LARGE-SCALE ELASTIC ARCHITECTURE FOR DATA AS A SERVICE

Project Number:	FP7-ICT-318809
Project Title:	Large-Scale Elastic Architecture for Data as a Service
Deliverable Number:	D3.4
Title of Deliverable:	Enhanced LEADS real-time processing platform and tools
Contractual Date of Delivery:	M36 – 2015-09-30
Actual Date of Delivery:	2015-09-29

Abstract

D3.4 is the final deliverable of WP3. Its purpose is to describe the processing platform and tools delivered, focusing on the changes and enhancements that occurred within the last year of the project. The deliverable includes a brief overview of the query engine and processing platform, and a detailed discussion on the following key aspects: (a) the updated multi-cloud MapReduce model and implementation, (b) the changes in the query processing engine, and (c) research results on the topic of algorithms for distributed stream processing.

List of Contributors

Name	Organization	E-mail
Gaurav Singh	BM-Y!	gauravonline20@gmail.com
Ioanna Tsalouchidou	BM-Y!	ioanna@yahoo-inc.com
Janette Lehmann	BM-Y!	janettel@yahoo-inc.com
Necati Bora Edizel	BM-Y!	edizel@yahoo-inc.com
David Garcia Soriano	BM-Y!	davidgs@yahoo-inc.com
Aslay Cidgem	BM-Y!	aslayci@yahoo-inc.com
B. Barla Cambazoglu	BM-Y!	barla@yahoo-inc.com
Hossein Vahabi	BM-Y!	puya@yahoo-inc.com
Emmanuel Bernard	Red Hat	ebernard@redhat.com
Jonathan Halliday	Red Hat	jonathan.halliday@redhat.com
Mark Little	Red Hat	mlittle@redhat.com
Mircea Markus	Red Hat	mmarkus@redhat.com
Pedro Ruivo	Red Hat	pedro@infinispan.org
Aggelos Aggelidakis	TSI	aaggelidakis@softnet.tuc.gr
Eleftherios Chatzilaris	TSI	echatzilaris@softnet.tuc.gr
Antonios Deligiannakis	TSI	adeli@softnet.tuc.gr
Ioannis Demertzis	TSI	idemertzis@softnet.tuc.gr
Minos Garofalakis	TSI	minos@softnet.tuc.gr
Odysseas Papapetrou	TSI	papapetrou@softnet.tuc.gr
Evagelos Vazeos	TSI	vagvaz@softnet.tuc.gr
Christof Fetzer	TUD	christof.fetzer@tu-dresden.de
André Martin	TUD	andre.martin@tu-dresden.de
Do Le Quoc	TUD	do@se.inf.tu-dresden.de
Jons-Tobias Wamhoff	TUD	jons@inf.tu-dresden.de
Pascal Felber	UniNE	Pascal.Felber@unine.ch
Raluca Halalai	UniNE	Raluca.Halalai@unine.ch
Marcelo Pasin	UniNE	Marcelo.Pasin@unine.ch
Etienne Rivière	UniNE	Etienne.Riviere@unine.ch
Valerio Schiavoni	UniNE	Valerio.Schiavoni@unine.ch
Anita Sobe	UniNE	Anita.Sobe@unine.ch
Pierre Sutra	UniNE	Pierre.Sutra@unine.ch

Document Approval

	Name	Email	Date
Approved by WP Leader	Minos Garofalakis	minos@softnet.tuc.gr	2015-09-30
Approved by GA Member 1	Hossein Vahabi	puya@yahoo-inc.com	2015-09-30
Approved by GA Member 2	Etienne Rivière	etienne.riviere@unine.ch	2015-09-16

Contents

Executive summary

D3.4 is the final deliverable of WP3. It describes the delivered processing platform and tools, focusing on the changes and enhancements that occurred within the last year of the project. The deliverable includes a brief overview of the query engine and processing platform, and a detailed discussion on the following key aspects: (a) the updated multi-cloud MapReduce model and implementation, (b) the changes in the query processing engine, and (c) research results on the topic of distributed algorithms. Appendix A includes results of the research effort in the context of WP3 (accepted and submitted papers, work in preparation), and Appendix B includes instructions for deploying and using the platform.

Query processing engine. Section 3 presents the updated query processing engine, which introduces new features and substantial performance improvements compared to the one delivered at M24. The major changes include: (a) the support for indexes that enable faster query execution for selective queries, (b) an updated cost-based query planner, (c) handling of intermediary data in local auxiliary storage, (d) batch messaging and (e) more efficient implementations of the operators.

Multi-cloud MapReduce. Section 4 includes a discussion on the multi-cloud MapReduce engine and its improvement compared to the M24 engine.

Research results on distributed algorithms. In the last year, WP3 participants had several publications submitted and accepted in international conferences. In Section 5 we briefly summarize our research results from the last year (accepted papers, work in preparation and papers under submission). We also describe in more detail two recently accepted publications, and present their relation to LEADS. The first publication proposes a sketching technique, called ECM-sketch, and shows how it can be used in large-scale distributed systems, like LEADS, for maintaining sliding window summaries of distributed streams. The second proposes TOPiCo, an algorithm for maintaining top-k lists over distributed networks. The full papers are included in Appendix A.

Appendix. Appendix A includes copies of the papers discussed in Section 5. Appendix B includes detailed instructions on deploying and using the processing platform and query engine in new virtual machines.

1. Introduction

WP3 is responsible for offering the distributed data processing and querying capabilities of the LEADS platform. The foundations of the platform were laid down on M12, with the system architecture and a single-cloud implementation. At M24 we delivered a multi-cloud implementation of the platform, which enabled scaling up of MapReduce and of the query engine over multiple micro-clouds. In this deliverable, we describe an advanced distributed processing platform that overcomes scalability issues and performance bottlenecks of the M24 version, and improves stability and functionality.

The main advancements for the processing platform over the last year are as follows. First, we have substantially enhanced our **multi-cloud MapReduce execution engine**. This enhancement drastically improved stability of the engine and offers increased functionality. For example, code is now deployed automatically across the micro-clouds participating in the computation, and synchronization across phases occurs automatically. The new implementation also includes several optimizations which drastically improve performance. Since multi-cloud MapReduce is utilized extensively in the project, e.g., from the query execution engine, these performance improvements have a large impact on the project as a whole.

Second, we integrated several improvements on the **query engine**. These include: (a) the integration of partial-local indexes over distributed key value stores, (b) an auxiliary, more efficient, storage for quickly collecting intermediary query results, (c) a more advanced query planner to determine the optimal utilization of indexes, (d) a batch messaging platform to release network congestion, and (e) the redesign and implementation of the SQL operators to fully exploit the batch messaging platform, the indexes, and the auxiliary storage.

Third, we produced several **research results** related to the platform, some of which are also integrated in the platform. These include sketches for compact statistics maintenance from within the listeners (some of which are already utilized for determining query selectivity by the planner) and algorithms for distributed monitoring of top-k and other complex queries.

The document is structured as follows. **Section 2** briefly outlines the processing platform, discussing its interaction with the whole LEADS platform. **Section 3** provides design and implementation details on the query execution engine. **Section 4** describes the multi-cloud MapReduce programming model, and the involved implementation challenges. In **Section 5** we briefly summarize the research results of the last year related to WP3. **Section 6** concludes the document. **Appendix A** includes the relevant scientific publications (papers and journal articles). Finally, **Appendix B** includes detailed instructions for installing the platform in a separate installation.

2. Overview of the processing platform

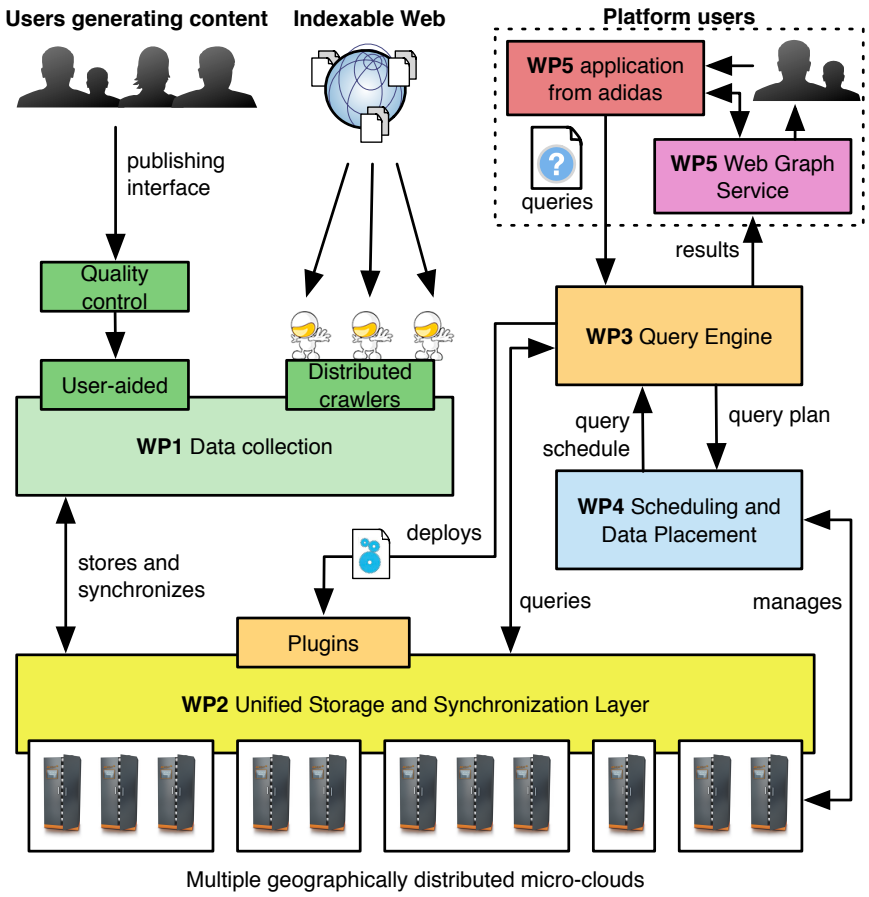

Figure 1 The real-time processing platform within LEADS

For completeness, in this section we present an overview of the real-time processing platform. The platform was originally designed at M12 (deliverable D3.1) and updated at M18 (deliverable D3.2) and M24 (deliverable D3.3). The platform is responsible for offering the distributed data processing and querying capabilities of the LEADS platform. It interacts with WP2 distributed storage engine, and with WP4 scheduler (cf. Figure 1). The core functional requirements for the platform are as follows:

1. **Querying tools**, for users to express their queries using both a declarative language for expert users and a graphical user interface for novice users.
2. **Novel programming models and tools** to enable the development of distributed data processing algorithms by implementing the MapReduce paradigm across multiple micro-clouds.
3. **Plugins** to support extending the data processing capabilities of the LEADS platform, to facilitate the development of distributed data mining algorithms using domain specific data processing.
4. **Privacy-preserving query capabilities** to enable users to store and query sensitive data safely.

The platform delivered at M36 satisfies all described functional requirements. In terms of querying tools, we have introduced a command-line interface and a graphical user interface based on Apatar

mashups[1]. Furthermore, the platform supports programmatic interaction for query execution from external applications through web services. These functionalities were described in more detail in D3.3 (M24).

Focusing on enabling developers to fully utilize the unique LEADS architecture, we developed and integrated a novel programming model that extends MapReduce to distributed micro-clouds. The model, which is first described in Section 4 of this deliverable, is a substantial extension over the M24 MapReduce engine. It has a low learning curve for existing MapReduce developers, and it is backward compatible with the M24 MapReduce model.

Arbitrary real-time data processing is supported by the use of plugins. Plugins were already introduced in M12 (deliverable D3.1), and made available over multiple micro-clouds in M24 (deliverable D3.3). Plugins support the core functionalities of the processing engine, including statistics maintenance for query execution, PageRank computation, and execution of data extraction pipelines required for end-user applications.

Finally, the platform supports privacy-preserving point queries over encrypted outsourced data. Privacy-preserving point querying was described already by M24 (deliverable D3.3). The functionality is available to the users via a web service.

In the remainder of this deliverable, we focus on the main achievements of WP3 in the last year. These include the novel MapReduce paradigm, the query processing engine, and some other research results, which are partially integrated in the LEADS platform.

3. The query processing engine

The querying requirements of LEADS users are supported by a distributed query execution engine that exploits the scalable storage capabilities of the LEADS storage layer (Ensemble) and the massive computing infrastructure enabled by the combination of micro-clouds. The engine supports data organization in a semi-relational format, i.e., it supports fixed-schema tables, indexes, and SQL queries, but in addition enables unbounded-size string attributes, which are critical in some contexts, e.g., storing of arbitrary web content. The query execution engine was operational since M12 in a single micro-cloud, and since M24 over multi-micro-cloud setups. Compared to M24 prototype, the M36 version improves drastically in terms of stability and performance by incorporating indexes, more advanced distributed implementations of the operators, a better query planner, and faster auxiliary local storage.

3.1. Major achievements since M24

Support for Indexes

Indexes have for long been utilized in relational databases to enable fast execution of queries with high selectivity. However, maintaining distributed indexes that can scale across micro-clouds is by no means straightforward. It can introduce unfavorable network bottlenecks during maintenance, since each tuple update may affect both the node storing the tuple and a remote node responsible for

[1] http://www.apatar.com

keeping the relevant portion of the index. Furthermore, distributed indexes are also more difficult to read efficiently, practically disabling parallelism. For example, the index segments holding all tuples that satisfy a range query predicate will be contiguous, and therefore will typically reside in a single node only. As such, adding more nodes in the network will not increase the degree of parallelism or performance of the index.

To alleviate these issues, in LEADS we maintain individual indexes at each node, containing only the data residing in the node. These indexes are created and maintained using Apache Lucene, and stored over a local Infinispan cache stored in the node. To avoid excessive I/O cost, indexes are built only for the attributes explicitly asked by the user, and not for all attributes that are present in the tuple. In terms of querying, all scan operators are broadcasted for local execution to all nodes. If the query contains a selection on an index attribute, the index is utilized to retrieve only the tuples from the index that satisfy the selection predicate, avoiding to parse the whole data set. The cost of querying each index is only $O(\log(N_i))$ per node i, where N_i denotes the size of the local index at the node. However, since the query is executed in parallel at all nodes, it will take at most $O(\max_i \log(N_i))$ time to locate and start retrieving the results from the index, i.e., the time required to search the largest of the distributed indexes.

Query Planner

To enable query planning, we extended Apache TAJO [Tajo], an open-source (Apache license) distributed query planner with LEADS-specific functionality. The major addition to the query planner since M24 was the utilization of cost-based optimization for selections. In particular, the query planner maintains distributed selectivity statistics on the indexed attributes of each table, and decides at query time whether the scan operator should load the whole data set at each node (sequential loading) or utilize the available index. Clearly, when the selectivity of the predicate is very high (i.e., very few tuples satisfy the predicate), it makes sense to use the index. On the other hand, if the predicate is not selective, it is more efficient to parse the whole data set and test the predicate on each tuple.

To maintain the necessary selectivity statistics, the planner utilizes the plugins architecture to install plugins on-the-fly at all nodes holding the index. The statistics are maintained using a distributed version of Count-Min sketch [CM05], stored over an in-memory Ensemble cache. Particularly, during the index creation, the query planner installs a plugin to each participant node in the Ensemble cache, which monitors all updates. All updates that cause a change in the Lucene index also update the distributed Count-Min sketch, i.e., collect the corresponding cells from the Ensemble cache, increase or decrease them accordingly, and commit them back to the Ensemble cache.

