Query Planning for Transactional Stream Processing on Heterogeneous Hardware: Opportunities and Limitations

– Novel Ideas & Experience Reports –

Philipp Götze[1] Constantin Pohl[1] Kai-Uwe Sattler[1]

Abstract: In a heterogeneous hardware landscape consisting of various processing units and memory types, it is crucial to decide which device should be used when running a query. There is already a lot of research done for placement decisions on CPUs, coprocessors, GPUs, or FPGAs. However, those decisions can be further extended for the various types of memory within the same layer of the memory hierarchy. For storage, a division between SSDs, HDDs or even NVM is possible, whereas for main memory types like DDR4 and HBM exist. In this paper, we focus on query planning for the transactional stream processing model. We give an overview of several techniques and necessary parameters when optimizing a stateful query for various memory types, outlined with chosen experimental measurements to support our claims.

Keywords: Stream Processing, Transactions, Cost Model, Xeon Phi, NVM, Non-Volatile Memory

1 Introduction

Technological advance has lead to a high degree of specialization in terms of hardware. Instead of a single processing device being capable of dealing with any requirements and applications, specialized variants improve performance more than a general approach would. GPUs and many-core CPUs come with an intense computational power through parallelism, FPGAs allow reconfigurations of functionality, NVM technology could provide persistence on main memory speed, HBM greatly increases available bandwidth for memory-bounded applications, etc. This heterogeneous landscape of hardware increases the possible search space to come to an optimal query plan, though. Therefore, it is important to reduce the complexity by isolating the factors that mostly influence query performance.
In this paper, we look at the possibilities of query planning for the transactional stream processing model, highlighting opportunities but also limitations of different approaches. In addition to the actual execution plan, the physical representation of states must also be considered in this model. Thus, the following aspects must be taken into account when selecting queries: ① the state representations (the underlying data structures), ② the data placement (on which medium the data is stored), and ③ the algorithms (i.e. the appropriate

[1] TU Ilmenau, Databases & Information Systems Group, Ilmenau, Germany, *first.last*@tu-ilmenau.de

doi:10.18420/btw2019-ws-05

implementation of the operators). All three points play closely together and can hardly be considered separately. The questions that arise from this are, on the one hand, which parameters are necessary for a suitable selection and, on the other hand, which of them are actually accessible. Since stream queries usually run on a long-term basis, data rates and characteristics (e.g., skew) may also change over time. This raises the further question whether plans should, therefore, be repeatedly adjusted or rather a plan should be as robust as possible right from the start.

2 Related Work

Query Planning. Cost models and query execution planning is a widely studied topic in the DBMS world. Therefore, we will only discuss the most profound and relevant work for us in the following. The work of Manegold [Ma02] was one of the first proposing a hardware-based cost model for modern CPUs considering sophisticated features like cache hierarchies. It investigated the typical memory access pattern of database operations and distinguished between logical, algorithmic, and physical costs. Sixteen years later, Zeuch [Ze18] has been working extensively on query planning for today's generation of CPUs that employ even more utilization techniques. A counter-based approach was proposed that progressively optimizes the queries at runtime. Krämer [Kr07] have dealt intensively with continuous queries and a corresponding cost-based resource management especially for sliding windows. Here, an adaptive approach was chosen too, where the window size is dynamically adjusted to the available resources within predefined bounds. Karnagel et al. [Ka17] discuss another adaptive approach to optimally utilize query execution on heterogeneous hardware. Their focus is on the optimal placement of work on compute units, particularly for OpenCL-based DBMSs.
It was found that little work has been done on a cost model for data stream processing. At the same time, the question arises which concepts of DBMSs are applicable to transactional stream processing. More details to these and further approaches are discussed in Sect. 4.

