Real Time Processing of Streaming and Static Information

Christoforos Svingos[1,*], Theofilos Mailis[1], Herald Kllapi[1,2], Lefteris Stamatogiannakis[1], Yannis Kotidis[3], Yannis Ioannidis[1]
{*csvingos, tmailis, herald, estama, yannis*}*@di.uoa.gr, kotidis@aueb.gr*
[1]*Dept. of Informatics and Telecomunications, University of Athens, Greece.* [2]*currently at Google.*
[3]*Dept. of Informatics, Athens University of Economics and Business, Greece.*

***Abstract*—*Big Data* applications require real-time processing of complex computations on streaming and static information. Applications such as the diagnosis of power generating turbines require the integration of high velocity streaming and large volume of static data from multiple sources. In this paper we study various optimisations related to efficiently processing of streaming and static information. We introduce novel indexing structures for stream processing, a query-planner component that decides when their creation is beneficial, and we examine precomputed summarisations on archived measurements to accelerate streaming and static information processing. To put our ideas into practise, we have developed EXASTREAM, a data stream management system that is scalable, has declarative semantics, supports user defined functions, and allows efficient execution of complex analytical queries on streaming and static data. Our work is accompanied by an empirical evaluation of our optimisation techniques.**

***Keywords*-Stream Processing, SQL, Static Data, Performance**

I. INTRODUCTION

Emerging *Big Data* applications require real-time processing of complex computations on streaming and static information. The latter is a challenging task since it involves the integration of high velocity streaming and large volume of static data from multiple sources, on many concurrent continuous queries that need to be executed.

A typical scenario described in [1] requires monitoring and diagnosing of power-generating turbines. In the described scenario, several service centres are dedicated to diagnosing by utilizing data from more than $100,000$ thermocouple sensors installed in 950 power generating turbines located across the globe. One typical task of such a centre is to detect in real-time potential faults of a turbine caused by, e.g., an undesirable pattern in temperature's behaviour within various components of the turbine. This task requires to extract, aggregate, and correlate *(i)* streaming data produced by up to $2,000$ sensors installed in different parts of the turbine, *(ii)* static data about the turbine's structure, *(iii)* and historical operational data of each sensor stored in multiple datasources.

This need has triggered the design of scalable approaches that provide low latency answering to queries on high-velocity live streams and high-volume static data sources. In this paper we study several novel optimisation techniques for efficiently processing analytical queries on streaming & static information. In particular: *(i)* we introduce novel in-memory indexing structures and algorithms dedicated to accelerating stream-processing; *(ii)* we propose the *adaptive stream indexing* technique that is responsible for creating on the fly the appropriate indexing structures that will accelerate execution of live-stream operations.

To put our ideas into practice, we have developed the EXASTREAM *Data Stream Management Systems* (DSMS), an experimental DSMS that fuses streaming operators to the SQLite database engine. EXASTREAM has several significant features such as: *(i) scalability:* the ability to run our system in a distributed environment and its capacity to easily add and remove queries without disrupting existing query execution; *(ii) declarative semantics:* our system provides for a declarative language, extending the SQL syntax and semantics for querying live streams and relations; *(iii) user defined functions:* our system natively supports *user defined functions* with arbitrary user code; *(iv) stream and static data integration:* based on its architecture and implementation, our system natively supports streaming and static data integration. It should be noted that the optimisations we have proposed are general optimisations that can be adopted by other stream processing systems as well.

In our experimental evaluation we study the effect of the proposed optimisations in a cloud deployment of EXASTREAM on up to 128 nodes using real sensor data from power generating turbines. Our findings demonstrate the effectiveness of our techniques in processing up to 1 thousand live stream queries and performing correlation analysis between live and archived stream measurements in real time.

