism is achieved by implementing each operator to use SIMD instructions effectively



Parallelism

Apart from **multi core chip design enabled parallelism**, chipmakers have also been increasing the power of a **second type of parallelism, instruction level parallelism**.

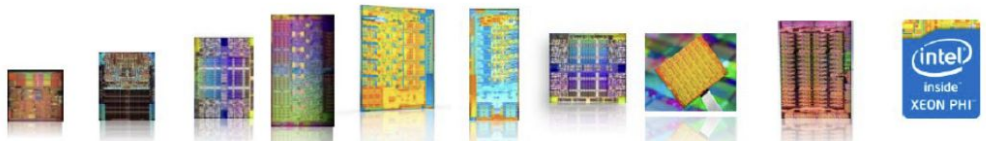
If we **parallelize our algorithm over 8 cores in a 4-wide SIMD** register architecture.

Speedup = **Cores** (Threads) + **Vectorization**.

$$\text{Speedup} = 8x * 4x = \mathbf{32x}$$

Adoption in Industry

Alongside the trend to increase core count, the width of SIMD registers has been steadily increasing.



	Intel® Xeon® processor 64-bit	Intel® Xeon® processor 5100 series	Intel® Xeon® processor 5500 series	Intel® Xeon® processor 5600 series	Intel® Xeon® processor code-named Sandy Bridge EP	Intel® Xeon® processor code-named Ivy Bridge EP	Intel® Xeon® processor code-named Haswell EP	Future Xeon	Intel® Xeon Phi™ coprocessor Knights Corner	Intel® Xeon Phi™ processor & coprocessor Knights Landing ¹
Core(s)	1	2	4	6	8	12	18	>18	61	70+
Threads	2	2	8	12	16	24	36	>36	244	280+
SIMD Width	128	128	128	128	256	256	256	512	512	512

Source: Intel



Literature Review:

- Research Papers
- Research articles
- Case study

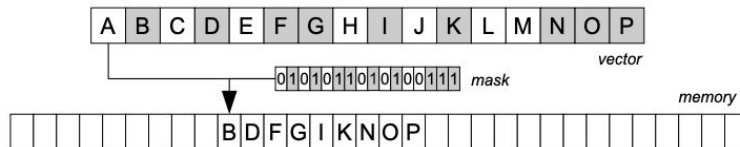


Rethinking SIMD Vectorization for In-Memory Databases

1. Paper **introduces design principles** for efficient vectorization of in-memory database operators and **defines fundamental vector operations** .
2. Design and implements vectorized selection scans, hash tables, and partitioning, that are combined to design and build sorting and multiple join variants.
3. Compares their implementations against state-of-the- art scalar and vectorized techniques. Authors achieved up to an order of magnitude speedups by evaluating on Xeon Phi as well as on the latest mainstream CPUs.

Fundamental Operations

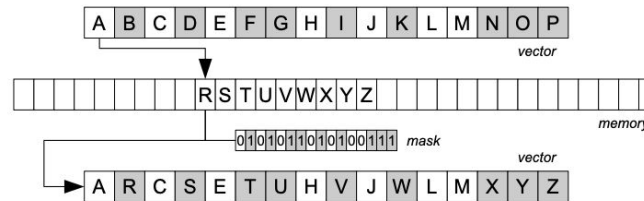
Selective load and Selective store - Spatially contiguous memory accesses that load or store values using a subset of vector lanes.



1. **Selective stores** write a specific subset of the vector lanes to a memory location contiguously. The subset of vector lanes to be written is decided using a vector or scalar register as the mask.

Fundamental Operations

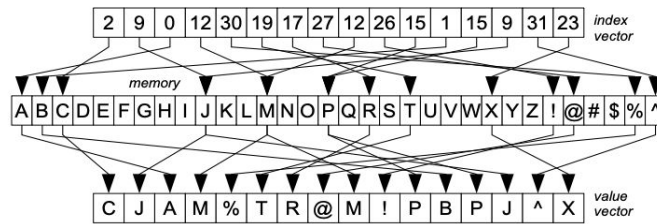
2. Selective loads are the symmetric operation that involves loading from a memory location contiguously to a subset of vector lanes based on a mask. The lanes that are inactive in the mask retain their previous values in the vector.