Transactional Stream Processing. The STREAM [Mo03] project was one of the first to deal with the combination of relational databases and stream processing. They have considered continuous queries, possibly with a synopsis (≡ state), and provide a way to share subplans and states. The plan selection is based on stream constraints with the sole goal of minimizing the use of resources. Although constraints can be modeled with the help of punctuations, there is no actual transaction management. Instead, a global scheduler coordinates the successive execution of the individual operators by assigning time slots.
A more recent system that supports transactions in data stream processing is S-Store [Me15], which is implemented on top of the main memory OLTP system H-Store [Ka08]. It can guarantee the ACID properties by reapplying the existing transaction concepts of H-Store for time-varying relations on, e.g., windows and streams. Queries over streams are expressed as dataflow graphs, where each node represents a stored procedure and the edges define the

execution order. Each execution of a stored procedure combined with an input batch forms a transaction. Although it is not addressed directly, it can be assumed that query planning was also inherited from H-Store.
Botan et. al. [Bo12] have defined a unified transactional model to combine traditional transaction and data stream processing. This is realized by transforming continuous queries into a sequence of one-time queries and, thus, treating streams and relations uniformly. They implemented this model on top of an existing storage manager and extended it by a transaction management component dealing with concurrency and failures. Continuous query planning, however, was not really addressed here either.

3 Background

3.1 Transactional Stream Processing

The transactional stream processing model considered in this context distinguishes between two types of data occurrence: *tables* to represent states and *streams* for data flowing through the queries. Similar to the relational model, tables are a finite collection of data divided into rows and columns. Streams, on the other hand, can be potentially infinite and are typically defined as a sequence of tuples. While streams are volatile, tables also require a physical representation.
Additionally, operators are needed to link these two concepts. We divide them into three classes: *TO_TABLE* which updates tuples from a stream in a table, *TO_STREAM* producing a stream of tuples based on either events in a table or the whole table, and *FROM* to attach to a stream or to read data from a table. These operators are illustrated in Fig. 1.

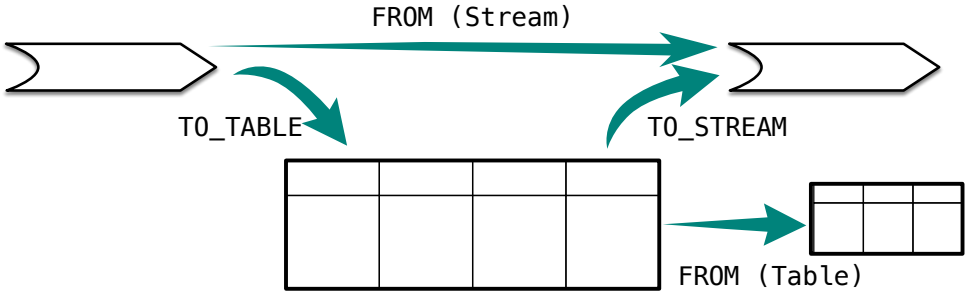


Fig. 1: Overview of the transactional semantics for data stream processing.

Manipulations of states and streams is performed within a transactional context to ensure atomicity and durability during writing. In order to guarantee atomicity, transaction boundaries are required, which could be, for instance, automatically defined per stream element or propagated alongside the actual data via punctuations. *FROM* and *TO_STREAM* operators, on the other hand, provide different isolation levels where the latter requires an additional trigger policy.

3.2 Hardware Considerations

Today's hardware landscape is becoming increasingly heterogeneous and there is a development from general-purpose processors to special hardware for specific applications or operators. This means that the corresponding application scenarios must first be identified in our model. Since GPUs can process massively independent data in parallel, but have comparatively high shipping costs, it is questionable whether they are suitable for stream processing at all or whether SIMD registers are sufficient. Possible scenarios could be ad-hoc queries on very large states or linear algebra operators for matrix calculations. FPGAs, on the other hand, can already be soldered on CPU sockets and, thus, would not have too high shipping costs. These could, e.g., be programmed for special operators or operator pipelines which are highly CPU bound [Mü09]. Another aspect of heterogeneous hardware are modern high-speed networks and technologies such as Infiniband, RoCE, and RDMA respectively, which can be advantageous for distributed data management systems [Bi18]. Here, however, we focus on local stateful operations, which are more likely to be memory or storage bound. Therefore, we aim our attention regarding modern hardware at Non-Volatile Memory (NVM) and many-core processors, which we will describe briefly in the following.

