Hasso Plattner Institute
Chair for Data Engineering Systems

Proposal Master Thesis

Using Column Stores for Stream Processing

Björn Daase

Time frame: August 2022 - February 2023

Supervisor
Prof. Dr. Tilmann Rabl

Advisor
Lawrence Benson, M. Sc.

1 Motivation

Current Stream Processing Engines (SPEs) can process terabytes of incoming data per second [4]. Because widely used SPEs such as Apache Flink [11] and Spark Streaming [37] do not fully utilize the underlying hardware and are resource inefficient [38, 39], the implementation of recent SPEs shifted towards system languages such as C and C++. This increased their throughput by up to two orders of magnitude compared to state-of-the-art SPEs [38].

However, along with these performance gains, the bottlenecks of SPEs have also shifted. In particular, it has become apparent that memory and not CPU performance is the new bottleneck for many queries [7].

The shift to system languages also brings SPEs much closer to classical database systems, which are usually also implemented in system languages. One trend observed in classical database systems in recent years is the introduction of more and more column-oriented database systems, like MonetDB [8, 10, 19], C-Store [31] or SAP HANA [29]. They work around the same memory bottleneck we now observe for SPEs [9] by accessing memory more efficiently [1], which makes them ultimately faster than row-based DBMS.

At first sight, this column-based storage of tuples seems inherently incompatible with true streaming, which assumes processing one tuple at a time. However, this understanding does not reflect the behavior of modern SPEs anymore. Taking the Apache Kafka [22] to Apache Flink interaction as an example, data is always transferred in so-called nano-batches between systems, i.e. tuples already arrive in Apache Flink in larger sets. Upon arrival, Apache Flink then processes those nano-batches in a batch processing manner accordingly. The exact size of the nano-batches depends on the actual application, i.e. its performance and configuration. In addition, MonetDB/Datacell [25] and Trill [13], two SPEs that use columnar storage, have already been developed in recent years.

Column-based systems store data as a Struct of Arrays (SoA) and thus the same attributes of different tuples next to each other in memory [1, 2]. If an attribute is iterated over in a hot loop, this has the advantage that when the attribute of one tuple is accessed, the corresponding attributes of the other tuples are already present in the same cache line. Thus, when they are requested in subsequent loop iterations, they do not have to be explicitly read from main memory again, but can be served from the cache and are available almost immediately.

This is not the case with row-based systems where the data is stored as an Array of Structs (AoS). There, the attributes of a single tuple are contiguous in memory. Now, when the attribute of a tuple is accessed, the cache line is polluted with surrounding irrelevant attributes [1, 2] instead of getting the attributes of the next few tuples already for free. Therefore, subsequent loop iterations must perform additional main memory accesses instead of serving the data from the cache.

Column stores however also come with a flip side. Since streaming is inherently tuple-based, data is exchanged between systems almost exclusively in a row-based manner (one of the few exceptions here is *Parquet*). Thus, column-based processing of tuples incurs additional costs for transformation at data ingress and retransformation at data egress.

Considering the combined costs, i.e. the gains in the actual processing step as well as the losses due to the conversion to a columnar format, it is interesting to investigate how a column-oriented SPE performs against a row-based one. Depending on the results, this also raises the question of whether hybrid forms of data storage are worthwhile. If, for example, some fields are not used during the actual processing (i.e. no filters, joins or aggregations take place on them), but they have to be finally issued with the tuples as payload, this cold data could be kept in a row-store while the hot data is processed in a column-store. Of particular interest is how the characteristics of the queries, schemas, and nano-batches influence the performance.

2 Goal of Thesis

In order to understand the impact of a column-based compared to a row-based layout for an SPE, we analyze their raw query performance along several microbenchmarking dimensions. Therefore, we select a set of Nexmark queries [35] to be able to pinpoint the influence on query performance depending on the relational algebra operators. We will further investigate a few side branches to also understand the influence of preceding input parsing, explicit SIMD operations, the underlying memory architecture and vectorizing extension, different window types, and multithreading. Independently of the benefits, we will look at the transformation of incoming tuples into a columnar-based store as well as their materialization back into a row-based format at the end of the query so that we can estimate the costs. As a result, this allows us to select a suitable storage format before query execution.