II. SYSTEM OVERVIEW

The EXASTREAM *Data Stream Management System* (DSMS) has been designed for efficiently processing on both static and streaming information. It is embedded in EXAREME (https://www.exareme.org), a system for elastic large-scale dataflow processing on the cloud [2], [3] that has been publicly available as an open source project under the MIT License. EXASTREAM was implemented as a key

* This research has been partially supported by the EU project Optique (FP7-IP-318338).


```
CREATE STREAM read_from_source AS
SELECT *
FROM (timeslidingwindow timewindow:3 frequency:3
      SELECT *
      FROM (http 'http://ip:port/stream1'));

CREATE STREAM avg_value AS
SELECT wid, AVG(value)
FROM read_from_source
GROUP BY wid;

SELECT * FROM avg_value;
```

Figure 1: A simple (a) ExaStream topology and (b) its syntactical representation

component of the Optique project. This section introduces some key aspects of ExaStream before presenting the optimisations that we have built upon it.

Data Model: A *topology* describes the flow of streaming and static records between *computational nodes*. Computational nodes are logical processing units that have one or more live-stream or static-data inputs and one output. They execute a set of *operations* on their input to produce the corresponding output. Computational nodes can be classified as either having exclusively *live-stream inputs*, exclusively *static-data inputs*, and *hybrid inputs*. Similarly, they can be classified to being *streaming* or *static*, based on the form of their output. A special type of computational nodes are those responsible for communicating external sources to our topology, similar to Storm's spouts.

Declarative Semantics for Computations: ExaStream takes advantage of existing Database Management technologies and optimisations by providing a declarative language, namely SQL$^{\oplus}$, extending the SQL syntax and semantics for querying live streams and relations. In contrast to most DSMSs, the user does not need to consider low-level details of query execution. Instead, the system's *query planner* is responsible for choosing an optimal plan depending on the query, the available stream/static data sources, and the execution environment.

In order to incorporate the algorithmic logic for transforming SQL into SQL$^{\oplus}$ several operators and statements have been implemented: *(i)* The *Create Stream* statement allows to add a new computational node to our topology that outputs a live stream. *(ii)* The *TimeSlidingWindow* groups tuples from the same time window and associates them with a unique window identifier corresponding to the `Wid` attribute. *(iii)* The *WCache* creates the indexing structures for answering efficiently equality constraints on the `Wid` and `Time` attributes when processing infinite streams.

Example 1. *Fig. 1a shows a simple topology. The input node receives information from a stream of temperature measurements acquired from a single sensor on some power generating turbine. The initial data contain the temperature measurement in Celcium degrees and the time that this measurement was acquired. The input node processes the records arriving from the source, acknowledges the temporal identifier indicated by the source, and relates each measurement to a time-sliding window mechanism that assumes a window of size* 3 *sec is produced every* 3 *sec. Then a second computational node calculates the average temperature value grouped by windows.*

In Fig. 1b we see the ExaStream *query, corresponding to the Fig. 1a topology. The* create stream *statement creates the two different computational nodes responsible for reading from the data source (*`read_from_source`*) and computing the average value per window (*`avg_value`*). As we see the* `read_from_source` *computational node uses two user defined functions: http reads the stream data that are pushed from an HTTP server; and timeslidingwindow is responsible for creating the windows based on the windowing mechanism expressed by the* timewindow *and* frequency *parameters. The frequency attribute defines that a window will be created every* 3 *secs and the timewindow defines that the length of the window is* 3 *secs. The* `avg_value` *computational node has* `read_from_source` *as its input and outputs a new stream that contains the average value per window. Final the select query is the one that shows the results of the* `avg_value` *stream.*

For the Optique project, ExaStream has been extended in order to support statements expressed in the STARQL language [4].

Architecture & Implementation: ExaStream supports *parallelism* by allocating processing across different workers in a distributed environment. Queries are registered through the Gateway Server. Each registered query passes through the ExaStream parser and then is fed to the Scheduler module. The Scheduler places data and compute operators (including UDFs and relational plans) on workers nodes based on each worker's load. These operators are executed by an SQLite (https://www.sqlite.org) database engine instance running on each worker.

ExaStream offers different types of parallelism depending on the type of operations performed within a query. Inter-query parallelism is supported for queries with an exclusively streaming input. This means that all the operations of a single query are executed on the same worker, while parallelism is achieved by distributing queries across workers. For computational nodes with a static input, ExaStream provides intra-query parallelism, i.e. each operation of a query is distributed on multiple workers.

Local node operations are handled by the *stream query planner*. ExaStream's query planner extends the one provided by SQLite in order to efficiently execute queries in a declarative language, such as SQL, without any concern for low-level execution details.