Many-Core Architecture. Since there are physical limits to the clock rate in the development of CPUs, nowadays the number of integrated cores is increased to enable high parallelism. In contrast to multi-core CPUs, the many-core architecture offers even more features for parallelization, such as more cores, each with hyperthreading, and wider SIMD registers. However, the high number of densely packed cores creates a strong heat, which is counteracted by simpler cores and a lower clock speed which in turn leads to poorer singlethreaded performance. The Intel® Xeon Phi™ KNL, for instance, has up to 72 cores (@ 1.5 GHz). In addition, the KNL features multi-channel DRAM (MCDRAM) which is a variant of high-bandwidth memory (HBM), to increase the bandwidth up to 400 GB/s. For query planning, the degree of parallelization and the partitioning of states is of particular interest. It is also important to consider whether it is worth using HBM and in which mode.

Non-Volatile Memory. So that states in our model can survive power loss or system failures, the data must be stored non-volatile. Whereas in the traditional way of persisting data, I/O has accounted for the majority of the cost, this could change significantly with the advent of NVM. This technology promises to combine the byte-addressability and low latency of DRAM with the persistence and density of block-based storage media. However, NVM suffers from a limited cell endurance and read-write asymmetry regarding the latency. These characteristics mean for an optimizer that storage accesses are no longer clearly dominating the costs. Furthermore, when selecting an execution plan and the data representation of the states, the properties of the device must be considered. For NVM, writes should be exchanged for multiple reads if possible to counteract the lower endurance and read-write asymmetry. In addition, byte-addressability can be exploited with regard to atomicity.

4 Continuous Query Planning

As stated earlier, when optimizing continuous queries not only the execution plan but also the physical schema design must be determined. This applies in particular to queries with stateful operators. Depending on the application, there can be various optimization goals, such as low latency, high throughput or resource efficiency. Achieving these goals depends on various factors. The objective is to identify the most influential of them. Therefore, we first look at important parameters for query planning for stateful operators. We then discuss various approaches how an optimizer can get and use these to construct suitable plans.

4.1 Stateful Processing

There are basically three scenarios when a state needs to be accessed with ACID guarantees: (a) persisting modifications to a state, (b) querying a shared state, and (c) recovering a state after a failure. Depending on the ratio and frequency with which these scenarios occur, combined with their access profile, several types of states are sometimes more suitable than others. State representations could be, e.g., a durable log, hash-based structures, tree-based structures, sorted arrays/lists, graphs, or even various combinations of these. In addition, there are available hardware and resource factors with appropriately optimized algorithms and data placement. This basically results in a huge decision space. The task of the optimizer is to select a suitable combination of state representation, data placement, and access algorithms. For this, it receives the requirement profile and the available resources as input and matches them with the state types and algorithms implemented in the system. Next, we will go into more detail about the some possible parameters.

Data-driven Parameters. Regarding the access profile, a distinction can be made primarily between dominant kinds of access (persisting, querying, recovery) and their ratio and frequency as described above. In a transactional system, there are additional very influential factors that affect the guarantee of ACID properties. For a shared state it is important to know the number of concurrent accesses as well as the approximate degree of contention. Based on this, an appropriate concurrency control protocol would have to be selected. The question with data stream processing is where to get this information when creating a query if no data has been seen yet. One possibility would be to rely on statistics and heuristics based on previous queries. A modified form of sampling, if feasible, would also be conceivable. Since a state is always part of an operator, it may be possible to derive typical access patterns from it. It becomes more complicated if the state is defined via a general purpose table. In this case, user annotations would be helpful. In order not to burden the users with the task, a dynamic adaptation based on the actual workload might be a better approach. However, this can lead to a situation in which another data representation or placement is suddenly classified as much more efficient. Therefore, a potentially expensive state conversion or migration may be worthwile. This would throttle the performance for

a certain time, especially during a high demand for the affected state. This is why robust query plans are often used that can withstand a wide range of scenarios.