Notice that this continuous maintenance of the distributed Count-Min sketch may cause network congestion at periods of batch updates, due to the interaction with the Ensemble cache. To alleviate this issue, we introduced two optimizations. First, during the `CREATE INDEX` command, each node builds the local Count-Min sketch at batch, and writes it only once. This, essentially, makes the network cost of `CREATE INDEX` linear with the number of nodes, and not with the number of tuples/records in the table. Second, during the maintenance phase, the plugin also executes all updates in a batch mode, i.e., the updated Count-Min sketch is committed in the Ensemble cache every *w* updates, where w is a system parameter (by default set to 1000). This reduces the amortized cost per update, since the w updates typically touch overlapping cells from the Count-Min sketch. Clearly, batching on the plugin also means that the Count-Min sketch can be slightly outdated. However, this cannot lead to wrong results – it can only lead to a sub-optimal plan from the query

planner. In practice, as long as the window size w is kept in the order of thousands, the performance difference between the optimal and the sub-optimal plan is negligible.

Auxiliary local storage

As discussed, the query engine utilizes the Ensemble cache as a storage layer for the relational data. In our M24 prototype, the intermediary data that was produced by the different operators during query execution was also stored over the Ensemble cache. However, the Ensemble cache does not offer an efficient way of iterating over the stored tuples in a sorted order. Therefore, all SQL operators that required sorted access over these intermediary results (e.g., `GROUP BY`) could not be executed efficiently. Instead, `GET` commands on sequential keys were typically resulting to cache misses, causing continuous loading of large disk blocks, out of which a very small portion was used. Assuming a random order in the persistent storage, each `GET` would cause a disk I/O access with a probability equal to $1 - \frac{memSize}{diskSize}$, where $memSize$ denotes the number of tuples stored in the cache in-memory, and $diskSize$ the total number of tuples stored in the cache (also in persistent storage). Since main memory is typically several orders of magnitude smaller than secondary storage, $\frac{memSize}{diskSize}$ typically becomes very close to zero for Big Data management, making the number of disk I/O accesses linear to the number of tuples in the Ensemble cache.

To address this problem, we integrated LevelDB [LevelDB], a lightweight filesystem-based storage that supports storing and sorted loading of records in blocks. Specifically, all intermediary results were properly partitioned using the Ensemble partitioner, but instead of being written to the Ensemble cache, they were written to a LevelDB index (local at each node), which enabled storing the tuples sorted by a pre-selected attribute. This, in turn, provided a very efficient sorted iterator, since reading a single disk block would bring in the disk cache many records, and in the correct order. As a consequence, the number of page faults and disk I/O accesses was drastically reduced to be equal to the number of total blocks in LevelDB. Unfortunately, inclusion of LevelDB introduced a new bottleneck, for storing the records in the correct order in the LevelDB index. To alleviate this bottleneck, intermediary results were loaded using batch loading. Batch loading enables pre-sorting of a large number of new tuples in main memory, before accessing the file system to make the changes permanent, thereby reducing the number of blocks that need to be read and written to the filesystem.

Batch messaging

As in any distributed data-intensive platform, network congestion constitutes the major bottleneck in the query engine. In particular, some SQL operators cause massive data copying over the network (i.e., remote `PUTs` over an Ensemble cache) due to their requirement that all tuples/records with the same key must reside in the same node for the operator to be correct. For example, executing a foreign-key `join` on two relations $R1$ and $R2$ requires sending over the network each tuple of $R1$ with a probability of $pr = (N(R1) - 1)/N(R1)$, where $N(R1)$ denotes the number of nodes storing $R1$. Similarly, for $R2$, this probability is $pr = \frac{N(R2)-1}{N(R2)}$. In large networks, pr will approximate 1, leading to an extremely large number of (typically small) messages over the network (one per tuple). Bloom joins [Mul90, MNP+07, RPS08], broadcast joins [SBS+13], and other similar techniques proposed in the context of distributed databases still do not alleviate the problem for the case of foreign-key joins, since all tuples typically participate in the join.

To alleviate this problem, we implemented and integrated in LEADS a batch messaging system over the LEADS storage layer and Ensemble. The system focuses on optimizing the cross-cloud messages, which are typically several orders of magnitude slower than the messages within the same micro-

cloud. The batching system is heavily utilized by the query execution engine as well as the MapReduce engine. Particularly, each node creates a message queue of bounded size *s* for each micro-cloud, and uses it to temporarily store all outgoing messages to this micro-cloud. Whenever a queue becomes full or after a timeout, the node packs together all messages in the queue, compresses them using an efficient compression algorithm (Snappy[2]), and sends them to the corresponding micro-cloud as a single message. This is achieved via a remote `PUT` on a specially constructed cache. A local listener installed on this cache reads and decompresses the message, and forwards it to the corresponding nodes inside the micro-cloud.

Notice that batch messaging also introduced the need for additional synchronization between the nodes. That is, batch `PUTs` needed to be considered as an asynchronous operator, since the individual `PUTs` resulting by any batch `PUT` are executed after the batch `PUT` function returns. This issue is handled by the messaging system transparently, such that any developer using the batch messaging system (e.g., for MapReduce applications) can focus explicitly on developing the application logic and not on the intrinsic behavior of batch messaging.

Reimplementation of the SQL operators

Most SQL operators were re-implemented to utilize the indexes and the architectural modifications made since the M24 prototype, including multi-cloud MapReduce, auxiliary intermediary storage, and batch messaging. We have also integrated in the query engine the standard Bloom Join algorithm [Mul90], which enables substantial network savings in the presence of high-selectivity filters at any of the relations. The new implementations improve the networking and computational performance, and reduce the memory requirements of the query engine.

3.2. Experiments

We evaluated the performance of the query execution engine on different configurations using the AMPLab benchmark.[3] The AMPLab benchmark contains both a configurable data generation tool and a query generator, and is frequently used for big data systems evaluation.

3.2.1. Query generation and system configuration

To evaluate our engine we used two types of queries: (a) the AMPLab queries that are generated by the benchmark scripts, and (b) a set of additional – synthetic -- queries on the same data set.

AMPLab scripts create four different query types. Queries 1-3 are standard SQL queries, whereas Query 4 involves user-defined functions implemented as external scripts, which is not supported by our query engine. We therefore focus on the first three queries. All AMPLab queries have a very high selectivity, i.e., they return several hundreds of thousands of results, sometimes even several millions of results, or they limit the size of the results at the last step. These types of queries are useful for offline data mining and data analytics, e.g., run a clustering algorithm on a subset of the crawled web pages.

In addition, we devised a set of 6 additional queries that are more selective than the previous (i.e., return a few hundreds of results). These selective queries are the ones that a human would be interested to execute, e.g., via an API, the command-line interface, or via our Apatar-based GUI. Notice that these queries are the ones that will typically utilize indexing functionalities and clever

[2] http://google.github.io/snappy/
[3] Available at https://amplab.cs.berkeley.edu/benchmark/

distributed processing algorithms, to avoid loading everything from disk or shipping too much data over the network. Table 1 presents both sets of queries.

Table 1 Evaluation Queries

Query id	Query
	AMPLab queries
Q1	`SELECT pageURL, pageRank FROM rankings WHERE pageRank > X` for X=1000 (**Q1a**), 100 (**Q1b**)
Q2	`SELECT SUBSTR(sourceIP, 1, X), SUM(adRevenue) FROM uservisits GROUP BY SUBSTR(sourceIP, 1, X)` for X=8 (**Q2a**), 10 (**Q2b**)
Q3	`SELECT sourceIP, totalRevenue, avgPageRank FROM (SELECT sourceIP, AVG(pageRank) as avgPageRank, SUM(adRevenue) as totalRevenue FROM rankings AS R, uservisits AS UV WHERE R.pageURL = UV.destURL AND UV.visitDate BETWEEN Date(`1980-01-01') AND Date(`X') GROUP BY UV.sourceIP) ORDER BY totalRevenue DESC LIMIT 1` for X='1980-04-01' (**Q3a**), '1983-01-01' (**Q3b**)
	Additional queries
Q4	`SELECT pageURL, pageRank FROM rankings WHERE pageRank = X` (executed with random X values which are contained in the data set)
Q5	`SELECT sourceIP FROM uservisits WHERE visitDate = Date(X)` (executed with random dates X which are contained in the data set)
Q6	`SELECT pageURL, pageRank FROM rankings WHERE pageRank >= X AND pageRank <= Y` (executed with random X and Y values which are contained in the data set, with Y-X=5)
Q7	`SELECT sourceIP FROM uservisits WHERE visitDate BETWEEN Date(X) AND Date(Y)` (executed with random dates X and Y which are contained in the data set, and are set such that the date predicate covers two days)
Q8	`SELECT sourceIP, totalRevenue, avgPageRank FROM (SELECT sourceIP, AVG(pageRank) as avgPageRank, SUM(adRevenue) as totalRevenue FROM rankings AS R, uservisits AS UV WHERE R.pageURL = UV.destURL AND UV.visitDate BETWEEN Date(X) AND Date(Y) GROUP BY UV.sourceIP) ORDER BY totalRevenue DESC LIMIT 1` (executed with random dates X and Y which are contained in the data set, and are set such that the date predicate covers two days)
Q9	`SELECT SUBSTR(sourceIP, 1, 10), SUM(adRevenue) FROM uservisits WHERE visitDate BETWEEN Date(X) AND Date(Y) GROUP BY SUBSTR(sourceIP, 1, 10)` (executed with random dates X and Y which are contained in the data set, and are set such that the date predicate covers two days)

Considered configurations. The considered experimental setups are presented in Table 2. Our strategy on choosing the configurations was to evaluate how the query engine scales with the data volume, and with the size of the network (number of nodes per micro-cloud, for two micro-clouds). All reported experiments were executed on resources provided by project partner Cloud&Heat (micro-clouds dd1a and dd2c). The used computing resources were shared, i.e., the real machines were hosting additional virtual machines (not related to LEADS). The virtual machines were configured with 4 cores and 8 GB RAM. We used the default configuration provided by the Bootstrapper (cf. Appendix B.3), which deploys one system component at each virtual machine. Infinispan Ensemble was configured to keep 1024 entries in main memory per node, and with 128 MB in-memory cache. Also, LevelDB auxiliary storage was configured to use a RAM buffer index of maximum size 32 MB.

In the following, to avoid repetition, we will be reporting only the configuration values that deviate from the default values of Table 2.

Table 2 Considered configurations

Setting	Possible values (default values are emphasized)
#tuples	4 Million per table, **16 Million per table**, All data (18 Million in Rankings, 155 Million in userVisits)
#nodes per micro-cloud	**4**, 6, 9

3.2.2. Varying the data set size

We first examine the effect of the data set size, by experiments with 4 and 16 Million tuples per table. Table 3 presents the time taken for answering the AMPLab queries in a deployment of 2 micro-clouds with 4 machines each. Our first observation is that execution time for the AMPLab queries increases almost linearly with the data set size. The small deviation on this linear relation is attributed to CPU spikes due to variances in the performance of the virtual machines, e.g., due to network load or workload caused by other virtual machines that were hosted at the same physical machine. This linear scaling is normal for these types of queries, since traditional centralized database optimizations cannot be utilized for these queries, due to the large volume of data that satisfies the predicates. For example, indexes cannot help; even though indexes can be used for quickly retrieving all tuple ids that satisfy the predicate, the actual tuples still need to be retrieved from the distributed storage infrastructure (from an Ensemble cache), a time-consuming task when there are many tuples in the intermediary result set. Therefore, the planner does not utilize indexes on these queries. In fact, since sequential distributed GETs are substantially slower than batch loading and iteration over all tuples (due to the distributed deployment of the Ensemble cache), the execution planner utilizes indexes only for fairly selective queries. Also, note that the size of the result also affects execution time, since more results need to be shipped over the participating network. For example, Q1a requires less time than Q1b, since the former is more selective than the latter (the results of Q1a are a subset of the results of Q1b).

In Table 4 we present the required execution time for the 6 additional queries. For comparison purposes, we present time with and without indexes. Interestingly, execution time with indexes grows sub-linearly with the number of stored tuples. This happens because the planner incorporates the indexes for answering these queries, which enable efficient (logarithmic-cost) retrieval of the tuple ids that belong in the results without parsing the whole data set. Since the number of results that need to be retrieved for these queries is relatively small (in the order of hundreds or a few thousands), distributed GETs over the Ensemble cache do not pose performance bottleneck.

Table 3 AMPLab queries

Tuples per table	Execution time (sec.)					
	Q1a	**Q2a**	**Q3a**	**Q1b**	**Q2b**	**Q3b**
4 M	8	177	28	11	199	61
16 M	22	687	95	30	738	257

Table 4 Additional queries

Tuples per table	Execution time (sec.)					
	Q4	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**
4 M	WO/Index:8	11	8	12	25	13
	W/Index:1	3	1	3	7	5
16 M	WO/Index:21	41	21	40	67	45
	W/Index:1	3	1	3	7	5

Table 5 Experiments with the full AMPLab data set
(18 Million in rankings, 155 Million in uservisits, 2 micro-clouds, 9 nodes each)

Presence of indexes	Execution time (sec.)					
	Q1a	**Q2a**	**Q3a**	**Q1b**	**Q2b**	**Q3b**
No (index not utilized)	34	4130	511	30	4280	2006
	Q4	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**
No	34	365	35	366	409	384
Yes	2	37	2	61	51	47

Table 5 presents the required time for executing the same queries over the full AMPLab data set (table uservisits with 155 Million tuples, and table rankings with 18 Million tuples). To achieve better performance, in this experiment we have used all the available free virtual machines in the two micro-clouds (9 VMs per micro-cloud). As expected, the queries require more time due to the increased data volume (overall an order of magnitude more tuples in table uservisits compared to the 16M data set). Nevertheless, utilization of indexes drastically cuts down on the IO costs, as evident, e.g., in queries 4 and 6, which are plain selections with filters. Queries Q5, Q7, Q8, Q9 also benefit from the index in terms of scanning the data, but the required time in these queries is dominated by the network time.

3.2.3. Varying the number of nodes per micro-cloud

This set of experiments was designed to test the availability of the system to scale out, i.e., incorporate the capacity of more nodes to increase system performance. Therefore, all queries described at Section 3.2.1 were executed in two different settings: (a) with 4 nodes per micro-cloud, and (b) with 6 nodes per micro-cloud.

Table 6 and Table 7 present the execution time for all queries. We see that querying performance improves by adding more nodes, which is a desired property for big data systems. The scale-up ratio is almost linear for the AMPLab queries, i.e., querying time is proportional to the number of machines. This happens because, as stressed earlier, these queries disable traditional relational databases optimization strategies, such as indexes.

Interestingly, the number of nodes per micro-cloud does not have a substantial performance impact for queries Q4 - Q9, when using an index. For instance, Q4 requires 1 second in both deployments. This happens because the generated query plans of Q4 – Q9 utilize indexes, which already improves performance by reducing the tuples that need to be loaded from disk (recall that indexes offer logarithmic-cost lookups). As such, adding more nodes has a negligible improvement on the already-fast loading time (logarithmic cost), but cannot help with the time spent on network operations, e.g., for shipping intermediary results across nodes and micro-clouds, which becomes the dominant cost. Nevertheless, the ability of the platform to run on larger infrastructure is pivotal for the scalability of the system with more data.

Summarizing, our experiments have shown that the query engine scales with the data set size. Scalability is achieved either by utilizing indexes, or by adding more computational capacity, e.g., more nodes, or more micro-clouds.