Fundamental Operations

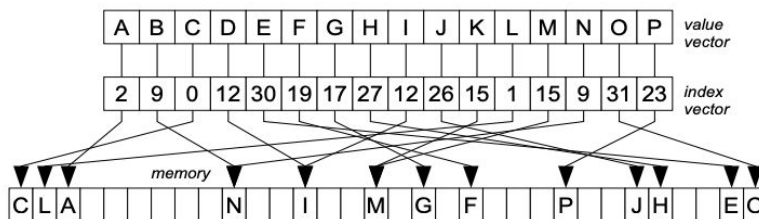
3. Gather: loads values from non-contiguous locations. The inputs are a vector of indexes and an array pointer. The output is a vector with the values of the respective array cells.

By adding a mask as an input operand, we define the selective gather that operates on a subset of lanes. The inactive mask lanes retain their previous contents.



Fundamental Operations

4. **Scatter** operations execute stores to multiple locations. The input is a vector of indexes, an array pointer, and a vector of values. If multiple vector lanes point to the same location, we assume that the rightmost value will be written.





Vectorized operators

Authors further Implement vectorized operators in the context of main-memory databases:

1. Selection scans
2. Hash tables
3. Partitioning

which are combined to build more advanced operators: sorting and joins.

- Joins
- Sorting
- Bloom Filters

Selection Scan

SELECT * FROM table
WHERE key \geq \$low AND key \leq \$high

Scalar (Branching)

```
i = 0
for t in table:
    key = t.key
    if (key  $\geq$  low) && (key  $\leq$  high):
        copy(t, output[i])
        i = i + 1
```

Scalar (Branchless)

```
i = 0
for t in table:
    copy(t, output[i])
    key = t.key
    m = (key  $\geq$  low ? 1 : 0) &&
        (key  $\leq$  high ? 1 : 0)
    i = i + m
```

Source: Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). Association for Computing Machinery, New York, NY, USA, 1493–1508.

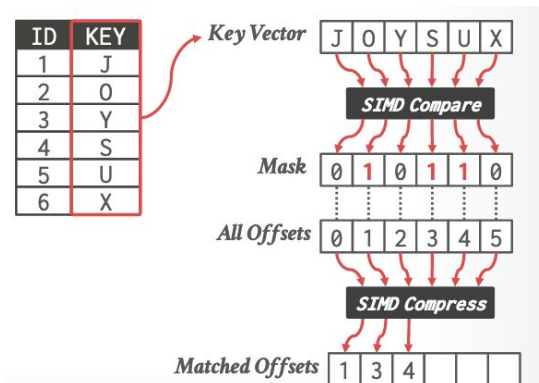
<https://doi.org/10.1145/2723372.2747645>

Andy P, CMU 15-721 Course

Selection Scan

Vectorized

```
i = 0
for vt in table:
    simdLoad(vt.key, vk)
    vm = (vk ≥ low ? 1 : 0) &
        (vk ≤ high ? 1 : 0)
    simdStore(vt, vm, output[i])
    i = i + |vm ≠ false|
```



Source: Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). Association for Computing Machinery, New York, NY, USA, 1493–1508.

<https://doi.org/10.1145/2723372.2747645>

Andy P, CMU 15-721 Course



Hash Table

Hash tables are used in database systems to execute joins and aggregations since they allow constant time key lookup

Using SIMD instructions in hash tables has been proposed as a way to build bucketized hash tables. Rather than comparing against a single key, we place multiple keys per bucket and compare them to the probing key using SIMD vector comparisons.

Source: Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). Association for Computing Machinery, New York, NY, USA, 1493–1508.

<https://doi.org/10.1145/2723372.2747645>

Andy P, CMU 15-721 Course



Hash Table

In this paper, the authors propose a generic form of hash table vectorization termed vertical vectorization that can be applied to any hash table variant without altering the hash table layout. The fundamental principle is to process a different input key per vector lane.

Source: Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). Association for Computing Machinery, New York, NY, USA, 1493–1508.

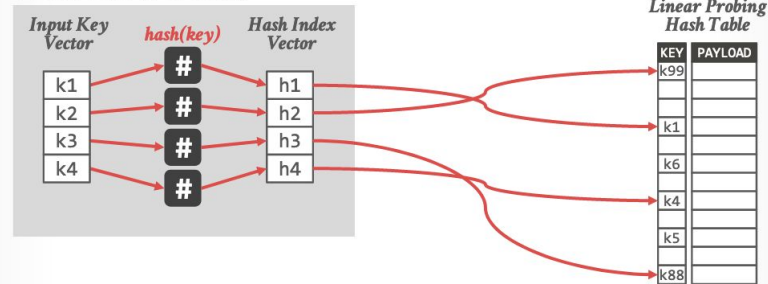
<https://doi.org/10.1145/2723372.2747645>