Hardware-based Parameters. In addition to data-related parameters, there are also hardware-based factors that can be indispensable for determining the optimal state representation, data placement, and access algorithms. The first relevant factor is the general availability of specific processing units (e.g., FPGA, GPU, many-core CPU) or, regarding our focus, memory and storage variants (e.g., HBM, NVM, SSD, HDD). Moreover, the available numbers of capacity, latency, and bandwidth are interesting with regard to storage media. For example, due to limited space or performance reasons, a data structure could be stored across multiple memory layers, as it is the case with the LSM-tree or hybrid structures such as the FPTree [Ou16]. Sticking with LSM-trees, a different buffering strategy could be chosen depending on the memory type [Le17]. Furthermore, depending on the access granularity of the devices, the basic organization (blocks, pages, tuples, etc.) of the data must be taken into account. For PUs, on the other hand, the degree of parallelizability and the clock frequency are important. So it is possible to use different partitioning approaches here to leverage the heterogeneous units. But the transport and merge times of a co-processor must also be included in order to determine the profitability (cf. [Po17]). In practice, it becomes even more complicated, especially with modern processors since sophisticated techniques like caching, prefetching, branch prediction, reordering, etc. are involved (cf. [Ze18]). Below, we outline how different approaches could acquire and use these parameters.

4.2 Optimizing Strategies

Existing Models. In the following, we discuss three classes of optimizing approaches, which we categorize as hardware-oblivious, hardware-conscious, and learning models. Cost models without hardware-related parameters (being *hardware-oblivious*) were relatively common in the early database systems. Such models are usually application-based [LN96], which means that a DBMS is manually tuned to the underlying hardware by identifying and re-implementing major performance bottlenecks [WK90]. This allows to reduce the number of parameters and, thus, the search space of cost models. However, a disadvantage of this approach is that a change of hardware or software requires manual adjustments again.
A cost model that depends on hardware parameters like memory access latencies or cache sizes is more robust against changes in general (being *hardware-conscious*). The usual way to get this hardware information is by running a calibration tool [Ma02] whenever hardware changes. The difficulty of such an approach is mainly to correlate those parameters correctly since they can influence each other. To point an example, a higher clock frequency also leads to a reduced latency of main memory accesses.
Recent research focuses on the combination of database models and machine learning. By feeding parameters into well-trained machine learning models, manual tuning and cost model adaptations are unnecessary. In [Or18], they applied deep reinforcement learning to

incrementally find the optimal query execution plan through subqueries. However, learned query optimization models can become difficult to configure correctly and also represent a black box, which makes it difficult to interpret how a certain result is achieved.

Strategy for Transactional Stream Processing. For the transactional stream processing model, an overall solution could be that the state type is determined by the access pattern (e.g., using operator characteristics) and the optimizations regarding data placement and algorithms are set by the hardware factors. Whether this is determined by a learning or a concrete cost model still has to be shown. As we have seen, there is a plethora of parameters, and only the most influential factors should be considered to avoid making the model too complex. Moreover, not all information is available at all times, which means that certain procedures may be excluded in advance. Therefore, we think an adaptive or progressive optimization might be quite reasonable.
As a concrete proposal, we extend our cost formula for stateful operations from [Po17]. Calibrated hardware parameters are assumed. Since we include NVM in the consideration, we separate read and write accesses. We further differentiate between state ($f_{<s>}$) and operator ($f_{<op>}$) dependent cost factors. The former contains the average state-specific and additional synchronisation (ACID) costs per access and the latter the logical operator-specific read or write accesses. The first factor must also be determined depending on the underlying hardware. This can be done either by another cost formula based on, e.g., the devices latency or by using performance counters (cf. [Ze18]). Using these factors, we provide a formula for every kind of state access (persisting, querying, and recovery) in Eqs. (1) to (3).

$$c_{<op>} = f_{<op>_r} \cdot f_{<s>_r} + f_{<op>_w} \cdot f_{<s>_w} \tag{1}$$

$$c_{<s>\_q} = \sigma \cdot <s>_{size} \cdot f_{<s>_r} \tag{2}$$

$$c_{<s>\_rec} = \Delta \cdot <s>_{size} \cdot (f_{<s>_r} + f_{<s>_w}) \tag{3}$$

Whereas the first depends on the operator, the other two cost formulas are rather state dependent ($<s>$). The persistence costs consist of the read and write costs for the operator-typical access and depend on the underlying state and hardware. State querying is similar to typical cost models and, thus, cardinality and selection (σ) based. For recovery, we can differentiate between three state dependent cases. The first case is when no recovery is necessary for the state (only atomically visible changes), then Δ and the total cost is zero. As typical in DBMS, however, certain checkpoints could also be set, whereby only changes up to the last checkpoint need to be considered. This distance (Δ) could, for instance, be given as a percentage of the total size. In the worst case ($\Delta = 1$), if there are no checkpoints, the entire state may have to be read and parts rewritten or removed. The precision of this model in practice still has to be examined in future work.