Table 6 AMPLab queries

Nodes per micro-cloud	Execution time (sec.)					
	Q1a	**Q2a**	**Q3a**	**Q1b**	**Q2b**	**Q3b**
4	22	687	95	30	738	257
6	15	465	63	21	510	185

Table 7 Additional queries

Nodes per micro-cloud	Execution time (sec.)					
	Q4	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**
4	WO/Index:21	41	21	40	67	45
	W/Index:1	4	1	4	9	5
6	WO/Index:16	25	15	27	47	30
	W/Index:1	5	1	7	9	6

4. MapReduce for multiple micro-clouds

A core focus of WP3 was to enable efficient arbitrary data processing over the distributed micro-cloud resources. Even though there exist several industrial-strength big data platforms for scalability over individual clusters, these platforms cannot be deployed over distributed cloud resources. Towards this direction, in WP3 we implemented a multi-cloud MapReduce engine supporting the traditional MapReduce programming paradigm (M24), and a new programming model that is more suitable for multi-cloud architectures (M36). We now briefly overview the progress over the M24 engine.

The main functional limitation of the M24 engine was that it did not support automatic deployment of arbitrary MapReduce code; manual work was still required for compiling and uploading the bytecode to the participating nodes. Automatic deployment is supported in the M36 prototype, i.e., the developer provides the bytecode and required libraries at a single node, and everything is deployed and configured automatically at all nodes holding portions of the cache.

In terms of non-functional requirements, performance of the M36 implementation is improved compared to the M24 platform, by utilizing batch messaging and auxiliary local storage (cf. also Section 3.1). The effect of batch messaging in multi-cloud MapReduce is substantial, since mappers generally create a vast number of messages (one per tuple), most of which need to be transferred

over the Internet to different micro-clouds. By batching these messages to a single compressed message we reduce network congestion and increase overall processing throughput.

Auxiliary storage is also used in MapReduce, for storing the output of the Mappers in the nodes that will execute the Reducers. In particular, Mappers produce key-value pairs, which are partitioned to the Reducers based on their keys. The LEADS storage layer and Ensemble cache in particular, however, does not yet support efficient sorted iteration over all keys, which is necessary for passing all tuples with the same key to the same Reducer. Instead, for each distinct key we need to execute subsequent `GETs` from the Ensemble cache, each of which causes a disk I/O access with high probability. To avoid this issue, the intermediary results from all Mappers were saved to a LevelDB index. Even though the index is stored in secondary memory (its performance is bounded by the disk controller's performance), it enables block-based data loading. As such, many subsequent tuples are loaded at each single disk read operation, and all tuples with the same key are passed to the same Reducer to execute the Reduce phase. This improves reading performance substantially and reduces disk I/O load.

5. Additional research results

In addition to the described progress on the query engine and the multi-cloud MapReduce paradigm, in the last year we conducted research activities in the area of large-scale distributed systems that do not belong in the core of the LEADS real-time processing platform, but directly benefit it by adding supportive functionality. In detail, the two major research topics we contributed to were:

A. Distributed sketching
B. Distributed top-k monitoring

Distributed sketching is integrated into the platform. We decided not to integrate distributed top-k monitoring into the platform due to manpower and time budgets. For matters of completeness we now briefly summarize the accepted publications in these topics, and explain how these are related to LEADS. The full publications can be found in Appendix A.

5.1. Distributed sketching

Focusing on distributed data stream processing, in [PGD15] we considered the problem of complex query answering over distributed, high-dimensional data streams in the sliding-window model. We introduced a novel sketching technique (termed ECM-sketch) that allows effective summarization of streaming data over both time-based and count-based sliding windows with probabilistic accuracy guarantees. ECM-sketch enables point as well as inner-product queries, and can be employed to address a broad range of problems, such as maintaining frequency statistics, finding heavy hitters, and computing quantiles in the sliding-window model.

The core of the sketch is a modified Count-Min sketch [CM05]. Count-Min sketches alone cannot handle the sliding window requirement. To address this limitation, ECM-sketches replace the Count-Min counters with sliding window structures, e.g., exponential histograms [DGI+02], or waves [GT02]. Each counter is maintained as a sliding window, covering the last N time units, or the last N arrivals, depending on whether we need time-based or count-based sliding windows.

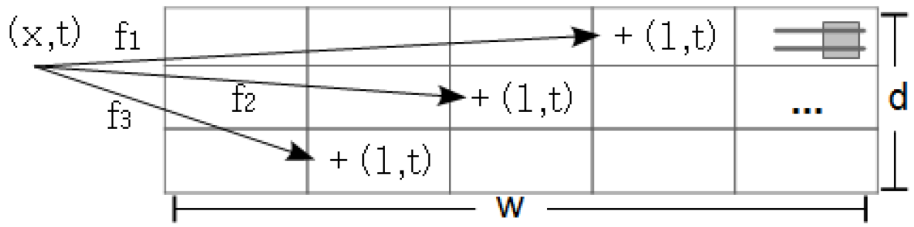

Figure 2 Adding an element to the ECM-sketch.

Adding an item e to the structure is similar to the case of the standard Count-Min sketches. The process for time-based sliding windows is depicted in Figure 2. First, the counters in position $(h_i(e), i)$, where i $\in$ {1 . . . d}, corresponding to the d hash functions are detected. For each of the counters, we register the arrival of the item at time t, and remove all expired information, i.e., the buckets of the exponential histogram that have no overlap with the sliding window range. The process for count-based sliding windows is similar, but instead of registering each arrival with system time t, we register it with the count of arrivals since the beginning of the stream.

Query execution. A point query (x, r) is a combination of an item identifier x, and the query range r defined either as the number of time units or the number of arrivals. Point queries are executed as follows. The query item is hashed to the d counters at positions $(h_i(x), i)$, where i $\in$ {1 . . . d}, and the estimate of each counter $E(h_i(x), i, r)$ for the query range is computed. The estimate value for the frequency of x in r is $\min_i E(h_i(x), i, r)$. Similar expressions exist for estimating quantiles and L2 norms.

Distributed sketching and LEADS. ECM-sketches support continuous queries over distributed streams. In particular, streams are partitioned to the available nodes, and monitored in real-time. Each node updates a local instance of the sketch, and runs all registered continuous queries (e.g., frequency, top-k, L2 norm queries) over this local instance, by converting the queries to threshold crossing queries and exploiting the geometric method for distributed monitoring [SSK04]. The local sketches are sent to the coordinator only when an update causes a threshold crossing of the geometric method.

ECM-sketches are highly related to LEADS, since LEADS handles fast data streams over a distributed network. As such, we have integrated ECM-sketches in LEADS as a library. Since ECM-sketches are focused on stream processing, this library is made accessible from within the plugins architecture of LEADS, which supports stream monitoring.

5.2. Distributed Top-k monitoring

Focusing on large-scale stream monitoring, in [SRS+15] we tackle the more specific problem of detecting the most-frequent (top-k) items from high-rate event streams. The collection of aggregates and statistical metrics over online data streams has attracted considerable attention from both academia and industry over the past decade. Mining the properties of such data streams can be used in various contexts, ranging from targeted advertisement [CJE14] to automated virus detection [SEV+04].

We propose TOPiCo, a protocol that computes the most popular events across geo-distributed sites in a low cost, bandwidth-efficient and timely manner. TOPiCo starts by building the set of most

popular events locally at each site. Then, it disseminates only events that have a chance to be among the most popular ones across all sites, significantly reducing the required bandwidth.

We consider the following motivating scenario. Several geo-distributed micro-clouds produce events, and each of them maintains the top-k frequent items over a sliding window, allowing observing the trends for a particular region. Simultaneously, we are also interested in computing the most frequent items at the scale of the geo-distributed infrastructure formed by all micro-clouds. We are thus interested in maintaining the *global* top-k frequent items in a multi-site information system. The global top-k frequent items set is referred as G in the remainder of this summary. We assume that each site maintains, using an existing single-site algorithm, the *local* top-k frequent items for its own stream.

In a nutshell, at each site TOPiCo executes the following two tasks: (Update) Site *i* computes locally a list of candidates items that it broadcasts together with their local number of occurrences to all sites. (Disseminate) Upon receiving a list of candidates from some distant site *j*, a site *i* updates its global view of the most frequent items G_i. To that end, *i* first sums-up for each item the contribution received in the candidate lists from the other sites (such contribution equals 0, if the item was not received). Then, site *i* sorts the global contributions and outputs G_i. TOPiCo constructs a list of candidates by determining at each site a value $l \geq k$ for which the top-l ranked items in L_i have a chance to enter in G. Such an estimation is based on (i) the global number of occurrences of each item among the top-k in G_i, (ii) the number of occurrences of each items in L_i, and (iii) the candidates received by remote sites.

Further details, such as a proof of correctness and an extensive evaluation on real-world data traces can be found in [SRS+15].

TOPiCo and LEADS. The TOPiCo proposal is particularly relevant in the context of LEADS. We consider the streams originating from the web-crawlers operating at the micro-cloud level. Periodically, i.e. after each crawling round, locally at each site a crawler needs to decide which are the most relevant sites to crawl, a process that typically involves the execution of a PageRank algorithm. The TOPiCo algorithm can be easily integrated in the LEADS plugin architecture, and can be used to drive an alternative prioritization strategy for the crawling process.

5.3. Work in progress

In addition to the accepted papers, WP3 participants are currently preparing several new submissions, to disseminate the project results:

- Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, Minos Garofalakis: Practical Private Range Search Revisited (under revision for the ACM SIGMOD'2016 Conference).
  Relation to the project: The paper proposes a privacy-preserving approach for range queries on outsourced data. It is conducted in the context of Task 3.4, and parts of it were integrated in the project already at the M24 milestone.
- Odysseas Papapetrou, Minos Garofalakis: Monitoring Distributed Fragmented Skylines (under revision for IEEE Transactions on Knowledge and Data Engineering).
  Relation to the project: The article proposes a distributed algorithm for monitoring fragmented skylines that can scale over wide area networks with limited bandwidth (e.g., Internet, as opposed to Ethernet/LAN). The LEADS multi-cloud platform is an ideal

deployment infrastructure for this algorithm, due to the (possibly) weak Internet connection across micro-clouds.

- Ioakim Perros, Nikolaos Pavlakis, Odysseas Papapetrou, Minos Garofalakis: Distributed Stochastic Pagerank Maintenance over Streaming Graphs (in preparation).
  Relation to the project: Although there have been earlier works on streaming PageRank maintenance, none of the works enabled the participating nodes to be widely distributed, e.g., across micro-clouds. This is an inherent requirement in the context of LEADS. In this paper we propose a stochastic PageRank maintenance algorithm that drastically reduces network requirements between micro-clouds. The algorithm is one of the core tools offered in WP3 as a plugin.

6. Conclusion

This deliverable described the final LEADS real-time processing platform developed in the context of WP3. Our discussion focused on the advancement in the MapReduce execution engine and in the query engine since M24. In terms of MapReduce, we proposed a novel programming paradigm (first described in this deliverable), which enables developers to introduce a partial reduce phase inside each micro-cloud, thereby drastically reducing data transfer across micro-clouds. In terms of the query engine, we presented several advancements over the M24 engine, with performance improvements by several orders of magnitude for some queries (mostly the highly selective queries, but also simple iceberg queries). Among the main improvements, we can mention indexes, a cost-based planner, and a local auxiliary storage at each node to improve computational and I/O performance, and a batch messaging system for optimizing network resources. The deliverable also includes summary of the research results delivered by WP3 partners in the last year. Furthermore, detailed deployment instructions for the whole data processing and query engine are included in Appendix B.

7. References

[Tajo] Tajo - A Big Data Warehouse System on Hadoop, available online at http://tajo.apache.org. Retrieved 2/9/2015.

[CM05] Graham Cormode, S. Muthukrishnan: An improved data stream summary: the count-min sketch and its applications. J. Algorithms 55(1): 58-75 (2005).

[ACZ+13] Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff M. Phillips, Zhewei Wei, Ke Yi: Mergeable summaries. ACM Trans. Database Syst. 38(4): 26 (2013).

[ZMH09] Weizhong Zhao, Huifang Ma, Qing He: Parallel K-Means Clustering Based on MapReduce. CloudCom 2009: 674-679.

[LevelDB] LevelDB.org, available online at http://leveldb.org. Retrieved 2/9/2015

[PGD15] Odysseas Papapetrou, Minos N. Garofalakis, Antonios Deligiannakis: Sketching distributed sliding-window data streams. VLDB J. 24(3): 345-368 (2015).

[DGI+02] Mayur Datar, Aristides Gionis, Piotr Indyk, Rajeev Motwani: Maintaining Stream Statistics over Sliding Windows. SIAM J. Comput. 31(6): 1794-1813 (2002).

[GT02] Phillip B. Gibbons, Srikanta Tirthapura: Distributed streams algorithms for sliding windows. SPAA 2002: 63-72.

[SSK07] Izchak Sharfman, Assaf Schuster, Daniel Keren: A geometric approach to monitoring threshold functions over distributed data streams. ACM Trans. Database Syst. 32(4) (2007).

[SRS+15] Valerio Schiavoni, Etienne Rivière, Pierre Sutra, Pascal Felber, Miguel Matos, Rui Oliveira: TOPiCo: Detecting most frequent items from multiple high-rate event streams. DEBS 2015: 58-67.

[CJE14] Culhane, W., Jayaram, K. R., and Eugster, P. Fast, expressive top-k matching. In Proceedings of the 15th International Middleware Conference (New York, NY, USA, 2014), Middleware '14, ACM, pp. 73–84.

[SEV+04] Singh, S., Estan, C., Varghese, G., and Savage, S. Automated worm fingerprinting. In Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6 (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 45–60.

[SBS+13] Sebastian Schelter, Christoph Boden, Martin Schenck, Alexander Alexandrov, and Volker Markl. Distributed matrix factorization with mapreduce using a series of broadcast-joins. In Proceedings of the 7th ACM conference on Recommender systems (RecSys '13). ACM, pp. 281-284.

[MNP+07] Loizos Michael, Wolfgang Nejdl, Odysseas Papapetrou, Wolf Siberski.
Improving distributed join efficiency with extended bloom filter operations. AINA 2007: 187-194.

[Mul90] Mullin, James K., "Optimal semijoins for distributed database systems," in IEEE Transactions on Software Engineering, vol.16, no.5, pp.558-560, May 1990

[RPS08] Sukriti Ramesh, Odysseas Papapetrou, Wolf Siberski: Optimizing Distributed Joins with Bloom Filters. ICDCIT 2008: 145-156

Appendix A. Publications

The following two publications accepted within 2015 are relevant to WP3 and were supported by LEADS.

- Odysseas Papapetrou, Minos N. Garofalakis, Antonios Deligiannakis: Sketching distributed sliding-window data streams. VLDB J. 24(3): 345-368 (2015).
- Valerio Schiavoni, Etienne Rivière, Pierre Sutra, Pascal Felber, Miguel Matos, Rui Oliveira:TOPiCo: Detecting most frequent items from multiple high-rate event streams. DEBS 2015: 58-67.

Noname manuscript No.
(will be inserted by the editor)

Sketching Distributed Sliding-Window Data Streams

Odysseas Papapetrou · Minos Garofalakis · Antonios Deligiannakis

the date of receipt and acceptance should be inserted later

Abstract While traditional data-management systems focus on evaluating single, ad-hoc queries over static data sets in a centralized setting, several emerging applications require (possibly, continuous) answers to queries on *dynamic* data that is widely *distributed* and constantly updated. Furthermore, such query answers often need to discount data that is "stale", and operate solely on a *sliding window* of recent data arrivals (e.g., data updates occurring over the last 24 hours). Such *distributed data streaming* applications mandate novel algorithmic solutions that are both time- and space-efficient (to manage high-speed data streams), and also communication-efficient (to deal with physical data distribution). In this paper, we consider the problem of complex query answering over distributed, high-dimensional data streams in the sliding-window model. We introduce a novel sketching technique (termed *ECM-sketch*) that allows effective summarization of streaming data over both time-based and count-based sliding windows with probabilistic accuracy guarantees. Our sketch structure enables point as well as inner-product queries, and can be employed to address a broad range of problems, such as maintaining frequency statistics, finding heavy hitters, and computing quantiles in the sliding-window model. Focusing on distributed environments, we demonstrate how ECM-sketches of individual, local streams can be composed to generate a (low-error) ECM-sketch summary of the order-preserving merging of all streams; furthermore, we show how ECM-sketches can be exploited for continuous monitoring of sliding-window queries over distributed streams. Our extensive experimental study with two real-life data sets validates our theoretical claims and verifies the effectiveness of our techniques. To the best of our knowledge, ours is the first work to address efficient, guaranteed-error complex query answering over distributed data streams in the sliding-window model.