Andy P, CMU 15-721 Course

Hash Table

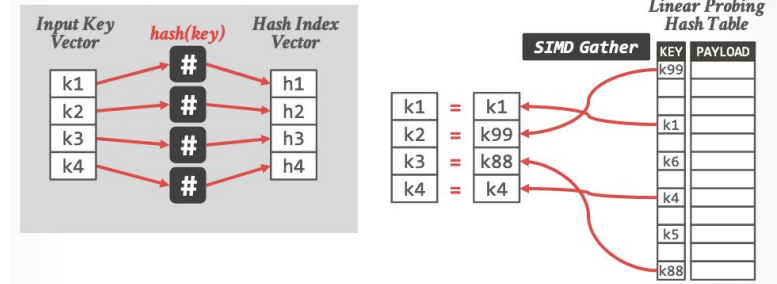
HASH TABLES - PROBING

Vectorized (Vertical)



HASH TABLES - PROBING

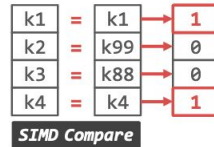
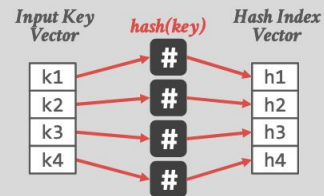
Vectorized (Vertical)



Hash Table

HASH TABLES – PROBING

Vectorized (Vertical)

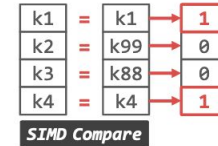
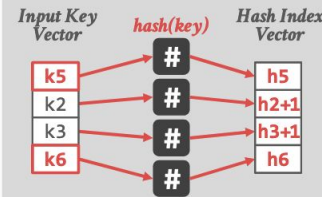


Linear Probing Hash Table

KEY	PAYLOAD
k99	
k1	
k6	
k4	
k5	
k88	

HASH TABLES – PROBING

Vectorized (Vertical)



Linear Probing Hash Table

KEY	PAYLOAD
k99	
k1	
k6	
k4	
k5	
k88	

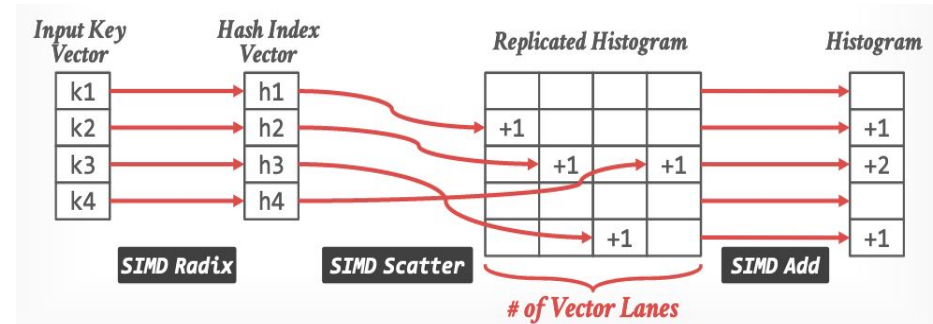
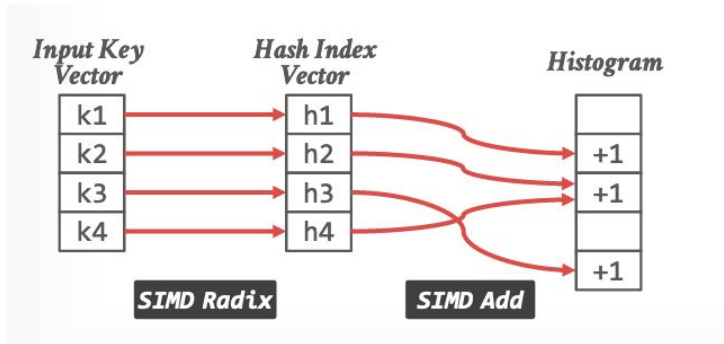


Partitioning

Partitioning is a ubiquitous operation for modern hardware query execution as a way to split large inputs into cache-conscious non-overlapping sub-problems.

Though this very large tables are divided into multiple smaller parts. By splitting a large table into smaller, individual tables, queries that access only a fraction of the data can run faster because there is less data to scan.

Partitioning



Source: Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). Association for Computing Machinery, New York, NY, USA, 1493–1508.

<https://doi.org/10.1145/2723372.2747645>

Andy P, CMU 15-721 Course



Vectorwise: A Vectorized Column-Store DBMS

1. The author presents Vectorwise, a relational DBMS that uses vectorized processing techniques to achieve high performance on a wide range of query types.
2. The authors describe how Vectorwise operates on arrays of data, using SIMD instructions to perform operations on multiple data items in a single instruction.
3. This allows Vectorwise to process data much more efficiently than traditional row-oriented DBMSs, which process data one row at a time.