III. Query Optimisations on Live & Archived Streams

ExaStream queries access information from both live streams and static data sources. A special form of static data are archived-streams that, though static in nature, accommodate temporal information that represents the evolution of a stream in time. Therefore, our analytical operations can be classified as: *(i) live-stream operations* that refer to analytical tasks involving exclusively live streams; *(ii) static-data operations* that refer to analytical tasks involving exclusively static information; *(iii) hybrid operations* that refer to analytical tasks involving live-streams and static data that usually originate from archived stream measurements.

For static-data operations we rely on standard database optimisation techniques. This section focuses on the live-stream optimisations we have developed:

A. Indexing Structures

Considering the particularities of live-streams with infinite records we have developed hybrid in-memory indexing structures and algorithms dedicated to accelerating stream-processing. For visualisation purposes, we will assume a *3D space* describing each stream and corresponding to the attributes (`Wid`, `Time`, `Measurement`). The corresponding structures can be applied for higher dimensional spaces.

Our technique considers two levels of indexing: *(i)* the first level, namely $\texttt{WCacheL}_1$, is for performing fast equality operations on the `Wid` attribute based on an hybrid merge/hash-join algorithm *(ii)* the second level, namely $\texttt{WCacheL}_2$, is for accelerating operations on the rest of the attributes, i.e. `Time` and `Measurement` for our description.

1) $\texttt{WCacheL}_1$*:* The $\texttt{WCacheL}_1$ index related to a stream is used for efficiently answering equality constraints on its `Wid` attribute. In particular, we use the $\texttt{WCacheL}_1$ in-memory hash-index with `Wid` as key and the list of tuples that belongs to that specific `Wid` as values. Each bucket on $\texttt{WCacheL}_1$ stores `Wids` in a sorted order, while records on the live stream also appear sorted on the `Wid` attribute –this property of live streams is credited to the *timeslidingWindow* operator–.

Because a stream is infinite, we need a mechanism to ensure that our hash-structure *moves* forward in time. This mechanism adds wids to the $\texttt{WCacheL}_1$ index, as soon as they appear in the stream. Since live streams arrive sorted on the `Wid` attribute, the $\texttt{WCacheL}_1$ related to it can be easily updated by inserting each new *wid* to the bottom of its corresponding hash-bucket.

Example 2. *The left hand side of Figure 2 shows the* $\texttt{WCacheL}_1$ *level of indexing. Bucket 0 contains in sorted order all the wids that have appeared till now and are mapped to the value of* 0*, as we can see both wids in buckets and in the actual stream, are sorted on the* `Wid` *attribute.*

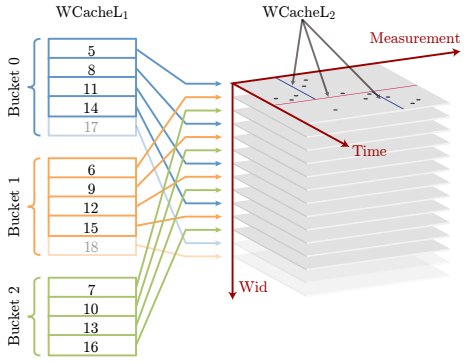

Figure 2: The $\texttt{WCacheL}_1$ and $\texttt{WCacheL}_2$ index structures

In Figure 2 the wids 17 *and* 18 *are added to the* 0 *and* 1 *buckets, as soon as they appear as records into our stream.*