After analyzing the design space in several microbenchmarks, we test the suitability of non-row store storage in the Darwin stream processing engine [6]. At this point, we get even more concrete application scenarios and can further tune our hybrid approach. Given that, we plan the following contributions:

1. We understand the impact of different data layouts (row-based, column-based, or hybrid) on individual relational algebra operators.

2. We relate the benefits and costs of column-based processing to each other, allowing the selection of a suitable format at query compilation time.

3. We validate our approach in the Darwin stream processing engine [6].

3 Approach

We begin by analyzing the advantages of a column store over a row store in an exhaustive set of microbenchmarks based primarily on Nexmark queries [35]. The queries are chosen to represent as exclusively as possible the relational algebra operation under consideration. We implement the processing of each operator once with data input, processed, and output as row storage, and once with data input, processed, and output as column storage. The queries are hard-coded in C++, with the data stored as AoSs if row memory is simulated, and as SoAs in the case of column memory. In a separate step for each operation, we will then streamline the selected query to contain only the operation and thus not be affected by external factors. Finally, we will also explore a few side strands for a few selected interesting and representative queries, which might further influence the row and column store performance.

After discussing the benefits, we will continue with the cost of converting to column format before the actual query, as well as converting back to a row format at the end of it. As a result, we can hopefully predict at query compilation time whether an SPE will process data more efficiently row-based, column-based, or in a hybrid matter. Finally, we test the suitability of non-row-store streaming in the Darwin stream processing engine [6].

3.1 Microbenchmark Analysis - Benefits

Relational algebra boils down to the following four query primitives: *Projections*, *Selections*, *Aggregations*, and *Joins*. We will approach them one by one, in the aforementioned order. Across the operator experiments, we assume tumbling windows. Orthogonally, we will consider for each primitive at least the influence of the data schema structure and the size of the input nano-batches. Additionally, we will analyze operator-specific characteristics.

3.1.1 Projection

A projection π is the simplest operation from the primitives of relational algebra, which is why we will start with it. It eliminates all attributes of the input relation but those mentioned in the projection list.

We select query 0 from the Apache Beam Nexmark benchmark suite extension [15]. It looks as follows

```sql
SELECT auction, bidder, price, dateTime, extra FROM Bid;
```

with the underlying schema

```cpp
Bid(size_t auction, size_t bidder, size_t price, std::string channel,
    std::string url, time_t dateTime, std::string extra);
```

Already on this projection, we will analyze the influence of the nano-batch size, i.e. the tuples arriving at once. Here we already hope for a better vectorization due to a larger number of available tuples.

In the next step, we will slightly modify the query so that all of the attributes of the input schema *Bid* also occur in the output, instead of additionally filtering attributes. As a result, the query represents a clean passthrough that is not influenced by any other operation and thus allows us to examine how unused attributes influence the query performance. Working only on a few columns allows dropping entire arrays from a SoA, which should heavily favor the column store.

Following, we modify the query to evaluate the influence of the schema on the projection. In order to do so, we vary the number of fields of the input and output relations, as well as the size of the individual relations themselves. The latter is controlled by modifying the length of the *std::string* fields.

In addition, projections also cover calculations on some attributes. This time we use query 1, which looks as follows

```sql
SELECT auction, bidder, 0.908 * price, dateTime, extra FROM Bid;
```

To fully understand the influence of the actual computation, we also vary the query to remove all attributes from input and output that do not have to do with the actual calculation. We further adapt this query during evaluation by replacing the floating point multiplication on the price with an integer multiplication, thus not getting a floating number as a result. To further analyze the data types, we will replace the operation with an operation on an *std::string* attribute. We also vary the complexity of the operation to understand its impact on performance.