Vectorwise: Working

Let's say Purdue maintains the following database to record tuition fees it collects for all students (fee depends in streams, no. of courses, level of education etc.),

Fee date	Student Name	Total Fee Paid
2022-01-01	Alice	15500
2022-01-02	John	6598
2022-01-03	Mary	20220
2022-01-04	Olivia	2800
2022-01-05	John	350



Vectorwise: Working

Purdue now wants to calculate the total fee paid by each student. When executed on a traditional database, this query would process one row at a time, calculating the sum of the fees for each name.



Vectorwise: Working

In VectorWise, the same query would be executed differently. Here's are the following steps:

1. **Parsing and Optimization:** The query is parsed and optimized to generate an execution plan. The optimizer identifies that the query involves a grouping operation, and generates a plan that uses a hash-based aggregation algorithm.
2. **Vectorization:** The execution plan is then transformed to take advantage of vectorization. In this case, the aggregation operation can be performed using SIMD instructions, which enables multiple values to be processed in parallel.
3. **Memory Layout:** The fees data is stored in a columnar format, with each column of the table stored separately in memory. This allows VectorWise to operate on entire columns at once, which is more efficient than processing individual rows.



Vectorwise: Working

1. **Pipelining:** The execution plan is pipelined to overlap the different stages of query execution. This allows VectorWise to operate on data while it is being processed, reducing overall latency.
2. **Cache-awareness:** VectorWise is designed to be cache-aware, which means that it takes advantage of the different levels of cache available on modern processors. The data that is frequently accessed is kept in the cache to reduce memory access latency.
3. **Execution:** Once the query has been optimized and transformed, it is executed using the VectorWise execution engine. The engine operates on multiple columns at once, using SIMD instructions to calculate the sum of fees for each student. The results are then pipelined to the next stage of the query, until the final results are produced.



Quickstep: A Data Platform Based on the Scaling-In Approach

1. This paper focuses on improving the query processing performance to exploit the hardware compute parallelism available for use in modern systems.
2. The crux of the design is to allow high intra-operator parallelism. (Dividing a single task into a set of sub-tasks that can run in parallel without the need for data exchange).
3. By using a block-based storage design, Quickstep generates independent tasks/jobs at the block level which in turn utilizes the parallelism provided by the hardware environment.



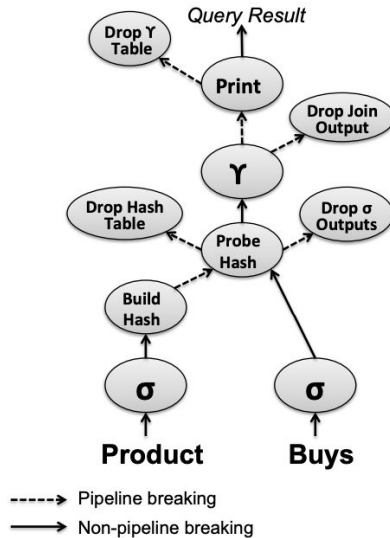
Quickstep: A Data Platform Based on the Scaling-In Approach

1. These jobs are scheduled by a highly parallelizable query execution paradigm.
2. Another focus of Quick step is to reduce the number of semi-joins by using cache-efficient filter data structures and by “pushing down” complex disjunctive predicates.
3. This helps in reducing the number of redundant rows to be processed later on by evaluating these predicates early on in the query execution process.



Caveats: Using a Hashtable

- Expenses due to Materialization are minimized by using hash tables.
- Quickstep has two hashing phases: Building and Probing
- In the building phases, Quickstep stores relations in the hash tables with the join predicates as the key.
- The probe hash table operator reads blocks of the probe relation, probes the hash table, and materializes joined tuples into in-memory blocks.
- Quickstep differs in the context in which the implementation of the hash table algorithm makes use of the block level parallelism effected by the storage manager as different blocks can utilize the hash table concurrently and “latch-free”.



DAG execution pipeline of a sample Query.
Image sourced from paper.

- The picture of the represents the process of how Quickstep processes a given query.
- Each node in the graph can issue independent work orders.
- We can see that the edge from Probe Hash to Aggregation allow for pipelining.
- By generating a work order for each streamed input block received from the probe operator, the aggregation operator is able to achieve pipelining.