The final publication is available at Springer via http://dx.doi.org/10.1007/s00778-015-0380-7.

O. Papapetrou
Technical University of Crete
E-mail: papapetrou@softnet.tuc.gr

M. Garofalakis
Technical University of Crete
E-mail: minos@softnet.tuc.gr

A. Deligiannakis
Technical University of Crete
E-mail: adeli@softnet.tuc.gr

1 Introduction

The ability to process, in real time, continuous high-volume *streams* of data is a common requirement in many emerging application environments. Examples of such applications include, sensor networks, financial data trackers, and intrusion-detection systems. As a result, in recent years, we have seen a flurry of activity in the area of *data-stream processing*. Unlike conventional database query processing that requires several passes over a static, archived data image, data-stream processing algorithms often rely on building concise, approximate (yet, accurate) *sketch synopses* of the input streams in real time (i.e., in one pass over the streaming data). Such sketch structures typically require *small space and update time* (both significantly sublinear in the size of the data), and can be used to provide *approximate query answers* with guarantees on the quality of the approximation. These answers can be more than sufficient for typical exploratory analysis of massive data, where the goal is to detect interesting statistical behavior and patterns rather than obtain answers that are precise to the last decimal. Large-scale stream processing applications are also inherently *distributed*, with several remote sites observing their local stream(s) and exchanging information through a communication network. This distribution of the data naturally imposes critical

communication-efficiency requirements that prohibit naïve solutions that centralize all the data, due to its massive volume and/or the high cost of communication (e.g., in sensornets). Communication efficiency is particularly important for *distributed event-monitoring* scenarios (e.g., monitoring sensor or IP networks), where the goal is real-time tracking of distributed measurements and events, rather than one-shot answers to sporadic queries [33].

Several query models for streaming data have been explored over the past decade. Streaming data items naturally carry a notion of "time", and, in many applications, it is important to be able to downgrade the importance (or, weight) of older items; for instance, in the statistical analysis of trends or patterns in financial data streams, data that is more than a few months old might be considered "stale" and irrelevant. Various *time-decay models* for querying streaming data have been proposed in the literature, mostly differentiating on the relation of an item's weight to its age (e.g., exponential or polynomial decay [7]). The *sliding-window* model [16] is one of the most prominent and intuitive time-decay models that considers only a window of the most recent items seen in the stream thus far (i.e., items outside the window are "aged out" or given a weight of zero). The window itself can be either *time-based* (i.e., items seen in the last N time units) or *count-based* (i.e., the last N items). Several algorithms have been proposed for maintaining different types of statistics over sliding-window data streams while requiring time and space that is significantly sublinear (typically, poly-logarithmic) in the window size [16,21,32,34]. Still, the bulk of existing work on the sliding-window model has focused on tracking basic counts and other simple aggregates (e.g., sums) over one-dimensional streams in a centralized setting. Recent work has also considered the case of distributed data; however, no existing techniques can handle flexible, complex aggregate queries over rapid, high-dimensional distributed data streams, e.g., with each dimension corresponding to the number of packets originating by an IP address, and the number of possible IP addresses reaching 2^{48} for IPv6.

Example: Recent work on effective network-monitoring systems (e.g., for detecting DDoS attacks or network-wide anomalies in large-scale IP networks) has stressed the importance of an efficient *distributed-triggering* functionality [26,28, 24,23]. In their early work, Jain et al. [26] discuss a generic distributed attack-detection scheme relying on the ability to maintain frequency statistics for high-dimensional data over sliding windows. In particular, each node (e.g., a network router implementing Cisco's Netflow protocol, a wireless access point, or a peer in a P2P network) maintains a sliding-window count of all observed messages for each target IP address. If this count exceeds a pre-determined threshold, which is determined based on the capacity of the target machine (possibly expressing the fair share of each client to the target machine), an event is triggered to a central coordinator as a warning of possible overloading. The coordinator then collects network-wide statistics to monitor overloaded nodes or abnormal behavior. More recent efforts have focused on different variants and extensions of this basic scheme, often requiring more extensive data/statistics collection and more sophisticated analyses [24,23]. (Note that such data collection mechanisms are supported by commercial products, such as the Cisco Netflow Collection Engine solution.)

The ability to efficiently summarize high-dimensional data over sliding windows is obviously crucial to such monitoring schemes, given the tremendous volume of network-data streams and their massive domain sizes (e.g., 2^{48} for IPv6 addresses). This raises a critical need for synopsis data structures that can compactly capture accurate frequency statistics for a vast domain space over sliding windows. Furthermore, to enable the coordinator to aggregate data coming from different nodes (a requirement for detecting DDoS attacks), we need to be able to *compose* individually constructed synopses to a single synopsis which can capture the global state of the network and help isolate network-wide abnormalities. Thus, we are faced with the difficult challenge of designing effective, composable synopses that can support potentially complex sliding-window analysis queries over massive, distributed network-data streams. □

Note that similar requirements are frequently observed in other domains, e.g., for identifying misbehaving nodes in large wireless networks, for training of classifiers with distributed training data that expires over time, and for ranking products in a cloud-based e-shop, based on the number of recent visits of each product.

Our Contributions. In this paper, we consider the problem of answering potentially complex continuous queries over distributed, high-dimensional data streams in the sliding-window model. Our contributions can be summarized as follows.

• **ECM-Sketches for Sliding-Window Streams.** We introduce a novel sketch synopsis (termed *ECM-sketch*) that allows effective summarization of streaming data over both time-based and count-based sliding windows with probabilistic accuracy guarantees. In a nutshell, our ECM-sketch combines the well-known Count-Min sketch structure [11] for conventional streams with state-of-the-art tools for sliding-window statistics. The end result is a sliding-window sketch synopsis that can provide provable, guaranteed-error performance for point, as well as inner-product, queries, and can be employed to address a broad class of queries, such as maintaining frequency statistics, finding heavy hitters, and computing quantiles in the sliding-window model.

• **Time-based Sliding Windows over Distributed Streams.** Focusing on distributed environments, we demonstrate how ECM-sketches summarizing time-based sliding windows of

individual streams can be composed to generate a guaranteed-error ECM-sketch synopsis of the order-preserving merging of all streams. While conventional Count-Min sketches are trivially composable, composing ECM-sketches is more challenging since it requires merging of the sliding-window statistics maintained in the sketch. Therefore, as part of our merging solution for ECM-sketches, we also provide the theoretical foundations and an efficient algorithm for merging sliding window statistics of deterministic algorithms [16, 21]. This is an important result on its own given the wide applicability of these algorithms, as well as their substantially higher efficiency and compactness compared to randomized sliding window algorithms, which are more easily composable [21, 35]. This increased efficiency comes at the cost of a slight inflation of the worst-case error guarantee due to the composition, which however can be easily controlled, even in large hierarchical networks with iterative mergings.

• **Continuous Query Monitoring for Complex Queries over Distributed Streams.** We show how ECM-sketches can be exploited in the context of the geometric framework of Sharfman et al. [33] for continuous monitoring of sliding-window queries over distributed streams. We demonstrate the sketch-enhanced geometric framework by addressing two frequent requirements of distributed stream monitoring applications: (a) maintaining the set of items with a frequency surpassing a threshold (e.g., the IP addresses that exchange an excessive amount of messages over a sliding window), and, (b) maintaining an estimate for the self-join size of a stream over the sliding window, a useful measure for constructing efficient distributed query execution plans. Empowered by the compactness and efficiency of the underlying sketches, the geometric framework can now monitor such queries in a both computational-efficient and network-efficient manner.

• **Experimental Study and Validation.** We perform a thorough experimental evaluation of our techniques using two massive real-life data sets, in both centralized and distributed settings. The results of our study verify the efficiency and effectiveness of our ECM-sketch synopses in a variety of applications, and expose interesting functional trade-offs. When compared to algorithms based on randomized sliding window synopses – which are the only ones that were considered for composition up to now – ECM-sketches reduce the memory and computational requirements by at least one order of magnitude with a very small loss in accuracy. Similar savings apply to the network requirements.

2 Related Work

Centralized and Distributed Data Streams. Most prior work on data-stream processing has focused on developing space-efficient, one-pass algorithms for performing a wide range of *centralized, one-shot computations* on massive data streams; examples include computing quantiles [22], estimating distinct values [19], counting frequent elements (i.e., "heavy hitters") [6, 10], and estimating join sizes and stream norms [1, 11]. Out of these efforts, flexible, general-purpose sketch summaries, such as the AMS [1] and the Count-Min sketch [11] have found wide applicability in a broad range of stream-processing scenarios. More recent efforts have also concentrated on *distributed-stream* processing, proposing communication-efficient streaming tools for handling a number of query tasks, including distributed tracking of simple aggregates [30], quantiles [9], and join aggregates [8], as well as monitoring distributed threshold conditions [33]. All the above-referenced works assume a traditional, "full-history" data stream and do not address the issues specific to the sliding-window model.

Sliding-Window Stream Queries. As mentioned earlier, the bulk of existing work on the sliding-window model has focused on algorithms for maintaining simple statistics, such as basic counts and sums, in space and time that is significantly sub-linear (typically, poly-logarithmic) in the sliding-window size N. *Exponential histograms* [16] are a state-of-the-art deterministic technique for maintaining ϵ-approximate counts and sums over sliding windows, using $O(\frac{1}{\epsilon}\log^2 N)$ space. *Deterministic waves* [21] solve the same basic counting/summation problem with the same space complexity as exponential histograms, but improve the worst-case update time complexity to $O(1)$; on the other hand, *randomized waves* [21] rely on randomization through hashing to track *duplicate-insensitive* counts (i.e., `COUNT-DISTINCT` aggregates) over sliding windows. While randomized waves can be easily composed (in distributed settings), they come with an increased space requirement of $O(\frac{\log(1/\delta)}{\epsilon^2}\log^2 N)$, where δ is a small probability of failure. Xu et al. [35] describe a randomized, sampling-based synopsis, very similar to randomized waves, for tracking sliding-window counts and sums with out-of-order arrivals (e.g., due to network delays) in a distributed setting. As with randomized waves, their space requirements are also quadratic in the inverse approximation error; furthermore, their approach requires knowledge of the maximum number of elements in any sliding window (to set up the synopsis data structure), which could be problematic in dynamic, widely-distributed environments. Cormode et al. [14] also propose randomized techniques for handling out-of-order arrivals for tracking duplicate-insensitive sliding window aggregates. To address the high cost associated with randomized data structures, Busch and Tirthapura propose a deterministic structure for handling out-of-order arrivals in sliding windows [3]. Similar to the other deterministic structures, this structure also does not allow composition and focuses only on basic counts and sums.

More recent works develop protocols for efficient continuous monitoring of sliding window aggregates over dis-

tributed architectures [5,12,13,15]. These techniques typically focus on reducing the network requirements for maintaining random samples or simple statistics (such as basic counts, heavy hitters, and quantiles) with accuracy guarantees. Some aspects of these techniques could find use in the case of ECM-sketches as well. In this work we have selected to build the continuous monitoring scheme over the geometric method. The geometric method goes beyond monitoring simple linear aggregates, by enabling distributed monitoring of (possibly) complex functions that can be expressed over the average values of the monitored variables, e.g., self-join and inner product sizes. As such, we are able to monitor any function that can be supported by the ECM-sketch.

Going beyond counts, sums, and simple aggregates, there is surprisingly little work in the more general problem of maintaining general, frequency-distribution synopses over high-dimensional streaming data in the sliding-window model. Hung and Ting [25] and Dimitropoulos et al. [17] propose synopses based on Count-Min sketches for tracking heavy hitters and frequency counts over sliding windows; still, their techniques rely on keeping simple equi-width counters within the sketch, and, thus, cannot provide any meaningful error guarantees, especially for small query ranges. Similarly, the hybrid histograms of Qiao et al. [32] combine exponential histograms with simplistic equi-width histograms for answering sliding-window range queries; again, these structures cannot give meaningful bounds on the approximation error and cannot be composed in a distributed setting.

Chakrabati et al. briefly sketched the combination of Count-min sketches and exponential histograms for computing the entropy of a stream over a sliding window [4]. Compared to that work, our work goes several steps forward. First, we provide important materialization details, which were not discussed in [4]. For example, we show how to automatically choose the sketch configuration that satisfies the accuracy requirements and minimizes space complexity. Second, we present merging algorithms for ECM-sketches (even the ones that are based on deterministic sliding window algorithms), which are necessary in many domains involving distributed stream processing. Finally, we present algorithms for distributed continuous monitoring using ECM-sketches.

An early version of this work has previously appeared in [31]. Compared to [31], in this article we follow a more rigorous analysis, which leads to tighter theoretical error bounds, and to substantial reduction of the size of the sketch. Sketch size is typically reduced by a factor of three for ECM-sketches based on deterministic sliding window algorithms, and by a factor of six for the ones based on randomized algorithms. Furthermore, we elaborate on continuous function monitoring with ECM-sketches, which was only briefly mentioned in the original paper. This elaboration includes a novel efficient monitoring algorithm, accompanied with proof of correctness, and with extensive experimental evaluation.

3 Preliminaries

ECM-sketches combine the functionalities of Count-Min sketches [11] and exponential histograms [16]. We now describe the two structures, focusing on the aspect related to our work.

Count-Min Sketches. Count-Min sketches are a widely applied sketching technique for data streams. A Count-Min sketch is composed of a set of d hash functions, $h_1(\cdot)$, $h_2(\cdot)$, ..., $h_d(\cdot)$, and a 2-dimensional array of counters of width w and depth d. Hash function h_j corresponds to row j of the array, mapping stream items to the range of $[1 \dots w]$. Let $CM[i,j]$ denote the counter at position (i,j) in the array. To add an item x of value v_x in the Count-Min sketch, we increase the counters located at $CM[j,h_j(x)]$ by v_x, for $j \in [1 \dots d]$. A point query for an item q is answered by hashing the item in each of the d rows and getting the minimum value of the corresponding cells, i.e., $\min_{j=1}^{d} CM[j,h_j(q)]$. Note that hash collisions may cause estimation inaccuracies – only overestimations. By setting $d = \lceil \ln(1/\delta) \rceil$ and $w = \lceil e/\epsilon \rceil$, where e is the base of the natural logarithm, the structure enables point queries to be answered with an error of less than $\epsilon||a||_1$, with a probability of at least $1-\delta$, where $||a||_1$ denotes the number of items seen in the stream. Similar results hold for range and inner product queries.

Exponential Histograms. Exponential histograms [16] are a deterministic structure, proposed to address the basic counting problem, i.e., for counting the number of true bits in the last N stream arrivals. They belong to the family of methods that break the sliding window range into smaller windows, called buckets or basic windows, to enable efficient maintenance of the statistics. Each bucket contains the aggregate statistics, i.e., number of arrivals and bucket bounds, for the corresponding sub-range. Buckets that no longer overlap with the sliding window are expired and discarded from the structure. To compute an aggregate over the whole (or a part of) sliding window, the statistics from all buckets overlapping with the query range are aggregated. For example, for basic counting, aggregation is a summation of the number of true bits in the buckets. A possible estimation error can be introduced due to the oldest bucket inside the query range, which usually has only a partial overlap with the query. Therefore, the maximum possible estimation error is bounded by the size of the last bucket.