We will demonstrate how our algorithm exploits the $\texttt{WCacheL}_1$ structure for a simple equi-join on two streams. The outer stream of the join operation makes a scan to its data and visits the $\texttt{WCacheL}_1$ of the inner one. If the outer stream scans the wid w and $\texttt{WCacheL}_1$ contains the finite set of wids denoted with $\mathcal{W}$ the following cases may occur: *(i)* $w \leq \max(\mathcal{W})$ and $w \notin \mathcal{W}$: In that case w does not appear as a value in the $\texttt{WCacheL}_1$-index and consequently in the `Wid` attribute of $\texttt{Stream}_{\texttt{inner}}$. Since values in $\texttt{Stream}_{\texttt{inner}}$ are ordered in `Wid`, we can safely assume that the window w will never appear as part of the inner stream and therefore the joining condition will never be satisfied for the w window. *(ii)* $w \in \mathcal{W}$: In that case we search the corresponding bucket of $\texttt{WCacheL}_1$ that contains the value of w. Since windows are stored in a sorted order per bucket, the algorithm searches for w using a merge-join algorithm. When w is found, our algorithm will return all the tuples in $\texttt{Stream}_{\texttt{inner}}$ that belong to the specific window. *(iii)* $\max(\mathcal{W}) < w$: In that case our algorithm will pull more tuples from the inner stream until we get a wid that is greater than the outer tuple's wid and then operate as in one of the previous cases. It should be noted that the joining algorithm on window identifiers is hybrid hash/merge-join since it takes advantage of a hash-index and the ordering of elements per hash-bucket.

2) $\texttt{WCacheL}_2$*:* The second level of indexing ensures the acceleration of data retrieval operations for attributes other than `Wid`. This index is nested on each window and we have adopted a KD-tree structure [5] for indexing in the rest of the dimensions that participate in a join between two streams. Each level of a KD-tree partitions the space into two subspaces. The partitioning is done along one dimension at the node at the top level of the tree, along another dimension in nodes at the next level, and so on, cycling through the dimensions. The partitioning proceeds in such a way that, at each node, approximately one-half of the points stored in the subtree fall on one side and one-half fall on the other. Partitioning stops when a node has less than a given

maximum number of points.

Example 3. *The right part of Figure 2 shows how a two level KD-tree partitions the $(Time, Measurement)$ space. The red line performs a data partitioning on the* `Time`*-axis, each partition containing* 6 *records. Then the blue lines perform data partitioning on the* `Measurement`*-axis, each partition containing exactly* 3 *records.*

B. Adaptive Stream Indexing

The *Adaptive Stream Indexing* technique is responsible for creating on the fly the appropriate $\texttt{WCacheL}_2$ structures that will accelerate execution of live-stream operations. This means that a KD-tree structure will only be created if the system's optimiser decides it beneficial for the query execution on the specific window of a stream. Formally, let's assume a set of stream-join operations that all have stream s as the inner relation of the join computation:

$$\bigcup_{i=1}^{\nu} \{s_i \bowtie_{\theta_i} s\}.$$

Moreover each join condition θ_i contains the conjunct $\texttt{Wid}_{s_i} = \texttt{Wid}_s$ Our problem constitutes in finding whether it is beneficial for the query execution speed to build a secondary level of KD-tree index on the attributes of s that appear in all θ_i conditions.

The *adaptive indexing* algorithm operates in two steps:

Step 1: With each new window w appearing in stream s, our algorithm first estimates the number of records that have a `Wid` of value w for all streams under consideration. The function $recs(t, w)$ that makes the estimation takes as input a stream t and the wid w. If all the records of stream t with a wid of w have already appeared, i.e. a record with a wid $w+1$ exists, our algorithm returns the actual number of records in window w. Otherwise, the number of records during the wth window is estimated based on what happened during the last n windows (where n has a default value of 10 but can be altered depending on the use case).

Step 2: The second step of the algorithm estimates whether it is beneficial to build a KD-tree index on the new window of stream s. If we assume that *(i)* the cost of computing the join operation between s_i and s on the wth window without any KD-tree index is denoted with $cost(s_i \bowtie_{\theta_i} s)$, *(ii)* the cost of performing the join operation on the wth window when having a KD-tree structure is denoted with $cost_{KD}(s_i \bowtie_{\theta_i} s)$, *(iii)* and the cost of building the actual KD-tree on the wth window of stream s is denoted with $cost_{KD}(s)$, then the algorithm decides that creating a KD-tree index is beneficial whenever:

$$\sum_{i=1}^{\nu} cost(s_i \bowtie_{\theta_i} s) > \sum_{i=1}^{\nu} cost_{KD}(s_i \bowtie_{\theta_i} s) + cost_{KD}(s).$$

Details on KD-trees and their corresponding cost functions can be found in [5].