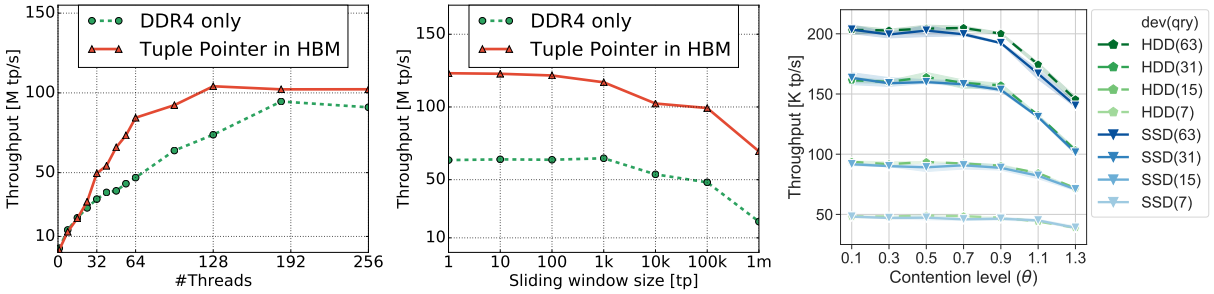
5 Experiments

The question we want to address in this section is whether it is possible to predict performance behavior by a cost model on the memory layer. For reasons of space, we pick two comparisons: DDR4-HBM and HDD-SSD, examining the performance of a sliding window operation and an LSM-Tree. We expect a higher bandwidth to result in improved sliding window performance if the rest of a query has few computations or the degree of inter-query parallelism is high (hence more bandwidth is used). For the LSM-Tree, its structure as well as modern prefetching and caching techniques should mask disk characteristics, almost eliminating performance differences.
The experiments run on the Intel® Xeon Phi™ Knights Landing 7210 with 64 cores à 4 threads @ 1.5 GHz (max), 96 GB DDR4, Linux kernel 3.10, and GCC 7.3. We compare different performance metrics regarding memory (DRAM vs. HBM) and storage technologies (HDD vs. SSD). Since we can only emulate NVM (using DRAM) so far, we have omitted this technology from the experiments for now. The tests run with our data stream processing framework PipeFabric[2] with prototypical transaction support.

5.1 Sliding Window

A sliding window is a common operator for data stream processing. It keeps track of incoming tuples, invalidating older ones to keep only the newest tuples available. The sliding effect allows to continuously update the window tuple by tuple by *sliding* over the input stream. In PipeFabric, a sliding window is a list of tuples where new entries are added at the end of the list. Older tuples are removed by a count- or time-based check afterwards. If the list is small enough to fit in one of the caches, fast access can be guaranteed. In addition, higher memory bandwidth can be useful for pushing down recent changes of the list from cache to main memory (inclusive caching). Furthermore, throughput of a window depends on additional operators that a thread has to run. The more computations per tuple must be done, the less bandwidth can be utilized. Fig. 2 shows the performance of inserts and deletes of a sliding window operator with (a) varying thread numbers (each thread running an own sliding window operator) and (b) varying the window size.
When inter-query parallelism gets higher (with more than 20 threads), the HBM can sustain a higher throughput on average for each sliding window, as expected. Increasing the window state size to keep track of more tuples leads to performance degradation, since the state cannot be kept in the CPU caches (inserts at the end, deletes at the front). The number of TLB and page misses also increases when more than 100k tuples per window are stored. While throughput behavior can be predicted for varying window sizes, it is hard to anticipate the query workload in real systems due to the interference of heterogeneous queries. Memory access can differ fundamentally between operators, not to mention UDFs with unpredictable behavior before execution. We therefore suggest a calibration approach for the available stream operators like in our previous work [Po17] as a possible approximation.

[2] PipeFabric: `https://github.com/dbis-ilm/pipefabric`



(a) Throughput with varying thread count (10k window size).