To reduce the space requirements, exponential histograms maintain buckets of exponentially increasing sizes. Bucket boundaries are chosen such that the ratio of the size of each bucket b with the sum of the sizes of all buckets more recent than b is upper bounded. In particular, the following invari-

Notation	Description
N	Length of the sliding window, in time units or in number of arrivals
$h_i(\cdot)$	Hash function i of the Count-Min sketch
a_r, b_r	Substream of stream a, b, within the query range r
$f_a(x, r)$	Frequency of item x in stream a, within the query range r
$E_a(i, j, r)$	Estimated value of the ECM-sketch counter for stream a in position (i, j) for query range r
$a_r \odot b_r, \widehat{a_r \odot b_r}$	Real and estimated inner product of a_r and b_r
$u(N, S)$	Upper bound of number of arrivals on stream S within the sliding window of length N

Table 1 Frequently used notation.

ant (*invariant 1*) is maintained for all buckets j: $C_j/(2(1 + \sum_{i=1}^{j-1} C_i)) \leq \epsilon$ where ϵ denotes the maximum acceptable relative error and C_j denotes the size of bucket j (number of true bits arrived in the bucket range), with bucket 1 being the most recent bucket. Queries are answered by summing the sizes of all buckets that fully overlap the query range, and half of the size of the oldest bucket, if it partially overlaps the query. The estimation error is solely contained in the oldest bucket, and is therefore bounded by this invariant, resulting to a maximum relative error of ϵ.

4 ECM-Sketches

We now describe ECM-sketches (short for Exponential Count-Min sketches), a composable sketch for maintaining data stream statistics over sliding windows in distributed environments. ECM-sketches combine the functionality of Count-Min sketches and sliding windows, and support both time-based and count-based sliding windows under the cash register model. Therefore, they can be used for compactly summarizing high-dimensional streams over sliding windows, i.e., to maintain the observed frequencies of the stream items within the sliding window range.

The core of the structure is a modified Count-Min sketch. Count-Min sketches alone cannot handle the sliding window requirement. To address this limitation, ECM-sketches replace the Count-Min counters with sliding window structures. Each counter is maintained as a sliding window, covering the last N time units, or the last N arrivals, depending on whether we need time-based or count-based sliding windows.

As discussed in Section 2, there have been several algorithms proposed for sliding window maintenance. Due to the large expected number of sliding window counters in ECM-sketches, we require an algorithm with a small memory footprint. Existing randomized algorithms for sliding window synopses (as discussed in Section 2) appear to have a quadratic dependence to ϵ and are therefore not good for our purposes. Instead, we employ exponential histograms, a compact and efficient deterministic synopsis [16]. Each of the Count-Min counters is implemented as an exponential histogram, configured to provide an ϵ approximation for any query within a sliding window of length N, i.e., the estimation $\hat{x}$ of the counter for any query range within the sliding window length is in the range of $(1 \pm \epsilon)x$ of the true value x of the counter. We will be discussing our choice for exponential histograms again in more detail in the following section, where we will consider alternative deterministic and randomized algorithms.

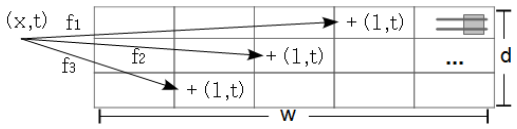

Fig. 1 Adding an element to the ECM-sketch.

Adding an item x to the structure is similar to the case of standard Count-Min sketches. The process for time-based sliding windows is depicted in Figure 1. First, the counters $CM[j, h_j(x)]$, where $j \in \{1 \dots d\}$, corresponding to the d hash functions are detected. For each of the counters, we register the arrival of the item at time t, and remove all expired information, i.e., the buckets of the exponential histogram that have no overlap with the sliding window range. The process for count-based sliding windows is similar, but instead of registering each arrival with system time t, we register it with the count of arrivals since the beginning of the stream.

The challenges that need to be addressed for the integration of exponential histograms with Count-Min sketches are: (a) to take into account the additional error introduced by the sliding window counters for deriving the accuracy guarantees for ECM-sketches (presented in the remainder of this section), and, (b) to enable composition of a set of ECM-sketches to a single ECM-sketch representing the order-preserving merging of the corresponding individual streams (Section 5).

4.1 Query Answering

We now explain how ECM-sketches support point queries, inner product and self-join size queries, and derive probabilistic guarantees for the estimation accuracy. Our analysis covers both sliding window models, i.e., count-based and time-based.

Point Queries. A point query (x, r) is a combination of an item identifier x, and the query range r defined either as number of time units or number of arrivals. Point queries are executed as follows. The query item is hashed to the d counters $CM[j, h_j(x)]$ where $(j \in \{1 \dots d\})$, and the estimate of each counter $E(j, h_j(x), r)$ for the query range

is computed. The estimate value for the frequency of x is $\hat{f}(x,r) = \min_{j=1...d} E(j, h_j(x), r)$.

Let δ_{cm} and ϵ_{cm} denote the configuration parameters of the Count-Min sketch, whereas ϵ_{sw} denotes the configuration parameter of the exponential histogram. With $||a_r||_1$ we denote the number of arrivals within the query range. The following theorem provides probabilistic guarantees for the approximation quality for point queries and enables optimally setting ϵ_{cm} and ϵ_{sw}. As is typical for small-space sketches, the error guarantees are relative to the stream characteristics, i.e., the L1 norm.

Theorem 1 *For any ϵ within $(0,1)$, an ECM-sketch constructed with $\epsilon_{cm} = \frac{\epsilon}{1+\epsilon}$ and $\epsilon_{sw} = \epsilon$ satisfies $Pr[|\hat{f}(x,r) - f(x,r)| \leq \epsilon||a_r||_1] \geq 1 - \delta_{cm}$. Furthermore, the aforementioned combination of ϵ_{cm} and ϵ_{sw} minimizes the space complexity of the sketch.*

Proof Special case of Theorem 3, with $\delta_{sw} = 0$. □

Inner Product and Self-Join Size Queries. Another frequent query type is the cardinality of the inner product. Given two streams a and b, the inner product is defined as $a \odot b = \sum_{x \in \mathcal{D}} f_a(x) \times f_b(x)$, where $\mathcal{D}$ denotes the input domain, i.e., the distinct input elements, and $f_a(x)$ (resp. $f_b(x)$) denotes the frequency of element x in stream a (resp. stream b). Self-join size queries, also called the second frequency moment F_2, are a special case of inner product queries defined over a single stream: $F_2(a) = \sum_{x \in \mathcal{D}} (f_a(x))^2$. Both inner product and self-join size queries are very important for databases, e.g., for building query execution plans, and they can be efficiently and accurately estimated for streams in both the cash register and turnstile model [29]. However, similar to point queries, computing these queries over sliding windows is challenging.

ECM-sketches can be used to address this type of queries as well. Let a_r (resp., b_r) denote the substream of stream a (resp., b) within the query range. With CM_a we denote the corresponding ECM-sketch for stream a_r, and with $E_a(i,j,r)$ we denote the estimated value of the counter of CM_a in position (i,j), for query range r. Also, $f_a(x,r)$ and $\hat{f}_a(x,r)$ denote the real and estimated frequency of x in stream a_r.

The inner product of two streams a and b in a range r is defined as $a_r \odot b_r = \sum_{x \in \mathcal{D}} f_a(x,r) f_b(x,r)$. Using the ECM-sketches of a and b, we estimate it as follows: $\widehat{a_r \odot b_r} = \min_j (\widehat{a_r \odot b_r})_j$, where $(\widehat{a_r \odot b_r})_j = \sum_{i=1}^{w} E_a(i,j,r) \times E_b(i,j,r)$. The following theorem bounds the approximation error.

Theorem 2 *For any ϵ within $(0,1)$, two ECM-sketches constructed with $\epsilon_{cm} = \epsilon/(\epsilon+1)$ and $\epsilon_{sw} = \sqrt{\epsilon+1} - 1$ satisfy $Pr[|\widehat{a_r \odot b_r} - a_r \odot b_r| \leq \epsilon||a_r||_1||b_r||_1] \geq 1 - \delta_{cm}$. Furthermore, the aforementioned combination of ϵ_{cm} and ϵ_{sw} minimizes the space complexity of the sketches.*

Proof In the appendix.

Time-based ECM-Sketches. Exponential histograms were originally developed for count-based sliding windows (e.g., count the number of true bits in the last 100 arrivals), but they can be extended for time-based sliding windows as well (e.g., count the number of true bits arriving in the last 1000 sec.). Our solution can handle concurrent bit arrivals as well as arrivals at arbitrary rates, and similar to the count-based histograms, its memory footprint (the number of buckets) scales logarithmically with the number of arrivals within the sliding window. First, each entry in the data structure is identified using its arrival time, instead of using its position in the stream. To reduce memory, arrival times are stored in wraparound counters of $O(\ln N)$ bits, where N is the length of the sliding window, e.g., in milliseconds. Second, entries expire based on their arrival time, and not on their position in the stream. Finally, we require an upper bound of the number of arrivals within the sliding window time range for each stream S, denoted as $u(N,S)$. Note that this is required only for computing the maximum memory requirements of the structure a priori; it does not have an impact on the actual required memory or quality of ECM-sketches. Furthermore, the bound can be very loose without a noticeable change on the estimated space requirements, because space complexity increases only logarithmically with $u(N,S)$.

Complexity. We use N to denote the length of the sliding window, either in number of arrivals or in time (depending on the desired sliding window model), and $u(N,S)$ as defined earlier. Also, $g(N,S) = \max(u(N,S), N)$; function g is used to enable unified cost expressions for both the time-based and count-based sliding window model.

To get an ϵ_{sw}-approximation of the number of one-bits in the sliding window, exponential histograms require $O(\ln N + \ln\ln(u(N,S)))$ memory per bucket, to store the bucket size and bucket boundaries. The number of buckets is $O(\ln(u(N,S))/\epsilon_{sw})$, yielding a total memory of $O(\ln^2(g(N,S))/\epsilon_{sw})$. The update cost per element is $O(\ln(u(N,S)))$ worst-case, and $O(1)$ amortized time. Queries covering the whole sliding window are executed in constant time. For queries with range $N' < N$, the required time is $O(\ln(u(N,S)/\epsilon_{sw}))$. The extra time is required for finding the oldest bucket overlapping with the query, assuming sequential access. If the storage model of the buckets supports random access, e.g., a fixed-length array, then this time can be further reduced to $O(\ln(\ln(u(N,S)/\epsilon_{sw})))$ with binary search.

The space complexity of ECM-sketches is as follows. For the Count-Min array, we require an array of width $w = \lceil e/\epsilon_{cm} \rceil$ and depth $d = \lceil \ln(1/\delta) \rceil$. Each cell in the array stores an exponential histogram, requiring $O(\ln^2(g(N,S))/\epsilon_{sw})$ bits. Therefore, the total required memory is $O(\frac{1}{\epsilon_{sw}\epsilon_{cm}} \ln^2(g(N,S)) \ln(1/\delta)) =$

$O(\frac{1}{\epsilon^2}\ln^2(g(N,S))\ln(1/\delta))$. Concerning time complexity, adding an element requires computing d hash functions, and updating d separate exponential histograms. The amortized complexity for each arrival is therefore $O(d) = O(\ln(1/\delta))$, whereas the worst-case complexity is $O(d\ln(u(N,S))) = O(\ln(u(N,S))\ln(1/\delta))$. Finally, query execution takes $O(\ln(1/\delta))$ time for a query of range N' equal to N. For $N' < N$, the execution cost is $O(d\ln(u(N,S))/\epsilon_{sw}) \leq O(\ln(1/\delta)\ln(u(N,S))/\epsilon)$ with sequential access to buckets, e.g., using a linked list. With random access support, binary search can be used for finding the last relevant bucket for each query, reducing the query cost to $O(\ln(1/\delta)\ln(\ln(u(N,S))/\epsilon))$.

4.2 Alternative Algorithms for Sliding Windows

Sliding window counters can also be materialized using other sliding window algorithms. In the literature, two such algorithms are particularly well-known: (a) deterministic waves, and, (b) randomized waves [21]. We now show how ECM-sketches can incorporate these algorithms, and discuss the positive and negative aspects of each variant.

Deterministic Waves. Deterministic waves [21] have identical memory requirements with exponential histograms, and they outperform exponential histograms with respect to worst-case complexity for updates, requiring always constant time. As such, the space and computational complexity of ECM-sketches based on deterministic waves is identical to that of sketches based on exponential histograms, with the only difference being the worst-case update complexity, which is $O(\ln(1/\delta))$.

A downside of deterministic waves is that they require knowledge of the upper bound of the number of arrivals $u(N,S)$ during the initialization of the data structures, to decide on the required number of queues/levels. Any overestimation of $u(N,S)$ is therefore translated to increased space requirements – logarithmic with $u(N,S)$. It is important to note that this constraint is substantially less limiting compared to the constraints of previous algorithms, e.g., [35], which required an upper bound for the total number of items in *all* streams, and therefore could not be applied to dynamic networks with an unknown number of participating nodes and streams.

Randomized Waves. Randomized waves [21] provide (ϵ,δ) approximation for the basic counting problem, i.e., $Pr[|\hat{x} - x| \leq \epsilon_{sw}x] \geq 1-\delta_{sw}$, where $\hat{x}$ and x denote the estimated and real number of true bits in the sliding window range respectively. They have substantially higher space complexity compared to their deterministic counterparts – $O(\ln(1/\delta_{sw})/\epsilon_{sw}^2)$ instead of $O(1/\epsilon_{sw})$. Nevertheless, they are important for distributed applications as they enable composition without causing an inflation of the worst-case error bounds; deterministic counterparts did not originally support any composition functionality. Therefore, we also consider randomized waves for integration with our ECM-sketch structures.

Theorem 3 *For any ϵ within $(0,1)$, an ECM-sketch constructed with $\epsilon_{cm} = \frac{\epsilon}{1+\epsilon}$, $\epsilon_{sw} = \epsilon$, and $\delta_{sw} = \delta_{cm} = \delta/2$ satisfies $Pr[|\hat{f}(x,r) - f(x,r)| \leq \epsilon||a_r||_1] \geq 1-\delta_{sw}-\delta_{cm}$. Furthermore, the aforementioned combination of ϵ_{cm} and ϵ_{sw} minimizes the space complexity of the sketch.*

Proof In the appendix.

The space complexity of ECM-sketches based on randomized waves is derived by multiplying the space complexity of the two basic structures: $O\left(\ln(1/\delta_{cm})\ln(1/\delta_{sw})\ln^2(g(N,S))/(\epsilon_{cm}\epsilon_{sw}^2)\right) = O\left(\ln^2(\delta)\ln^2(g(N,S))/\epsilon^3\right)$. Inserting a new element requires $O(\ln(\delta_{cm})\ln(\delta_{sw})) = O(\ln^2(\delta))$ amortized time, and $O(\ln(\delta_{cm})\ln(\delta_{sw})\ln(u(N,S))) = O(\ln^2(\delta)\ln(u(N,S)))$ worst-case time. Finally, query execution takes $O(\ln(\delta_{cm})\ln(\delta_{sw})\ (\ln(u(N,S)) + 1/\epsilon_{sw}^2)) = O(\ln^2(\delta)(\ln(u(N,S)) + 1/\epsilon^2))$ with sequential access to buckets and $O(\ln(\delta_{cm})\ln(\delta_{sw})\ (\ln\ln(u(N,S))+\ln(1/\epsilon_{sw}^2))) = O(\ln^2(\delta)(\ln\ln(u(N,S))+\ln(1/\epsilon_{sw}^2)))$ time with random access.