C. Query Optimisations on Hybrid Operations

Regarding optimisations involving the fusion of static and streaming information, we have already presented in [6]: *(i)* the relational schema used to efficiently archive streaming information; *(ii)* the *materialised window signature* technique that permits precomputation of frequently requested aggregates on archived-stream measurements.

The idea underlying *Materialised Window Signatures* (MWS) is to facilitate preoccupation of frequently requested aggregates on each archived window of a stream. The latter helps accelerate analytical tasks that include hybrid operations over archived streams. These MWSs are stored in the backend and are later utilised while performing complex calculations between archived windows and a live stream. The summarisation values are determined by the analytics under consideration, e.g., for the computation of the Pearson correlation, we precompute the *avg* value and *standard deviation* on each archived window measurements.

In order to further expedite computation of similarity expressions that are common in the described scenario of diagnosing power-generating turbines, we utilize the *locality-sensitive hashing* (LSH) technique and the embedding of LSH information into MWSs. The premise of the LSH technique [7], [8] is that in many cases it is not necessary to insist on the exact answer; instead, determining an approximate answer with strong accuracy bounds should suffice. The above argument relies on the assumption that approximate similarity search can be performed much faster than the exact one. Detailed studies of applying LSH on live streams can be found in the literature [9], [10].

IV. Experimental Evaluation

The aim of our evaluation is to study how our optimisation techniques and query distribution to multiple workers accelerate the overall execution time of different analytic queries. We deployed our system to the Okeanos Cloud Infrastructure (www.okeanos.grnet.gr/) and used up to 128 virtual machines (VMs) each having a $2.100\,GHz$ processor with two cores and $4\,GB$ of main memory. We used streaming and static data with measurements produced by $100,000$ thermocouple sensors installed in 950 Siemens power generating turbines.

For the experimental evaluation, the following queries were adopted: *Query I*: The first query computes an equality join on the `Wid` and `Time` attributes between two live-streams. *Query II*: This query computes the Pearson correlation, above a threshold, of a live stream with a varying number of archived streams. *Queries III & IV*: These two queries are variations of Query II but, instead of the Pearson correlation, they compute similarity based on either the *average* or the *minimum* values within a window, e.g. $|\text{avg}(\vec{w}) - \text{avg}(\vec{v})| < 10^{\circ}C$. *Queries V*: This query calculates the Pearson correlation between two live

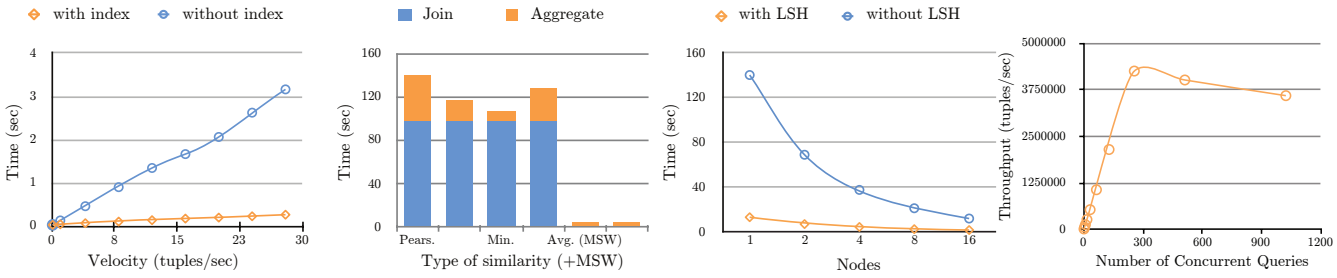

Figure 3: Effect of *(a)* adaptive indexing, *(b)* MWS optimisation, *(c)* intra-query parallelism and the LSH technique, *(d)* inter-query parallelism on live-streams

streams. We provide experimental evaluation for each of the techniques:

1) Adaptive Indexing Optimisation: We show how the adaptive indexing optimisation and the related indexing structures affect query-response times. We execute *Query I*: *(i)* on a single VM-worker; *(ii)* processing is performed on windows of 100 secs; *(iii)* the evaluation is performed on the live streams A and B (A being the inner relation of the join operation), building an index on stream A whenever appropriate ; *(iv)* stream A has a velocity of 10 $^{\text{tuples}}/_{\text{sec}}$, while we vary the velocity of stream B from 1 $^{\text{tuple}}/_{\text{sec}}$ to 28 $^{\text{tuples}}/_{\text{sec}}$. We measured the processing time for computing the join between a pair of windows of stream A and B with and without enabling the adaptive indexing technique that creates the necessary $\texttt{WCacheL}_2$ structures. In Fig. 3a, we observe that adaptive indexing helps accelerate the join computation and its benefits increase by increasing the velocity of the stream.