3.1.2 Selections

A selection σ is used for selecting a subset of the tuples according to a given selection condition. We will use query 2 from the Apache Beam Nexmark benchmark suite extension to understand its impact on column and row stores. It represents a slightly modified query 2 from the original Nexmark queries since the original query will only yield a few hundred results over event streams of arbitrary size. To make it more interesting, they instead choose a modulo operation for filtering. It looks as follows

```sql
SELECT auction, price FROM Bid WHERE MOD(auction, 123) = 0;
```

Again, we are looking at the size of the nano-batches, i.e. tuples that can be filtered at once. Here we expect that an increased number of tuples leads to a much better vectorization on the hot filter attribute *auction* and thus significantly improves the query performance. Unique to the selection investigation, however, is the filter condition, which we will look at in detail. Specifically, we investigate **(a)**, the selectivity of the filter, **(b)** the data type of the column being filtered on, and **(c)** the effects of filters that span multiple columns.

3.1.3 Aggregations

Aggregations are an umbrella term for the following five aggregate functions: *Sum*, *Count*, *Average*, *Maximum*, and *Minimum*. They differ in the concrete calculation of the aggregates, but they have in common that first a grouping according to a certain attribute has to be done.

Unfortunately, Nexmark does not provide a query that is exclusively an aggregation. Therefore, we use the subquery of query 5 to represent an aggregation. It consists only of a *Count*, and thus the most CPU resource-friendly, aggregate function. It looks as follows

```
SELECT auction, count(*) AS num FROM Bid GROUP BY auction;
```

To fully isolate the effect of the aggregation, we will again slim down this query to just the actual aggregation column. As with the previous operators, we will also examine the impact of the scheme on query performance. We however note at this point, that the aggregation payload might significantly influence the performance.

In addition to again varying the size of the nano-batches, the window size is also of particular interest for an aggregation. While the nano-batch size represents the input tuple size, the window size defines how many tuples can be processed and output at once. We will further investigate the influence of the actual aggregation by adapting the aforementioned query for all four other aggregation functions accordingly.

3.1.4 Join

A join θ is an operation that combines two relations with respect to a condition. Thus, it is the only one of the operations considered that necessarily requires more than one column, which is why we consider it last.

This time we use Nexmark query 3, which looks as follows

```
SELECT
    P.name, P.city, P.state, A.id
FROM
    auction AS A INNER JOIN person AS P on A.seller = P.id
WHERE
    A.category = 10 and (P.state = 'OR' OR P.state = 'ID' OR P.state =
    'CA');
```

with

```
Auction(size_t id, std::string itemName, std::string description,
    size_t initialId, size_t reserve, time_t dateTime, time_t expires,
    size_t seller, size_t category, std::string extra);
```

and

```
Person(size_t id, std::string name, std::string emailAddress, std::
    string creditCard, std::string city, std::string state, time_t
    dateTime, std::string extra);
```

As a next step, we again modify the query so it only contains the columns relevant for the join. For the payload, we will then proceed exactly as we did for the aggregation.

Besides again investigating the influence of the nano-batch size as well as the schema structure, a unique parameter to joins is the likelihood of finding a join partner. Since the Nexmark queries do not come up with ranges for the attributes in the schema, we have to predict values and distributions.

3.2 Microbenchmark Analysis - Costs

Using a column store does not come without a cost. The tuples must first be converted from a row-based format to a column-based format upon arrival in the system, whereas with row-based processing a bare copy of the corresponding memory areas is sufficient. As a final step after the query finished processing a tuple, it has to be egested as a full tuple again, i.e. transformed back into a row representation. Again, a bare copy of the corresponding memory areas is sufficient when doing row-based processing.

3.2.1 Input Transformation

The additional work of data transformation from a row-based to a column-based format also affects regular DBMS during data ingestion. However, in a DBMS the data is read again multiple times after ingestion since multiple queries are expected to be executed on the same data. This is usually not the case with SPEs, where data is written once and only read once again.

At this point, we compare the transformation of all Nexmark schemas (*Auction*, *Bid*, and *Person*) from a row-based to a column-based format. The baseline against which we compare performance is a transformation of the schemas from a row-based to a row-based format, which can be represented by a *memcpy*.