Table 2 summarizes the main results for the combination of ECM-sketches and the three sliding window structures. The results correspond to both time-based and count-based sliding windows.

5 Order-Preserving Merging

For many distributed applications, such as the network monitoring application described in the introduction, we require merging of individual ECM-sketches $CM_1, CM_2, \ldots, CM_n$, each one corresponding to stream S_1, S_2, …, S_n, to get a single ECM-sketch $CM_\oplus$ that corresponds to the logical stream $S_\oplus = S_1 \oplus S_2 \oplus \ldots \oplus S_n$. The $\oplus$ operator is defined as a merging operator that preserves the ordering and arrival time of the events. Standard Count-Min sketches allow merging, as long as all sketches are constructed with identical dimensions and hash functions. For this, they rely on the linearity of the Count-Min counters, which are simple integers in the general case. However, this does not trivially hold for ECM-sketches, where the counters are not simple numbers but complex sliding window structures, since exponential histograms (as well as all other deterministic sliding window structures), do not support this kind of merging. Although randomized structures enable lossless merging (cf. Section 5.2), they come with a substantially higher space complexity, and are thus not preferable for ECM-sketches. Therefore, we first consider the order-preserving merging of

	Exponential Histogram	Deterministic Wave	Randomized Wave
Memory	$O\left(\frac{1}{\epsilon^2}\ln(\frac{1}{\delta})\ln^2(g(N,S))\right)$	$O\left(\frac{1}{\epsilon^2}\ln(\frac{1}{\delta})\ln^2(g(N,S))\right)$	$O\left(\frac{1}{\epsilon^3}\ln^2(\delta)\ln^2(g(N,S))\right)$
Amort. update	$O(\ln(1/\delta))$	$O(\ln(1/\delta))$	$O(\ln^2(\delta))$
Worst update	$O(\ln(1/\delta)\ln(u(N,S)))$	$O(\ln(1/\delta))$	$O(\ln^2(\delta)\ln(u(N,S)))$
Query	$O(\ln(1/\delta)\ln(u(N,S))/\epsilon)$	$O(\ln(1/\delta)\ln(u(N,S))/\epsilon)$	$O(\ln^2(\delta)(\ln(u(N,S))+1/\epsilon^2))$

Table 2 Computational and space complexity of ECM-sketches. Function $g(N,S)$ is used as a shortcut for $\max(u(N,S),N)$.

deterministic sliding window structures. Note that this problem is interesting in itself, since these data structures are widely used in the literature for maintaining statistics over sliding windows. We then extend our results to cover merging of randomized waves, and of ECM-sketches.

For completeness, before presenting the details of our merging algrorithm, we note that other types of merging are also possible. For example, Gibbons and Tirthapura [21], have considered utilizing more than one randomized waves for generating their position-wise union, i.e., for maintaining count-based sliding window statistics. Their scenario and query types are fundamentally different than ours.

5.1 Merging of Exponential Histograms

Consider a set of exponential histograms $EH_1, EH_2, \ldots, EH_n$, summarizing time-based sliding windows. All are configured to cover a sliding window of N time units. The merging operation is denoted with $\oplus$, i.e., $EH_\oplus = EH_1 \oplus EH_2 \oplus \ldots \oplus EH_n$. With EH_i^j we denote bucket j of EH_i, and $|EH_i^j|$ denotes the bucket size (number of true bits). By convention, buckets are numbered such that bucket 1 is the most recent. The ending time of the bucket is denoted as $e(EH_i^j)$. To ease exposition, we use $s(EH_i^j)$ to denote the starting time of the bucket, even though this is not explicitly stored in the buckets. By construction, the starting time of a bucket is equal to the ending time of the previous bucket, i.e., $s(EH_i^j) = e(EH_i^{j-1})$.

To construct $EH_\oplus$ our methodology considers the individual exponential histograms as logs. The basic idea is to reconstruct $EH_\oplus$ by assuming that half of the elements arrive at the starting time of each bucket, and the remaining at the ending time of the bucket. Precisely, let $\mathcal{B}$ denote the list containing all buckets of all sliding windows. We initialize an empty time-based exponential histogram with error ϵ', configured to keep the last N time units, and a maximum of $\sum_{i=1}^{n}|EH_i|$ elements. For each bucket $\mathcal{B}[i] \in \mathcal{B}$, we simulate the insertion in $EH_\oplus$ of $|\mathcal{B}[i]|$ true bits. Half of the bits are inserted with timestamp $s(\mathcal{B}[i])$, and the other half at time $e(\mathcal{B}[i])$. Insertions are simulated in the order defined by the starting and ending timestamps of the buckets.

Theorem 4 *Consider n time-based exponential histograms EH_1, EH_2, ..., EH_n, initialized with error parameter ϵ, and covering the same time range. The exponential histogram $EH_\oplus$ initialized with error parameter ϵ', and constructed with the proposed merging algorithm answers any query within its time range for the stream $S_\oplus$ with a maximum relative error of $(\epsilon + \epsilon' + \epsilon\epsilon')$.*

We will now give the intuition of the proof. The formal proof is presented in the appendix. Each exponential histogram EH of stream S configured with error parameter ϵ can be used to reconstruct an approximate stream S', as follows: For each bucket b in EH, add $|b|/2$ true bits in time $s(b)$, and $|b|/2$ true bits in time $e(b)$. We argue that answering any query with starting time s_q within the range of EH using the reconstructed stream S' will result to a maximum relative error ϵ. Let b_j be the bucket s.t. $s(b_j) < s_q \leq e(b_j)$. Therefore, the accurate answer x of the query for stream S is lower bounded by $l = \sum_{i=1}^{j-1}|b_i| + 1$ and upper bounded by $h = \sum_{i=1}^{j-1}|b_i| + |b_j|$. By construction, the reconstructed stream will contain a total of $\sum_{i=1}^{j-1}|b_i| + |b_j|/2$ items with timestamp greater than or equal to s_q. Therefore, answering the query by counting the number of true bits in the reconstructed stream with timestamp after s_q will have a maximum error of $\max(h - \sum_{i=0}^{j-1}|b_i| + |b_j|/2, \sum_{i=0}^{j-1}|b_i| + |b_j|/2 - l) = |b_j|/2$. By invariant 1 of exponential histograms, $|b_j|/2 \leq \epsilon(1 + \sum_{i=1}^{j-1}|b_i|) \leq \epsilon x$. Therefore, the maximum difference between the answer estimated by stream S' and the correct answer x will be less than or equal to ϵx.

Our merging algorithm is equivalent to reconstructing each stream S_i' from exponential histogram EH_i, and using these to recreate an exponential histogram $EH_\oplus$. The reconstruction of stream S' introduces a maximum relative error ϵ, as explained above. Summarizing S' with a new exponential histogram we get an additional error ϵ'. However, ϵ' is relative on the answer provided by stream S', and not by S. Therefore, the absolute error due to the exponential histogram summarization will be $\epsilon' x'$, where $x' \in (1 \pm \epsilon)x$ and x denoting the accurate answer on S_i. Summing both errors, we get a total relative error of $\epsilon + \epsilon' + \epsilon\epsilon'$.

For the special case when $\epsilon' = \epsilon$, the maximum relative error becomes $2\epsilon + \epsilon^2$. Concerning space and computational complexity, $EH_\oplus$ behaves as a standard exponential

histogram, and therefore has the same complexity as presented in [16]. □

Multi-level Merging. It is frequently desired to merge sliding windows in more than one levels. For example, consider a hierarchical P2P network, where each peer maintains its own exponential histogram, and pushes it to its parent for merging at regular intervals. Since the merged exponential histograms have the same properties as the individual exponential histograms (albeit with a higher ϵ), the above analysis also supports iterative merging of exponential histograms.

There are two types of approximation error that influence the estimation of a merged exponential histogram. A possible approximation error, denoted as err_1, is introduced due to halving of the size of the last bucket of the merged exponential histogram. This error occurs only at query time, and is independent of the number of performed merges. Therefore, at a multi-level merging scenario this error does not need to be propagated at the intermediary exponential histograms. A second type of error, termed as err_2, occurs due to the inclusion (exclusion) of data that arrived before (after) the query starting time in buckets that are accounted (not accounted) in the query result.

It turns out that the error err_2 is additive at the worst case (in absolute value). For instance, in the lowest level (Level 0) of the hierarchy, merging two exponential histograms (all with relative error ϵ), having a true number of bits (in a given query range) equal to i_1 and i_2, will result at a maximum value for $\text{err}_2 \leq \epsilon(i_1 + i_2)$. In Level 1, in addition to the previous possible errors, $\epsilon(i_1+i_2)+\epsilon(i_3+i_4)$ stream items may be incorrectly registered at the wrong side of the query start time. A recursive repetition for h levels results to $\text{err}_2 \leq h\epsilon i$, where $i = \sum_j i_j$. The total absolute error (including err_1) then becomes $\text{err} = \text{err}_2 + \text{err}_1 \leq h\epsilon i + \epsilon(i + h\epsilon i)$, resulting to a maximum relative error of $h\epsilon(1 + \epsilon) + \epsilon$.

In many applications, the number of merging levels can be predicted, or even controlled when constructing the network topology. For example, consider DHT-based or hierarchical P2P topologies, which typically enable a balanced-tree access to the peers of height $h = \log(N)$, where N is the number of nodes. In such systems, initializing the individual exponential histograms with error $\frac{\sqrt{1+2h+h^2+4h\epsilon}-1-h}{2h}$ yields a final merged exponential histogram of relative error ϵ. Naturally, this causes a slight inflation of the size of the sliding window, by $O(\log(N))$. However, even with this inflation, exponential histograms are – even for extremely large networks – substantially smaller and more efficient than randomized data structures that enable error-free merging in the expense of memory proportional to $O(\ln(1/\delta)/\epsilon^2)$ (see also Section 5.2).

Deterministic Waves. The merging technique trivially extends for deterministic waves. Recall that each wave is composed of l levels, each covering a different range. To perform the merging, we start from the lowest wave level $l - 1$, and switch to a higher level every $(1/\epsilon + 1)/2$ bits, i.e., when the first entry in the higher level has arrived before the next entry in the current level. Repeating the calculation of the error bounds for the merging of deterministic waves becomes straightforward when we notice that invariant 1 of the exponential histograms is also true for deterministic waves.

	EH_1		EH_2				
Bucket id	2	1	5	4	3	2	1
Size	1	1	8	4	2	1	1
Completion time	3	20	3	5	10	15	19
Arrivals	500	1000	900	950	980	990	1000

Fig. 2 An example why merging of count-based exponential histograms is not possible.

Count-based Exponential Histograms. Although exponential histograms cover both time-based and count-based sliding windows, merging of exponential histograms is specific to time-based sliding windows. Count-based sliding windows do not contain sufficient information for enabling order-preserving merging. Even storing the system-wide time of the buckets would not be sufficient to allow such a merging. To illustrate this limitation, consider the two count-based exponential histograms depicted in Fig. 2. For each bucket we store the size of the bucket, the bucket completion time and the total number of arrivals until that time. An arrival in count-based sliding windows might be a true or a false bit. An example query can then be: *how many true bits arrived in the last 100 system-wide arrivals*. If these 100 system-wide arrivals were read between time 19 and 20, then the correct answer would be 1. However, it is also possible that the last 100 system-wide arrivals have arrived between time 3 and time 20, in which case the correct answer could be anything between 2 and 9. The information contained in the two exponential histograms is not sufficient to estimate this type of queries, as it only allows us to preserve the order of the true bits, but looses the order of the false bits, which is also important. Therefore, given only the exponential histograms, it is not possible to merge them in a way that preserves the ordering of both true and false bits. Deterministic and randomized waves also have the same limitation when it comes to order-preserving merging of count-based sliding windows.

5.2 Merging of Randomized Waves

Randomized waves were proposed in [21] to address the problem of distributed union counting: *counting the number of 1's in the position-wise union of t distributed data streams, over a sliding window*. Even though the algorithm of [21] can utilize more than one waves constructed at different nodes to answer queries, it does not consider merging of several waves to generate a single wave. Instead, it as-

sumes that individual randomized waves can be stored and accessed any time, which is inconvenient for large networks. To eliminate this assumption, we now describe a slight variation of the initial algorithm that can produce a single randomized wave out of a set of individual waves, with the same probabilistic accuracy guarantees as the individual waves.

Our algorithm simulates the construction of the merged randomized wave $RW_{\oplus}$ by using only the information included in the individual randomized waves. Consider a set $\mathcal{R}$ of randomized waves RW_1, RW_2, ..., RW_n, configured to store a sliding window of N time units, with error parameters ϵ and δ. The merged randomized wave $RW_{\oplus}$ is initialized with the same ϵ and δ parameters, for storing a maximum of $\sum_{i=1}^{n} |RW_i|$ events over N time units. Each level l of $RW_{\oplus}$ is then constructed by concatenating the corresponding level l from all individual randomized waves, sorting all events based on the timestamp, and keeping the last c/ϵ^2 events. Recall that the number of levels of individual randomized waves is determined based on the maximum number of events in the sliding window. Therefore, it may happen that $RW_{\oplus}$ has more levels than the individual randomized waves. To populate the lower levels of $RW_{\oplus}$, we rehash the events populating the last level of each individual randomized wave, as proposed in [21] when merging different levels from randomized waves.

The process of query execution and the accuracy guarantees remain the same as for the standard randomized waves.

5.3 Merging of ECM-Sketches

Consider a set of ECM-sketches CM_1, CM_2, ..., CM_n with identical dimensions and hash functions. The ECM-sketch $CM_{\oplus}$ with each counter set to the sum of all corresponding counters from the individual sketches (as defined by the $\oplus$ operator), summarizes the information found in the individual sketches:

$$CM_{\oplus}[j,k] = CM_1[j,k] \oplus CM_2[j,k] \oplus \ldots \oplus CM_n[j,k]$$

To bound the estimation error, we consider the two sources of error in the merged ECM-sketch. The error due to the Count-Min sketch ϵ_{cm} does not change, since it only depends on the dimensionality of the Count-Min array, which is fixed. However, the error due to sliding window estimations at each counter might change with each merging. Let ϵ'_{sw} denote the error produced by the merging of the corresponding Count-Min counters, as discussed in Sections 5.1 and 5.2. If ϵ_{sw} and ϵ_{cm} are configured according to Theorem 3, it can be easily shown that ϵ'_{sw} will always be greater than or equal to $\epsilon_{cm}/(1-\epsilon_{cm})$. Then, the error bounds follow directly by Lemma 3: $|\hat{f}(x,r) - f(x,r)| \leq \epsilon'_{sw}||a_r||_1$ with probability $1-\delta_{cm}$.

6 Continuous Function Monitoring with ECM-Sketches

A substantial number of distributed applications requires continuous monitoring of complex functions defined over high-dimensional domains. For example, network administrators frequently require to monitor the (sliding-window) heavy-hitter IP addresses over distributed streams of network packets (e.g., received by the edge routers of the corporate network), as these IPs are potentially launching a DoS attack. ECM-sketches can be exploited in these applications, such that each network node can compactly and efficiently maintain its local state, as well as effectively propagate it over the network. In this section, we show how ECM-sketches can leverage the geometric method [33,27], to enable continuous function monitoring.

We illustrate our technique by addressing two frequent requirements of distributed applications: (a) monitoring items with frequency over a user-defined threshold τ, and, (b) monitoring self-join size queries. In principle, any query type that can be answered by (a sequence of) point queries can be monitored in the lines of the algorithm that we will present for query (a). Some examples include hierarchical heavy hitters, quantiles, range queries, and maximum frequency queries (see also [11,31] for a more detailed discussion on how centralized Count-min sketches and ECM-sketches can address these problems using point queries). It is also straightforward to extend the algorithm for query (b) for inner-product size queries.