Case Study: CockroachDB

1. CockroachDB is an OLTP database which aims to execute queries rapidly with high throughput.
2. The data in CockroachDB is stored in as contiguous rows in disk which led to declining speed in complex analytical queries like aggregations.
3. Upon realizing that the row-oriented execution engine comes up higher cost, they adapted a columnar storage and execution engine.
4. Since, CockroachDB follows a key-value architecture, they converted rows into batches of columnar data after reading them from the disk.
5. It's interesting to think about the challenges they must have faced during this conversion.



Challenges of Database Vectorization

1. Storage of disk data in columnar format: The storage engine needs to be modified to adapt to the new columnar data format.
2. Changes in downstream process: Multiple changes are necessary in order to account for this new storage architecture. This would result in extensive code-changes to support vectorized operations, functions, etc.
3. Need for new data structures: This change might necessitate development of new data structures to support vectorizations.
4. Memory Management: The memory management scheme needs to be rethought in order to fully leverage and reap the benefits of vectorized execution (through SIMD CPUs).



Future Papers - Brief Introduction

1. [The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases](#) is a paper that proposes a new data structure called the Adaptive Radix Tree (ART) for indexing main-memory databases. The paper argues that traditional indexing structures such as B-trees and hash tables are not well-suited for modern main-memory databases, as they suffer from cache inefficiency, memory fragmentation, and poor scalability.
2. [Efficiently Compiling Efficient Query Plans for Modern Hardware](#) is a paper that addresses the challenge of efficiently compiling query plans for modern hardware architectures. Traditional query optimization techniques often fail to fully leverage the performance potential of modern hardware, such as multicore CPUs and advanced instruction sets.



Future Papers - contd.

3. **Data Blocks:** Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation proposes an architecture, called Data Blocks, that uses a hybrid approach that combines in-memory columnar storage, vectorized query execution, and code generation. Data Blocks stores data in compressed blocks that are optimized for both OLTP and OLAP workloads. The architecture also incorporates a compilation process that generates efficient code for executing queries on the compressed data.
4. **Quickstep :** A Data Platform Based on the Scaling-In Approach - So far we have seen the entire process overview and how the hash table operations are performed to reduce intermediate outputs overhead. In the next milestone, we plan to dig deeper into how Quickstep pushes down partial predicates and how its evaluated against other vectorized database system implementations.



Future Work

1. Analysis of various vectorized operators.
2. Compiler Hints and Vectorization
3. Overcoming challenges for vectorization in databases
4. Vectorization Implementation in modern databases
5. Deep diving into surveyed papers



References

1. Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15). Association for Computing Machinery, New York, NY, USA, 1493–1508. <https://doi.org/10.1145/2723372.2747645>
2. "Zukowski, M., van de Wiel, M., & Boncz, P. (2012). Vectorwise: A Vectorized analytical DBMS. *2012 IEEE 28th International Conference on Data Engineering*. <https://doi.org/10.1109/icde.2012.148> - <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6228203>
3. Leis, V., Kemper, A., & Neumann, T. (2013). The adaptive radix tree: Artful indexing for main-memory databases. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. <https://doi.org/10.1109/icde.2013.6544812> - <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6544812>
4. Neumann, T. (2011). Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9), 539–550. <https://doi.org/10.14778/2002938.2002940> - <https://dl.acm.org/doi/abs/10.14778/2002938.2002940>
5. Lang, H., Mühlbauer, T., Funke, F., Boncz, P. A., Neumann, T., & Kemper, A. (2016). Data Blocks. *Proceedings of the 2016 International Conference on Management of Data*. <https://doi.org/10.1145/2882903.2882925> - <https://dl.acm.org/doi/abs/10.1145/2882903.2882925>
6. Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: a data platform based on the scaling-up approach. *Proc. VLDB Endow.* 11, 6 (February 2018), 663–676. <https://doi.org/10.14778/3184470.3184471>
7. Yoon-Min Nam Nam, Donghyoung Han Han, and Min-Soo Kim Kim. 2020. SPRINTER: A Fast n-ary Join Query Processing Method for Complex OLAP Queries. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20). Association for Computing Machinery, New York, NY, USA, 2055–2070. <https://doi.org/10.1145/3318464.3380565>
8. Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2019. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.* 11, 13 (September 2018), 2209–2222. <https://doi.org/10.14778/3275366.3284966>



References

9. Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. Proc. VLDB Endow. 11, 13 (September 2018), 2209–2222.:<https://doi.org/10.14778/3275366.3284966>
10. W. Zhang and K. A. Ross, "Exploiting Data Skew for Improved Query Performance," in *IEEE Transactions on Knowledge and Data Engineering*:
<https://ieeexplore.ieee.org/document/9130938>
11. Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2019. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask:<https://doi.org/10.14778/3275366.3284966>