At this point, we expect again particular influences from the scheme as well as the size of the nano-batches arriving in the system. While larger nano-batches should allow transforming the data from a row store into a column store more efficiently, many attributes could hinder the performance since during the split-up of the input tuple multiple output arrays have to be accessed in that case.

3.2.2 Output Transformation

The same transformation that happens at the beginning of each query has to happen vice-versa at the end of it. Since at this point we have looked at Nexmark queries 0, 1, 2, 3, and 5, we will also look at conversion from their result schemas. Those are

```
Query0(size_t auction, size_t bidder, size_t price, time_t dateTime,
    time_t expires, std::string extra);
```

```
Query1(size_t auction, size_t bidder, double price, time_t dateTime,
    time_t expires, std::string extra);
```

```
Query2(size_t auction, size_t price);
```

```
Query3(std::string name, std::string city, std::string state, size_t id
    );
```

```
Query5(size_t auction, size_t num);
```

The baseline will again be transforming data from a row-based to a row-based format which is most efficiently done by a *memcpy*.

3.3 Additional Influences

While the aforementioned operators will be the focus of our analysis, we presume that there are additional factors that influence the performance of a row store as well as a column store for SPEs. Those are the influence of additional input parsing and combined transformation into the correct data layout, the explicit vectorization of code, examining different memory architectures, the influence of window types, and, if time permits, how multithreading affects the comparison.

3.3.1 Explicit Input Parsing

While our previous additional cost analysis expected tuples to be arriving in C structs, the exchange format between stream processing systems is usually somewhat different. JSON [23, 24, 28] and CSV [16, 27, 30] files, in particular, are taken for exchange, which then have to be parsed when the tuples arrive. Thus, we want to examine to what extent the advantage of the row-based processing when using advanced exchange formats compared to a simple *memcpy* is forfeited. We expect, however, that the tuple exchange format should still favor the row-based store when ingesting data.

3.3.2 Explicit SIMD

Previous research has shown that inserting explicit SIMD operations compared to relying on compiler auto-vectorization can further improve query performance significantly [20]. At this point, we want to investigate to what extent this effect also applies to data that is already stored in a SoA, and thus should already be easy for compilers to auto-vectorize.

3.3.3 Memory Architectures

As shown by Kersten et al. [20], vectorizing plays a central role in efficient query processing. However, with different vectorizing extensions on different platforms as well as different kinds of memory on different machines [7], especially High Bandwitch Memory (HBM), the performance of row-store-based and column-store-based streaming could be influenced significantly. For reference, the HBM ARM systems can reach up to 3x sequential memory bandwidth compared to state-of-the-art Intel, AMD, and Power systems which peak around 200 GB/s [7]. Therefore, we want to understand how these different kinds of memory influence row-store and column-store streaming.

3.4 Windowing Types

While we only work with tumbling windows up to this point, we will extend the space by examining different windowing types. Here's what we expect: While with tumbling windows each tuple is written once and read exactly once again, the situation is different for sliding windows. The slices created by stream slicing are now read again for each window the slices belong to. This results in a write-once-read-many scenario much closer to conventional DBMS which should favor the column store.

3.4.1 Multithreading

While multithreading, in general, is orthogonal to our design space in theory, designing multithreading-aware memory accesses often has tripping hazards and is therefore often non-trivial. Especially cache line reuse and invalidation tend to be tricky in a multithreading scenario. It is important to note, that, at this point, we do not plan to design a fully parallel column-based SPE. The primary focus is to verify the design and ensure that the column-based SPE is also multithreading capable.

3.5 Combined Costs

Finally, after all the individual analyses of the row-store and column-store-based design, we will evaluate the combined end-to-end of costs of a few selected queries.