Section 6.1 provides an introduction to the geometric method. In Section 6.2 we introduce the integration of ECM-sketches with the geometric method, and discuss the main challenges that need to be addressed. Then, in Section 6.3, we briefly discuss an algorithm for query (a). This discussion serves mainly as a first, simple, example for the integration. An algorithm for query (b) is presented in more detail in Sections 6.4 and 6.5. Our discussion for query (b) includes an efficient monitoring algorithm and novel theoretical results to enable dimensionality reduction of the monitoring problem (from $d \times w$ to d), which translates to drastic network savings and better scalability.

6.1 An Introduction to the Geometric Method

Sharfman et al. [33] consider the basic problem of monitoring *distributed threshold-crossing queries*; that is, monitoring whether $f(\mathbf{v}) < \tau$ or $f(\mathbf{v}) > \tau$ for a possibly complex, non-linear function f and a high-dimensional vector $\mathbf{v}$ computed as the aggregate of the corresponding local/partial vectors $\{\mathbf{v}(p_1), \mathbf{v}(p_2), \ldots, \mathbf{v}(p_n)\}$ at a set of n sites. The key idea of the method is, since it is generally impossible to connect the values of f on the local statistics vectors to the global value $f(\mathbf{v})$, one can employ geometric arguments

to monitor the *domain* (rather than the range) of the monitored function f. The monitoring protocol works as follows. Assume that at any point in time, each site p_i has informed the coordinator of some prior state of its local vector $\mathbf{v'}(p_i)$; thus, the coordinator has an estimated global vector $\mathbf{e} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{v'}(p_i)$. Clearly, the updates arriving at sites can cause the local vectors $\mathbf{v}(p_i)$ to drift too far from their previously reported values $\mathbf{v'}(p_i)$, possibly leading to a violation of the threshold τ. Let $\Delta\mathbf{v}(p_i) = \mathbf{v}(p_i) - \mathbf{v'}(p_i)$ denote the local *delta vector* (due to updates) at site i, and let $\mathbf{u}(p_i) = \mathbf{e} + \Delta\mathbf{v}(p_i)$ be the *drift vector* from the previously reported estimate at site p_i. We can then express the current global statistics vector $\mathbf{v}$ in terms of the drift vectors:

$$\mathbf{v} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{v'}(p_i)+\Delta\mathbf{v}(p_i)) = \mathbf{e}+\frac{1}{N}\sum_{i=1}^{N}\Delta\mathbf{v}(p_i) = \frac{1}{N}\sum_{i=1}^{N}\mathbf{u}(p_i).$$

That is, the current global vector is a convex combination of drift vectors and, thus, guaranteed to lie somewhere within the convex hull of the delta vectors around $\mathbf{e}$. Fig. 3 depicts an example in $d = 2$ dimensions. The current value of the global statistics vector lies somewhere within the shaded convex-hull region; thus, as long as the convex hull does not overlap the inadmissible region (i.e., the region $\{\mathbf{v} \in \mathbb{R}^2 : f(\mathbf{v}) > \tau\}$ in Fig. 3) we can guarantee that the threshold has not been violated (i.e., $f(\mathbf{v}) \leq \tau$).

The problem, of course, is that the $\Delta\mathbf{v}(p_i)$'s are spread across the sites and, thus, the above condition cannot be checked locally. To transform the global condition into a local constraint, we place a d-dimensional *bounding ball* $B(\mathbf{e}, \Delta\mathbf{v}(p_i))$ around each local delta vector, of radius $\frac{1}{2}\|\Delta\mathbf{v}(p_i)\|$ and centered at $\mathbf{e} + \frac{1}{2}\Delta\mathbf{v}(p_i)$ (see Fig. 3). It can be shown that the union of these balls completely covers the convex hull of the drift vectors [33]. This observation effectively reduces the problem of monitoring the global statistics vector to the local problem of each remote site monitoring the ball around its local delta vector.

More specifically, given the monitored function f and threshold τ, we can partition the d-dimensional space to two regions $V = \{\mathbf{v} : f(\mathbf{v}) > \tau\}$ and $\overline{V} = \{\mathbf{v} : f(\mathbf{v}) \leq \tau\}$. (Note that each of these can be arbitrarily complex, e.g., they may comprise multiple disjoint regions of $\mathbb{R}^d$.) The basic protocol is now quite simple: Each site monitors its delta vector $\Delta\mathbf{v}(p_i)$ and, with each update, checks whether its bounding ball $B(\mathbf{e}, \Delta\mathbf{v}(p_i))$ is *monochromatic*, i.e., all points in the ball lie within the same region (either V, or $\overline{V}$). If this is not the case, we have a *local threshold violation*, and the site communicates its local $\Delta\mathbf{v}(p_i)$ to the coordinator. The coordinator then initiates a *synchronization process* that typically tries to resolve the local violation by communicating with only a subset of the sites in order to "balance out" the violating $\Delta\mathbf{v}(p_i)$ and ensure the monochromicity of all local bounding balls [33]. Briefly, this process involves collecting the current delta vectors from (a subset of) the sites, and recomputing the minimum and maximum values of $f(\mathbf{v})$ according to the new, partial, average. If both values reside at the same side of the threshold, the coordinator computes a slack vector for each site in the synchronization set that shifts the local vector to the partial average. In the worst case, the delta vectors from all N sites are collected, leading to an accurate estimate of the current global statistics vector, which is by definition monochromatic (since all bounding balls have 0 radius).

In more recent work, Sharfman et al. [27] show that the local bounding balls defined by the geometric method are actually special cases of a more general theory of *Safe Zones (SZs)*, which can be broadly defined as *convex subsets of the admissible region* of a threshold-crossing query. Then, as long as the local drift vectors stay within such a SZ, the global vector is guaranteed (by convexity) to be within the admissible region of the query.

6.2 ECM-Sketches and the Geometric Method

We are interested in domains where the local and global statistics vectors ($\mathbf{v}(p_i)$ and $\mathbf{v}$ respectively) are defined over a user-chosen sliding window range, and are expected to be high-dimensional, e.g., they may contain the frequency of each item within the user-defined sliding window, for a large number of items. Clearly, accurate maintenance of these statistics for high-velocity data streams is computationally challenging. Furthermore, the aggregation of the local statistics vectors in order to compute the global statistics vector is costly, since it requires exchanging large vectors during synchronization. Both computational and network cost can be substantially reduced with a small trade-off on quality, by using ECM-sketches. This requires the following modifications in the geometric method: a) sites use ECM-sketches to approximate their local statistics vectors, b) the global statistics/estimate vector, the local delta vectors and the drift vectors, are all represented as Count-min sketches, extracted by the ECM-sketches (at query time), and finally, c) during configuration of the geometric method, the query is described on top of the sketch representations of the local and global statistics vector.

Clearly, a naive implementation of the above changes would be subject to substantial constraints, since the size of the domain space of geometric monitoring would be equal to the dimensionality of the ECM-sketches ($d \times w$), and the geometric method is known to be inefficient in high-dimensional domains. It is therefore imperative to reduce the dimensionality of the problems to monitor. In the following sections we show how this is achieved for the frequent items query and for the self-join size queries.

Before going into further details, notice that the above method enables concurrent monitoring of multiple queries (not necessarily of the same type) with a single ECM-sketch per node, which satisfies the strictest accuracy requirements

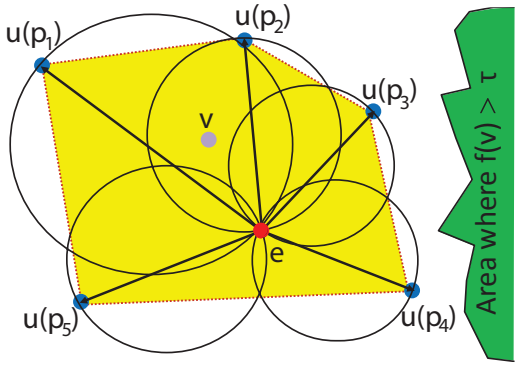

Fig. 3 Estimate vector $\mathbf{e}$, $\Delta\mathbf{v}(p_i)$ (arrows out of $\mathbf{e}$), drift vectors $\mathbf{u}$, convex hull enclosing the current global vector $\mathbf{v}$ (dotted outline), and bounding balls $B(\mathbf{e}, \Delta\mathbf{v}(p_i))$.

of all queries and covers the largest window. Multiple instances of the geometric method (in the simple case, one per query), could then be executed in parallel, coordinating the synchronization process to reduce network cost. The network cost for monitoring the queries is determined by the network requirements of the geometric method, i.e., it depends mainly on the stability of the answer and the acceptable error parameter θ. Fully analyzing all query types, examining the involved challenges, and exploiting the parallel execution of the queries for network and computational benefits is an interesting open problem, and part of our future work.

6.3 Monitoring Frequent Items

Let $p_1, p_2, \ldots, p_n$ denote all n network nodes, $S_1, S_2, \ldots, S_n$ their corresponding streams, and S_0 the order-preserving union of these streams. We use $\mathcal{D}$ to denote the domain of S_0, i.e., all distinct items appearing in the stream, and $S_{i,r}$ the sub-stream of S_i within query range r. The algorithm addresses the problem of distributed continuous monitoring of the set $\mathcal{F}_\tau^r$ of items with frequency in $S_{0,r}$ greater than a user-chosen threshold τ. It works by decomposing the problem to a set of smaller individual problems, one for each distinct item occurring in the stream, yet without requiring the knowledge of all distinct items a priori.

The user first selects the frequency threshold τ, the desired accuracy of ECM-sketches (δ and ϵ), and an acceptable error parameter $\theta \geq 0$ that defines the error tolerance of the geometric method algorithm, i.e., it is acceptable for the algorithm to misclassify items with frequency in the range of $\tau(1 \pm \theta)$. At initialization time t_0, each site p_i constructs an empty ECM-sketch ECM_i to be used as its local statistics vector, and an empty Count-min sketch $CM(t_0)$ to be used as the reference vector (we drop the site id from the notation since the reference vector is always identical at all sites). Both sketches are of the same size $d \times w$, and are initialized with identical hash functions at all sites. After initialization, sites enter the monitoring phase: For each item $x \in \mathcal{D}$, we define a d-dimensional threshold-crossing query as the boolean condition:

$$Q(\boldsymbol{f}, \mathbf{v}, x, \tau, \theta) \equiv \begin{cases} \boldsymbol{f}(\mathbf{v}(t,x)) \geq \tau(1-\theta) \text{ if } \boldsymbol{f}(\mathbf{v}(t_0,x)) < \tau \\ \boldsymbol{f}(\mathbf{v}(t,x)) < \tau(1+\theta) \text{ if } \boldsymbol{f}(\mathbf{v}(t_0,x)) \geq \tau \end{cases}$$

with function vector $\boldsymbol{f} : \mathbb{R}^d \rightarrow \mathbb{R}$ defined as $\boldsymbol{f}(\mathbf{v}) = n \times \min_{j=1}^{d} \mathbf{v}[j]$. The d-dimensional vectors $\mathbf{v}(t,x)$ and $\mathbf{v}(t_0,x)$ are extracted by ECM_i and CM respectively, as follows. $\mathbf{v}(t)[j] = E_i(j, h_j(x), r)$ (the estimation from the counter of ECM_i at position $(j, h_j(x))$) and $\mathbf{v}(t_0)[j] = CM[j, h_j(x)]$ (the value of the counter of the reference Count-min sketch at position $(j, h_j(x))$).

Using the geometric method, sites monitor the threshold-crossing queries in order to detect item arrivals or expirations that potentially invalidate the set of estimated frequent items $\hat{\mathcal{F}}_\tau^r$. An arrival of any item x is handled as follows. First, the local ECM-sketch is updated to include the arrival. If x is already frequent, nothing else needs to be done. In the opposite case, the site probes the corresponding threshold query $Q(\boldsymbol{f}, \mathbf{v}, x, \tau, \theta)$, initiating a synchronization if threshold crossing occurs. Notice that, for synchronization, the coordinator needs to collect only the values of the ECM-sketch counters corresponding to x, i.e., $E(j, h_j(x), r)$ for $j = 1 \ldots d$, in order to update the reference Count-min sketch and decide whether the item causing the violation is frequent. The actual sliding window structures do not need to be exchanged.

Counter updates due to expirations are slightly more complicated (these could cause the removal of a frequent item from $\hat{\mathcal{F}}_\tau^r$). The technical challenge comes from the fact that a bucket expiration at the sliding window of any counter from the ECM-sketch may affect many items, introducing computational complexity. One approach would be to have each site execute the threshold crossing queries for all frequent items at regular intervals. To reduce computational complexity, each site p_i instead maintains a balanced binary search tree that contains all counters of ECM_i and the set of frequent items corresponding to each counter, ordered by the expiration time of their oldest bucket. This tree enables p_i to quickly detect (in constant time) whether any of the counters of ECM_i contains expired buckets, and test only the relevant threshold-crossing queries. The quality guarantees and memory footprint of the above algorithm are summarized by the following lemma.

Lemma 1 *The algorithm guarantees that with probability greater than or equal to* $1 - \delta$*, any item* x *contained in* $\hat{\mathcal{F}}_\tau^r$ *has a real frequency in* $S_{0,r}$ *greater than* $(1-\theta)\tau - \epsilon||S_{0,r}||_1$*, whereas any item* x *not contained in* $\hat{\mathcal{F}}_\tau^r$ *has a real frequency less than* $(1+\theta)\tau + \epsilon||S_{0,r}||_1$*. The algorithm requires memory of* $O(\frac{1}{\epsilon^2}\ln(\frac{1}{\delta})\ln^2(||r||_1) + |\hat{\mathcal{F}}_\tau^r|)$.

6.4 Monitoring Self-Join Size Queries

In the previous case, the problem to be monitored was always d-dimensional, with a small d ($d \leq 5$ for $\delta \geq 0.01$). As such, the geometric method was able to bound the convex hull using relatively small balls, effectively filtering out local updates. Furthermore, threshold violations could be resolved by exchanging only d counters. Estimation of the self-join size, however, involves all $d \times w$ counters, with $(d \times w)$ typically in the hundreds. A naive application of the geometric method for self-join size monitoring would therefore require exchanging $d \times w$ counters at each threshold violation. The problem is further aggravated by the high dimensionality of the bounding balls (equal to the number of counters), which increases the frequency of threshold crossings.

Our attack to this problem is twofold. First, we adapt a recently-proposed insight [18] that enables us to reduce the problem to d dimensions, by monitoring upper and lower bounds of the self-join size estimate instead. This adaptation includes repeating the analysis of [18] for the ECM-sketch (monitoring the min instead of the median, and providing error guarantees relevant to the stream length). However, the bounds offered by this method alone turn up to be quite loose when it comes to ECM-sketches, causing frequent threshold crossings. Therefore, we offer a new, second, approach that further tightens these bounds by exploiting the sliding-window property of ECM-sketches. Compared to the first approach, the second approach drastically reduces the number of threshold crossing, enabling substantial network gains.