2) MWS Optimisation: We show how the MWS optimisation affects the query's response time. We executed test *Queries II*, *III*, and *IV*: *(i)* on a single VM-worker; *(ii)* for a fixed live-stream velocity of 1 $^{\text{tuple}}/_{\text{min}}$; *(iii)* for a fixed window size of 1 hour which corresponds to 60 tuples of measurements per window; *(iv)* and the current live stream window was measured against $100,000$ archived ones. In Fig. 3b we present the results of measuring the window processing time with and without the MWS optimization. The horizontal axis displays the three test queries with and without the MWS optimisation, while the vertical axis measures the time it takes to process 1 live-stream window against all the archived ones. This time is divided to the time it takes to join the live stream and the `Measurements` relation and the time it takes to perform the actual computations. Observe that the MWS optimisation reduces the time for the Pearson query only by 8.18%. This is attributed to the fact that the join operation between the live stream and the archived information takes 69.58% of the overall query execution time and cannot be avoided. For the other two queries, we not only reduce the CPU overhead of the query, but the optimiser further prunes this join from the query plan as it is no longer necessary. Thus, for these queries, the benefits of the MWS technique are substantial.

3) Parallelism between live & archived streams: For complex analytics such as the Pearson correlation that necessitates access to the archived windows, ExaStream permits us to accelerate queries by distributing the load among multiple worker nodes. We use the same setting as before for the Pearson computation without the MWS technique, but we vary this time the number of available workers from 1 to 16. In Fig. 3c, one can observe a significant decrease in the overall query execution time as the number of VM-workers increases. ExaStream distributes the archived-stream measurements between different worker nodes. Each node computes the Pearson coefficient between its subset of archived measurements and the live stream. As the number of archived windows is much greater than the number of available workers, intra-query parallelism results in significant decrease of the time required to perform the join operation.

4) Parallelism between live streams: This experiment focuses on the effect of accelerating live-stream operations by distributing the load to multiple worker nodes via inter-query parallelism. We executed *Query V* (Pearson correlation) *(i)* for a varying number of 1 to 1024 of concurrent queries between different pairs of live streams; *(ii)* for a fixed window size of 60 tuples; *(iii)* on non-overlapping windows; *(iv)* using 128 ExaStream worker nodes. We measured the window throughput, as the number of stream tuples that are processed per sec. Recall that each node is equipped with a two-core processor. We can see from Fig. 3d that initially, the overall throughput of the system increases linearly with the number of queries. This is because ExaStream utilizes the available workers and distributes the load evenly among them. When the number of queries reaches the number of cores available (256) we observe the maximum throughput of $4,250,226$ $^{tuples}/_{sec}$. From that point onward, the additional queries injected in Exareme result in multiple queries sharing the same core and, as a result, the cumulative throughput decreases.

5) LSH *Optimisation:* Our final experiment focuses on the LSH technique and how the intermix of MWSs, LSH buckets, and parallelism accelerates the computation of com-

plex similarity measures between live and archived streams. We perform the same experiment as in Section IV-3 for *parallelism between live & archived streams*, only this time we employ the LSH variation of MWSs. For the interested reader in the LSH parameterisation we used a combination of 7 AND-constructors and 6 OR-constructors. The results of this experiment are also displayed in Fig. 3 that compares performance with and without the optimisation. We observe a significant decrease in the overall query execution time when we adopt the combination of the MWS and LSH techniques. The price we have to pay for this increase in performance is 3% of false negative results for finding all Pearson correlations with an equality degree above 0.7.

V. Related Work

Query planning techniques for stream processing have already been proposed in the literature. For example, in [11] a query processing mechanism is proposed that continuously reorders operators in a query plan as it runs. In this context, we have introduced in-memory general purpose indexes for stream processing and their dynamic creation based on the adaptive indexing technique.