3.6 Darwin Integration

As a very last step, we test the suitability of non-row-store streaming in the Darwin stream processing engine [6]. A complete integration, which then allows both row-based and column-based processing of the data, appears to be very maintenance-intensive. Accordingly, if column-based processing only brings small advantages, we will only test its use in a prototype. If it turns out that column-based processing is indeed decisively superior in SPEs, the entire Darwin memory model should be adapted accordingly. Provided that it turns out that there are queries where one has advantages with column-based processing as well as others with row-based processing, we will have to implement both storage models in Darwin.

4 Related Work

Both SPEs, as well as column stores, are well-researched topics. However, their combination has so far been limited to their integration by MonetDB/Datacell [25] and Trill [13] without a detailed analysis of the advantages and disadvantages of such an integration.

4.1 Stream Processing Egines

A repeatedly mentioned difference between modern SPEs is the underlying processing model [38], which can be either micro-batching or pipeline-tuple-at-a-time processing [3, 5, 12, 14, 36]. Micro-batching SPEs divide data streams into batches, i.e. finite sets of tuples, and process one batch at a time. Prominent examples of that execution model are Spark [37] and SABER [21]. In contrast, tuple-at-a-time systems, have long-standing data-parallel pipelines that consist of stream transformations, where instead of splitting streams into micro-batches, operators retrieve individual tuples and continuously generate output tuples. SPEs such as Apache Flink [37] and Storm [32] use this approach. However, usually left out of this discussion is the fact that in both models, the data already arrives in so-called nano-batches from other systems. Thus, regardless of the specific execution model, it is possible to perform batch processing even in tuple-at-a-time systems.

Additionally, in the last decades SPEs have been developed in two different directions: As scale-out as well as scale-up systems. Scale-out systems are optimized for executing streaming queries on distributed, heterogenous, shared-nothing architectures and thus are of limited relevance to our analyses. Of much more interest to us are scale-up systems, which are optimized for execution on a single machine to efficiently utilize the capabilities of modern high-end systems. Current SPEs in this category include Streambox [26], Trill [13], and SABER. Streambox in particular aims to optimize execution for multicore machines, while SABER considers heterogeneous processing through the use of GPUs. Finally, Trill scores high on flexibility, supporting a wide range of queries with SQL as well as streaming queries. However, all of these systems focus primarily on optimizing computational performance, but not on optimizing memory patterns and access.

4.2 Column Stores

The MonetDB systems [8, 10] pioneered the development of modern column-oriented database systems and vectorized query execution. As a result, a wide range of columnar database systems was created to follow this approach [17, 29, 31, 40]. Liarou et al. [25] already showed that column-oriented designs can significantly outperform row-based databases in standardized benchmarks such as TPC-H due to superior CPU and cache performance [1].

The work of Sikka et al. [29] is thereby of particular interest. Not only have they presented further optimizations for column stores, but they have also disproved the myth that column stores are better than row stores only for OLAP queries. For increasingly complex OLTP queries with even newly designed and more complex benchmarks like TPC-E [29, 33, 34], the nano-batch sizes in the streaming copuld already contain enough tuples, so that column stores deliver better performance than row stores.

A detailed comparison of how column-oriented and row-oriented databases compare to each other was given by Abadi et al. [1], however, showing that simulating a row store in a column store does not yield good results. While vectorization of specific parts of the code can improve performance dramatically [20], Harizopoulos et al. [18] compare the performance of a row and column store built from scratch and demonstrates that in a carefully controlled environment, column stores still outperform row stores.

5 Project Plan

Please find a timetable with major milestones here.

Time	Writing/Research	Implementation
	– Background	– Microbenchmark setup
Aug – Sep	– Related Work	– Relational Operators
		– Column/Row Transformation
	– Microbenchmarks - Benefits	– Explicit SIMD
Sep – Oct	– Microbenchmarks - Costs	– Memory Architectures
	– Microbenchmarks - Combined	– Window Types
Nov – Jan	– Evaluation	– Multithreading
		– Integration into Darwin
	– Abstract	
Jan	– Conclusion	
	– Proofread	

Table 1: Planned Time Table

References

[1] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 967–980. ACM, 2008. doi: 10.1145/1376616.1376712. URL `https://doi.org/10.1145/1376616.1376712`.