(b) Throughput with varying window size (64 threads).

(c) LSM-tree with varying devices (dev) and ad-hoc queries (qry).

Fig. 2: Transactional Stream Processing experiments.

5.2 LSM-Tree as State Representation

This experiment aims to show that hardware and data structures are interdependent and that it is essential which and where the hardware parameters are added to the model. Therefore, we run the tests in our transactional stream processing framework having one continuous and multiple ad-hoc queries accessing the same states. As state representation we used an LSM-tree (RocksDB[3]) and varied the number of ad-hoc queries and the contention level. Due to the nature of LSM-trees where write costs are amortized over time, we expect little difference in throughput when storing it on an HDD and an SSD respectively. Theoretically, the SSD is 10 times faster than the HDD. The results are shown in Fig. 2c.
As expected the performance for HDD and SSD is nearly the same. Instead, the concurrency control protocol and the contention have more influence for such a data structure. This also raises the question whether the use of NVM is worthwhile in this case. However, with other data structures or queries that are, e.g., more bandwidth dependent, the selected device could have a severe impact. Thus, there is an interdependence between hardware and data structures which needs to be determined in the state-specific factor ($f_{<s>}$).

6 Conclusion

There have already been a number of studies on query planning in the DBMS and DSMS world. For transactional stream processing, the task is to unify them into one model that processes both streams and tables. Since there is hardly any apriori knowledge about the data in streams, a progressive approach seems to be useful. However, as constant changes can be very expensive, we think that a combination of adaptive and robust query planning is necessary. In this paper, we have briefly discussed which data- and hardware-based parameters can be chosen for this and how they can be used. In doing so, we drew attention to the opportunities and limitations of existing approaches and have underpinned these with first experiments. A fully functional model is part of future work.

[3] RocksDB (version 5.15.10): `https://github.com/facebook/rocksdb`

Acknowledgments

This work was partially funded by the German Research Foundation (DFG) within the SPP2037 under grant no. SA 782/28.

References

[Bi18] Binnig, C.: Scalable Data Management on Modern Networks. Datenbank-Spektrum 18/3, pp. 203–209, 2018.

[Bo12] Botan, I. et al.: Transactional Stream Processing. In: EDBT. Pp. 204–215, 2012.

[Ka08] Kallman, R. et al.: H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. PVLDB 1/2, pp. 1496–1499, 2008.

[Ka17] Karnagel, T. et al.: Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources. PVLDB 10/7, pp. 733–744, 2017.

[Kr07] Krämer, J.: Continuous Queries over Data Stream - Semantics and Implementation, PhD thesis, University of Marburg, Germany, 2007.

[Le17] Lersch, L. et al.: An analysis of LSM caching in NVRAM. In: DaMoN@SIGMOD. 9:1–9:5, 2017.

[LN96] Listgarten, S.; Neimat, M.: Modelling Costs for a MM-DBMS. In: RTDB. Pp. 72–78, 1996.

[Ma02] Manegold, S.: Understanding, Modeling, and Improving Main-Memory Database Performance, PhD thesis, 2002.

[Me15] Meehan, J. et al.: S-Store: Streaming Meets Transaction Processing. PVLDB 8/13, pp. 2134–2145, 2015.

[Mo03] Motwani, R. et al.: Query Processing, Approximation, and Resource Management in a Data Stream Management System. In: CIDR. 2003.

[Mü09] Müller, R. et al.: Data Processing on FPGAs. PVLDB 2/1, pp. 910–921, 2009.

[Or18] Ortiz, J. et al.: Learning State Representations for Query Optimization with Deep Reinforcement Learning. In: DEEM@SIGMOD. 4:1–4:4, 2018.

[Ou16] Oukid, I. et al.: FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In: SIGMOD. Pp. 371–386, 2016.

[Po17] Pohl, C. et al.: A Cost Model for Data Stream Processing on Modern Hardware. In: ADMS@VLDB. 2017.

[WK90] Whang, K.; Krishnamurthy, R.: Query Optimization in a Memory-Resident Domain Relational Calculus Database System. TODS 15/1, pp. 67–95, 1990.

[Ze18] Zeuch, S.: Query Execution on Modern CPUs, PhD thesis, 2018.