One of the leading edges in database management systems is to extend the relational model to support for continuous queries based on declarative languages analogous to SQL. Following this approach, systems such as TelegraphCQ [12], STREAM [13], and Aurora [14] take advantage of existing Database Management technologies, optimisations, and implementations. In the era of big data and cloud computing, a different class of DSMS has emerged. Systems such as Storm [15], Millwheel [16], and Apache Flink (https://flink.apache.org) offer an API that allows the user to submit data-flows of user defined operators. ExaStream combines characteristics of both approaches by allowing to describe in a declarative way complex data-flows of (possibly user-defined) operators.

VI. Conclusions

In this paper, we have studied various optimisations related to efficiently processing of streaming information, specifically: we have introduced novel indexing structures for stream processing and a query-planner component that decides when their creation is beneficial; we have suggested optimisations for efficient CPU and memory handling; we have proposed the appropriate structures to archive streaming information; and we have suggested some precomputed summarisations on the archived part of the stream, i.e. materialised window signatures. To put our ideas into practise, we have developed ExaStream, a data stream management system that is scalable, has declarative semantics, supports user defined functions, and allows for the stream and static data integration. Our work is accompanied by an empirical evaluation of our optimisation techniques.

References

[1] E. Kharlamov, S. Brandt, E. Jiménez-Ruiz, Y. Kotidis, S. Lamparter, T. Mailis, C. Neuenstadt, Ö. L. Özçep, C. Pinkel, C. Svingos, D. Zheleznyakov, I. Horrocks, Y. E. Ioannidis, and R. Möller, "Ontology-based integration of streaming and static relational data with optique," in *SIGMOD*, 2016.

[2] M. M. Tsangaris, G. Kakaletris, H. Kllapi, G. Papanikos, F. Pentaris, P. Polydoras, E. Sitaridi, V. Stoumpos, and Y. E. Ioannidis, "Dataflow processing and optimization on grid and cloud infrastructures." *IEEE Data Eng. Bull.*, 2009.

[3] H. Kllapi, P. Sakkos, A. Delis, D. Gunopulos, and Y. Ioannidis, "Elastic processing of analytical query workloads on iaas clouds," *arXiv preprint arXiv:1501.01070*, 2015.

[4] Ö. L. Özçep, R. Möller, and C. Neuenstadt, "A stream-temporal query language for ontology based data access," in *KI*, 2014.

[5] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, 1975.

[6] E. Kharlamov, Y. Kotidis, T. Mailis, C. Neuenstadt, C. Nikolaou, Ö. L. Özçep, C. Svingos, D. Zheleznyakov, S. Brandt, I. Horrocks, Y. E. Ioannidis, S. Lamparter, and R. Möller, "Towards analytics aware ontology based access to static and streaming data," in *ISWC*, 2016.

[7] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. ACM, 1998, pp. 604–613.

[8] A. Gionis, P. Indyk, R. Motwani *et al.*, "Similarity search in high dimensions via hashing," in *VLDB*, 1999.

[9] K. Georgoulas and Y. Kotidis, "Distributed similarity estimation using derived dimensions," *VLDB J.*, vol. 21, no. 1, pp. 25–50, 2012. [Online]. Available: http://dx.doi.org/10.1007/s00778-011-0233-y

[10] N. Giatrakos, Y. Kotidis, A. Deligiannakis, V. Vassalos, and Y. Theodoridis, "In-network approximate computation of outliers with quality guarantees," *Information Systems*, 2013.

[11] R. Avnur and J. M. Hellerstein, "Eddies: Continuously adaptive query processing," in *SIGMOD Record*, 2000.

[12] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah, "TelegraphCQ: Continuous Dataflow Processing," in *SIGMOD*, 2003.

[13] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom, "STREAM: the stanford stream data manager," in *SIGMOD*, 2003.

[14] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin *et al.*, "Aurora: A Data Stream Management System," in *SIGMOD*, 2003.

[15] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *SIGMOD*, 2014.

[16] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: fault-tolerant stream processing at internet scale," *VLDB Endowment*, 2013.