[2] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 169–180. Morgan Kaufmann, 2001. URL `http://www.vldb.org/conf/2001/P169.pdf`.

[3] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, 2015. doi: 10.14778/2824032.2824076. URL `http://www.vldb.org/pvldb/vol8/p1792-Akidau.pdf`.

[4] Alibaba. Four billion records per second! `www.alibabacloud.com/blog/four-billion-records-per-second-stream-batch-integration-implementation-of-alibaba-cloud-realtime-compute-for-apache-flink-during-double-11_596962`, 2020.

[5] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum, editors, *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 119–130. ACM, 2010. doi: 10.1145/1807128.1807148. URL `https://doi.org/10.1145/1807128.1807148`.

[6] Lawrence Benson and Tilmann Rabl. Darwin: Scale-in stream processing. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022. URL `https://www.cidrdb.org/cidr2022/papers/p34-benson.pdf`.

[7] Lars Jonas Bollmeier, Björn Daase, and Richard Ebeling. Processor-specific stream processing query compilation. Technical report, Hasso Plattner Institute, University of Potsdam, 2021.

[8] Peter A. Boncz and Martin L. Kersten. MIL primitives for querying a fragmented world. *VLDB J.*, 8(2):101–119, 1999. doi: 10.1007/s007780050076. URL `https://doi.org/10.1007/s007780050076`.

[9] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 54–65. Morgan Kaufmann,

1999. URL http://www.vldb.org/conf/1999/P5.pdf.

[10] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 225–237. www.cidrdb.org, 2005. URL http://cidrdb.org/cidr2005/papers/P19.pdf.

[11] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015. URL http://sites.computer.org/debull/A15dec/p28.pdf.

[12] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 363–375. ACM, 2010. doi: 10.1145/1806596.1806638. URL https://doi.org/10.1145/1806596.1806638.

[13] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, John C. Platt, James F. Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proc. VLDB Endow.*, 8(4):401–412, 2014. doi: 10.14778/2735496.2735503. URL http://www.vldb.org/pvldb/vol8/p401-chandramouli.pdf.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In Eric A. Brewer and Peter Chen, editors, *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150. USENIX Association, 2004. URL http://www.usenix.org/events/osdi04/tech/dean.html.

[15] The Apache Software Foundation. Nexmark benchmark suite. https://beam.apache.org/documentation/sdks/java/testing/nexmark/, 2022.

[16] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. Speculative distributed CSV data parsing for big data analytics. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 883–899. ACM, 2019. doi: 10.1145/3299869.3319898. URL https://doi.org/10.1145/3299869.3319898.

[17] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. HYRISE - A main memory hybrid storage engine. *Proc. VLDB Endow.*, 4(2):105–116, 2010. doi: 10.14778/1921071.1921077. URL http://www.vldb.org/pvldb/vol4/p105-grund.pdf.

[18] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel Madden. Performance tradeoffs in read-optimized databases. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 487–498. ACM, 2006. URL http://dl.acm.org/citation.cfm?id=1164170.

[19] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012. URL `http://sites.computer.org/debull/A12mar/monetdb.pdf`.

[20] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, 2018. doi: 10.14778/3275366.3275370. URL `http://www.vldb.org/pvldb/vol11/p2209-kersten.pdf`.

[21] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter R. Pietzuch. SABER: window-based hybrid stream processing for heterogeneous architectures. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 555–569. ACM, 2016. doi: 10.1145/2882903.2882906. URL `https://doi.org/10.1145/2882903.2882906`.

[22] Jay Kreps. Kafka : a distributed messaging system for log processing. 2011.

[23] Geoff Langdale and Daniel Lemire. Parsing gigabytes of JSON per second. *VLDB J.*, 28(6):941–960, 2019. doi: 10.1007/s00778-019-00578-5. URL `https://doi.org/10.1007/s00778-019-00578-5`.

[24] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. Mison: A fast JSON parser for data analytics. *Proc. VLDB Endow.*, 10(10):1118–1129, 2017. doi: 10.14778/3115404.3115416. URL `http://www.vldb.org/pvldb/vol10/p1118-li.pdf`.

[25] Erietta Liarou, Stratos Idreos, Stefan Manegold, and Martin L. Kersten. Monetdb/datacell: Online analytics in a streaming column-store. *Proc. VLDB Endow.*, 5(12):1910–1913, 2012. doi: 10.14778/2367502.2367535. URL `http://vldb.org/pvldb/vol5/p1910_eriettaliarou_vldb2012.pdf`.

[26] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. Streambox: Modern stream processing on a multicore machine. In Dilma Da Silva and Bryan Ford, editors, *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, pages 617–629. USENIX Association, 2017. URL `https://www.usenix.org/conference/atc17/technical-sessions/presentation/miao`.

[27] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Instant loading for main memory databases. *Proc. VLDB Endow.*, 6(14):1702–1713, 2013. doi: 10.14778/2556549.2556555. URL `http://www.vldb.org/pvldb/vol6/p1702-muehlbauer.pdf`.

[28] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. Filter before you parse: Faster analytics on raw data with sparser. *Proc. VLDB Endow.*, 11(11):1576–1589, 2018. doi: 10.14778/3236187.3236207. URL `http://www.vldb.org/pvldb/vol11/p1576-palkar.pdf`.

[29] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis

Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 731–742. ACM, 2012. doi: 10.1145/2213836.2213946. URL https://doi.org/10.1145/2213836.2213946.

[30] Elias Stehle and Hans-Arno Jacobsen. Parparaw: Massively parallel parsing of delimiter-separated raw data. *Proc. VLDB Endow.*, 13(5):616–628, 2020. doi: 10.14778/3377369.3377372. URL http://www.vldb.org/pvldb/vol13/p616-stehle.pdf.

[31] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented DBMS. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 553–564. ACM, 2005. URL http://www.vldb.org/archives/website/2005/program/paper/thu/p553-stonebraker.pdf.

[32] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy V. Ryaboy. Storm@twitter. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 147–156. ACM, 2014. doi: 10.1145/2588555.2595641. URL https://doi.org/10.1145/2588555.2595641.

[33] Pinar Tözün, Ippokratis Pandis, Cansu Kaynak, Djordje Jevdjic, and Anastasia Ailamaki. From A to E: analyzing tpc's OLTP benchmarks: the obsolete, the ubiquitous, the unexplored. In Giovanna Guerrini and Norman W. Paton, editors, *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 17–28. ACM, 2013. doi: 10.1145/2452376.2452380. URL https://doi.org/10.1145/2452376.2452380.

[34] TPC. Tpc-e. http://www.tpc.org/tpce, 2010.

[35] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. Nexmark—a benchmark for queries over data streams draft. Technical report, Technical report, OGI School of Science & Engineering at OHSU, Septembers, 2008.

[36] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association, 2012. URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[37] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016. doi: 10.1145/2934664. URL http://doi.acm.org/10.1145/2934664.

[38] Steffen Zeuch, Sebastian Breß, Tilmann Rabl, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, and Volker Markl. Analyzing efficient stream processing on modern hardware. *Proc. VLDB Endow.*, 12(5):516–530, 2019. doi: 10.14778/3303753.3303758. URL http://www.vldb.org/pvldb/vol12/p516-zeuch.pdf.

[39] Shuhao Zhang, Bingsheng He, Daniel Dahlmeier, Amelie Chi Zhou, and Thomas Heinze. Revisiting the design of data stream processing systems on multi-core processors. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 659–670. IEEE Computer Society, 2017. doi: 10.1109/ICDE.2017.119. URL https://doi.org/10.1109/ICDE.2017.119.

[40] Marcin Zukowski, Mark van de Wiel, and Peter A. Boncz. Vectorwise: A vectorized analytical DBMS. In Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 1349–1350. IEEE Computer Society, 2012. doi: 10.1109/ICDE.2012.148. URL https://doi.org/10.1109/ICDE.2012.148.