We initiate the discussion with some basic notation. Let r denote the query range, and $\mathbf{v}_i(t)$ the Count-min sketch extracted by the ECM-sketch of node p_i, with each counter computed as $\mathbf{v}_i(t)[row, col] = E_i(row, col, r)$.[1] Also, $\mathbf{v}(t) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{v}_i(t)$ (the average of $\mathbf{v}_i(t)$). Function $\boldsymbol{f}(\mathbf{v})$ corresponds to the self-join size estimate function with Count-min sketches, i.e., $\boldsymbol{f}(\mathbf{v}) = \min_{row=1}^{d} \sum_{col=1}^{w} (\mathbf{v}[row, col])^2$. Finally, $\boldsymbol{d}_i$ is the d-dimensional vector computed as $\boldsymbol{d}_i[row] = ||\mathbf{v}_i(t)[row] - \mathbf{v}_i(t_0)[row]||$, and $\boldsymbol{d} = \frac{1}{n}\sum_{i=1}^{n} \boldsymbol{d}_i$. The following lemma enables us to extract d-dimensional threshold-crossing queries:

Lemma 2 *If* $\min_{row=1}^{d}\{\frac{||\mathbf{v}(t_0)[row]||}{n} - \boldsymbol{d}[row]\} \geq \frac{1}{n}\sqrt{\frac{\boldsymbol{f}(\mathbf{v}(t_0))}{1+\theta}}$ *and* $\min_{row=1}^{d}\{\frac{||\mathbf{v}(t_0)[row]||}{n} + \boldsymbol{d}[row]\} \leq \frac{1}{n}\sqrt{\frac{\boldsymbol{f}(\mathbf{v}(t_0))}{1-\theta}}$, *then* $\boldsymbol{f}(\mathbf{v}(t_0)) \in (1 \pm \theta)\boldsymbol{f}(\mathbf{v}(t))$.

Proof In the appendix.

[1] The geometric method is trivially extended to handle matrices instead of vectors by applying vectorization on the matrices, and adjusting the monitored function to use the corresponding vector dimensions. We use the matrix notation for the sketches only for convenience.

Since $\boldsymbol{d}$ is a convex combination (the average) of the distributed values $\boldsymbol{d}_j$, we can already exploit the geometric method to monitor the self-join size estimate. This can be achieved by defining two queries, the first (Q_u) for upper-bounding $\min_{i=1}^{d}\{||\mathbf{v}(t_0)[i]|| - n\boldsymbol{d}[i]\}$, and the second ($Q_l$) for lower-bounding $\min_{i=1}^{d}\{||\mathbf{v}(t_0)[i]|| + n\boldsymbol{d}[i]\}$. A key observation, however, is that the definition of $\boldsymbol{d}$ does not account for the direction of each update: any update of a counter on a local ECM-sketch that shifts a counter away from its last synchronized value (either decreasing the counter value due to an expiration of a bucket, or increasing the value due to a new arrival) will lead to an increase of $\boldsymbol{d}$. This, however, results in unnecessary threshold crossings. For example, an increase of a counter at a peer p_j may lead to a threshold crossing on the lower-bound threshold query, even though in practice an increase of the counter can only lead to an increase of the self-join size.

To circumvent this problem we introduce two auxiliary matrices per peer, one for the upper bound that includes only the counter shifts which increase the counters' values since last synchronization, and another with the shifts that decrease the counters' values. Formally, for the upper bound, $\mathbf{v}_i^u(t)$ is computed as: $\mathbf{v}_i^u(t)[row, col] = \max(\mathbf{v}_i(t), \mathbf{v}_i(t_0))$, and for the lower bound $\mathbf{v}_i^l(t)[row, col] = \min(\mathbf{v}_i(t), \mathbf{v}_i(t_0))$. Similarly, $\boldsymbol{d}_i^u = ||\mathbf{v}_i^u(t)[row] - \mathbf{v}(t_0)[row]||$ and $\boldsymbol{d}_i^l = ||\mathbf{v}_i^l(t)[row] - \mathbf{v}(t_0)[row]||$. $\boldsymbol{d}^u$ and $\boldsymbol{d}^l$ are the corresponding averages over all nodes. Then, we can show the following:

Theorem 5 *If* $\min_{row=1}^{d}\{\frac{||\mathbf{v}(t_0)[row]||}{n} - \boldsymbol{d}^l[row]\} \geq \frac{1}{n}\sqrt{\frac{\boldsymbol{f}(\mathbf{v}(t_0))}{1+\theta}}$ *and* $\min_{row=1}^{d}\{\frac{||\mathbf{v}(t_0)[row]||}{n} + \boldsymbol{d}^u[row]\} \leq \frac{1}{n}\sqrt{\frac{\boldsymbol{f}(\mathbf{v}(t_0))}{1-\theta}}$, *then* $\boldsymbol{f}(\mathbf{v}(t_0)) \in (1 \pm \theta)\boldsymbol{f}(\mathbf{v}(t))$.

Proof In the appendix.

This leads to the following threshold crossing queries (the queries become true when threshold violation occurs):

$$Q_u(\boldsymbol{f}, \boldsymbol{d}^u, \mathbf{v}, \theta) \equiv \min_{row=1}^{d}\{\boldsymbol{d}^u[row] + \frac{||\mathbf{v}(t_0)[row]||}{n}\} > \frac{1}{n}\sqrt{\frac{\boldsymbol{f}(\mathbf{v}(t_0))}{1-\theta}}$$

and for the lower bound:

$$Q_l(\boldsymbol{f}, \boldsymbol{d}^l, \mathbf{v}, \theta) \equiv \min_{row=1}^{d}\{\frac{||\mathbf{v}(t_0)[row]||}{n} - \boldsymbol{d}^l[row]\} < \frac{1}{n}\sqrt{\frac{\boldsymbol{f}(\mathbf{v}(t_0))}{1+\theta}}$$

Synchronization. A two-phase synchronization algorithm is used to handle threshold violations. Without loss of generality we will demonstrate the algorithm assuming a threshold violation in Q_u (the case of Q_l is analogous). In a first phase, all nodes p_i send their local values of $\boldsymbol{d}_i^u$ to the coordinator in order to compute the accurate average value $\boldsymbol{d}^u$. If the updated $\boldsymbol{d}^u$ no longer causes threshold violation, it is sent back to all nodes to be used in the monitoring algorithm. However, if this first phase is not sufficient to address

the threshold violation, the coordinator collects also the local values of $\mathbf{v}_i(t)$, recomputes the average value $\mathbf{v}(t)$ and the updated self-join size, and reinitializes the monitoring algorithm with the updated values. Both phases can be further extended such that synchronization stops as soon as balancing between the retrieved vectors is possible, as explained in Section 6.1. The first phase has a network cost (in transfer volume) of $O(d \times n) = O(n \ln(1/\delta))$, whereas the cost of the (more infrequent) second phase is $O(d \times w \times n) = O(n \ln(1/\delta)/\epsilon)$.

6.5 Efficient Monitoring of the Minimum

The previous discussion has abstracted away the details of the geometric monitoring of functions containing the minimum. In principle, the standard geometric monitoring algorithm can be used, as described above. However, the nature of the monitored function enables substantial optimizations. We distinguish two types of queries: (a) the queries where the last estimate vector is located above the threshold, and (b) the queries where the last estimate vector is below the threshold.

For the first type of queries, we will use as a running example the query Q_l, introduced in Section 6.4 (the same principles apply to the queries introduced in Section 6.3 that correspond to frequent items). The admissible region of the monitored vector in this query is already convex (i.e., in two dimensions, this will be the L-shaped area above the threshold). Hence, the monochromicity test becomes fairly simple: a node p_i reading an update only needs to test whether the local value of $\boldsymbol{d}_i^l$ stays within the convex admissible region, in which case the update is guaranteed to be safe.

Query Q_u, and the queries from Section 6.3 corresponding to infrequent items belong to the second type of queries. For these queries, the admissible region is non-convex. However, the inadmissible region is now convex, and we can apply a different technique based on convex safe zones [27]: In the absense of statistics for the velocity and direction of $\boldsymbol{d}^u$, we choose the safe zone such that it maximizes the slack in all dimensions, as follows. First, we find the point p of the inadmissible region that is closest to $\frac{\mathbf{v}(t_0)}{n}$. It is easy to show that this point is $p[i] = \max\left(\frac{\mathbf{v}(t_0)}{n}, \frac{1}{n}\sqrt{\frac{\boldsymbol{f}(\mathbf{v}(t_0))}{1-\theta}}\right)$. Then, we find the hyperplane H passing from p that is perpendicular to vector $(p - \frac{\mathbf{v}(t_0)}{n})$ (see Fig. 4 for a two-dimensional illustration of this process). Hyperplane H divides the d-dimensional space to two convex subspaces. By construction, the one of the two subspaces (in the two-dimensional example, the subspace in the right of H, denoted with R) contains the inadmissible region and possibly some admissible area, whereas the second subspace (denoted with L) contains only admissible area w.r.t. query Q_u. Since L is convex, it can be used as a safe zone for the geometric method.

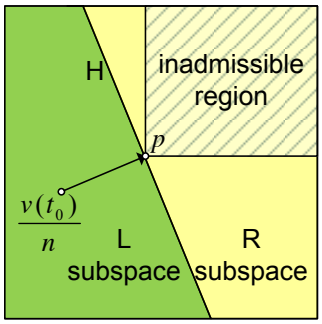

Fig. 4 Monitoring of the minimum for Q_u with safe-zones. The inadmissible region is fully covered by the R subspace (yellow). The L subspace (green) can be used as a safe zone for $\mathbf{v}(t_0)/n + \boldsymbol{d}^u$.

In particular, after each update, nodes only need to check whether $\mathbf{v}(t_0)/n + \boldsymbol{d}^u$ is still within L. This is guaranteed to be the case whenever $\mathbf{v}(t_0)/n + \boldsymbol{d}_i^u$ remains within L for all nodes i. The computational complexity for this process is only $O(d) = O(\ln(1/\delta))$ for computing the hyperplane and checking whether $\boldsymbol{d}_i^u$ is still in the safe zone.

7 Experimental Evaluation

Our experiments focused on evaluating ECM-sketches with respect to their scalability, effectiveness, and efficiency, as well as their suitability for distributed setups. The experiments were conducted using two large real-life data sets, the world-cup'98 [2] (*wc98*) and the CAIDA Anonymized Internet Traces 2011 data set (*caida*[2]). The wc98 data set consists of all HTTP requests that were directed within a period of 92 days to the web-servers hosting the official world-cup 1998 website. It contains a total of 1.089 billion valid requests, served by 33 server mirrors. Each request was indexed using the web-page url as a key, i.e., the ECM-sketch could be used for estimating the popularity of each web-page. The caida data set consists of Internet traces collected by passive monitors installed in Chicago and San Jose. For this experiment we have used the subset of data collected from the Chicago monitors in 17th February 2011, which contained a total of 345 Million IPv4 packets. Each packet was indexed using the source's IP address. Therefore, the ECM-sketch enabled estimating the number of packets sent by each IP address.

We compared three sketch variants, differentiating on the employed sliding window algorithm: (a) the default variant described earlier which is based on exponential histograms, denoted as *ECM-EH*, (b) a variant using deterministic waves (*ECM-DW*), and, (c) a variant based on randomized waves (*ECM-RW*). Comparison between the variants was performed to examine the influence of the sliding window algorithm to the performance of ECM-sketches, in both centralized

[2] Available from `http://www.caida.org/data/`

	ECM-EH	ECM-DW	ECM-RW
wc98			
Update rate	4343095	6067130	70468
Query rate	1641935	1850909	11905
caida			
Update rate	5344982	6205999	72778
Query rate	2377502	3827267	15108

Table 3 Indicative update and query rates (per second) for the centralized setup

ϵ	Data set	Point queries ECM-EH Centr.:Distr.	Inc.rate	Self join ECM-EH Centr.:Distr.	Inc.rate	ECM-RW Centr.:Distr.
0.1	wc98	0.018:0.020	1.114	0.018:0.019	1.053	0.017:0.017
0.2	wc98	0.034:0.040	1.181	0.034:0.037	1.092	0.031:0.031
0.1	caida	0.020:0.021	1.043	0.020:0.020	1.018	0.018:0.018
0.2	caida	0.038:0.041	1.079	0.037:0.039	1.042	0.034:0.034

Table 4 Observed errors and error increase rate – inflation is due to the iterative merging.

and distributed environments. Comparison with ECM-RW was of particular interest, since randomized algorithms for sliding window maintenance (such as the randomized waves employed by ECM-RW) were the only ones supporting merging prior to this work. Therefore, examining the performance of ECM-RW experimentally also serves the purpose of examining the importance of the merging mechanism for deterministic sliding window algorithms, proposed in this work.

7.1 Implementation Details

ECM-sketches were implemented in Java 1.7 using 32-bit addressing. The timing experiments were executed on a single idle core of an Intel Xeon E5-2450, clocked at 2.1 GHz. For the wc98 data set, deterministic and randomized waves were initialized with an upper bound of 1000 events per second, whereas for the caida dataset we have used an upper bound of 1000 events per millisecond. In practice, it is rarely possible to predict the maximum number of events per sliding window, and therefore such estimates (typically decided by analyzing a small stream sample) are often the only option. Exponential histograms did not require such knowledge at initialization time.

Particularly for randomized waves, Gibbons and Tirthapura [21,20] explain that a correctness probability of $1-\delta_{sw}$ requires the parallel maintenance of $c\ln_2(1/\delta_{sw})$ independent instances of the data structure, where the constant $c = 36$ is determined by worst-case analysis. This number of repetitions, in combination to the space complexity of each instance (c/ϵ_{sw}^2), can make the exchange of randomized waves over a distributed network extremely inefficient – hence the importance of the ability to merge deterministic data structures that is proposed in this article. Notice however that, as suggested in [20], smaller constants may also be used in practice in order to reduce the space and computational complexity. This, of course, comes at the cost of meaningless worst-case guarantees. In the following experiments, we set $c = \sqrt{36} = 6$, which reduces the cost by a factor of 36, but still offers an empirical estimation accuracy that is comparable to the one of deterministic sliding window structures configured with the same ϵ.

Unless otherwise mentioned, ECM-sketches were set to monitor a sliding window of 2 million time units. For wc98, this corresponded to 2 million seconds, i.e., 23 days, whereas for caida, it corresponded to 2 million microseconds, i.e., 33 minutes. Queries smaller than the sliding window were executed as well, using the same ECM-sketches. In particular, queries were generated with an exponentially increasing range, i.e., query q_i covered the range $[t-10^i, t]$, with t denoting the time of the last arrival. For each range, a self-join size query, as well as a set of point queries were constructed and executed. For thorough evaluation, we constructed one point query for each distinct item in the query range (i.e., estimating the popularity of each web-page in the wc98 dataset, or the number of packets sent by each IP address in the caida dataset).

7.2 Centralized Setup

In the centralized scenario, a single site monitors the whole stream and maintains an ECM-sketch, which is subsequently used for answering the queries. We first consider the trade-off between memory requirements and estimation error. For this, we vary ϵ within the range of $[0.05, 0.3]$, keeping $\delta = 0.15$. For each ϵ value, we use the analysis presented in Section 4 for point and self-join size queries to configure the ECM-sketch, such that the required memory for the targeted query type is minimized.

Figures 5(a)-(d) plot the average and maximum observed error in correlation to the required memory for the two data sets. The plots are annotated with indicative ϵ values. The displayed error at the Y axis is relative to the number of events arriving within the query range, i.e., for point queries, $\text{err} = |\hat{f}(x,r) - f(x,r)|/||a_r||_1$ and for self-joins, $\text{err} = |\widehat{a_r \odot a_r} - a_r \odot a_r|/(||a_r||_1)^2$. Recall that ECM-RW does not allow probabilistic guarantees for self-join size queries, and is therefore not considered for this type of queries. Table 3 presents indicative update and point query rates for the considered sketches.

We first observe that both the average and maximum observed errors are lower than the user-selected value ϵ for all ECM-sketch variants. However, the memory requirements of ECM-RW are typically two to three orders of magnitude higher than the requirements of ECM-sketches based on the two deterministic structures configured with the same ϵ. As an example, for the wc98 experiment with a moderate value

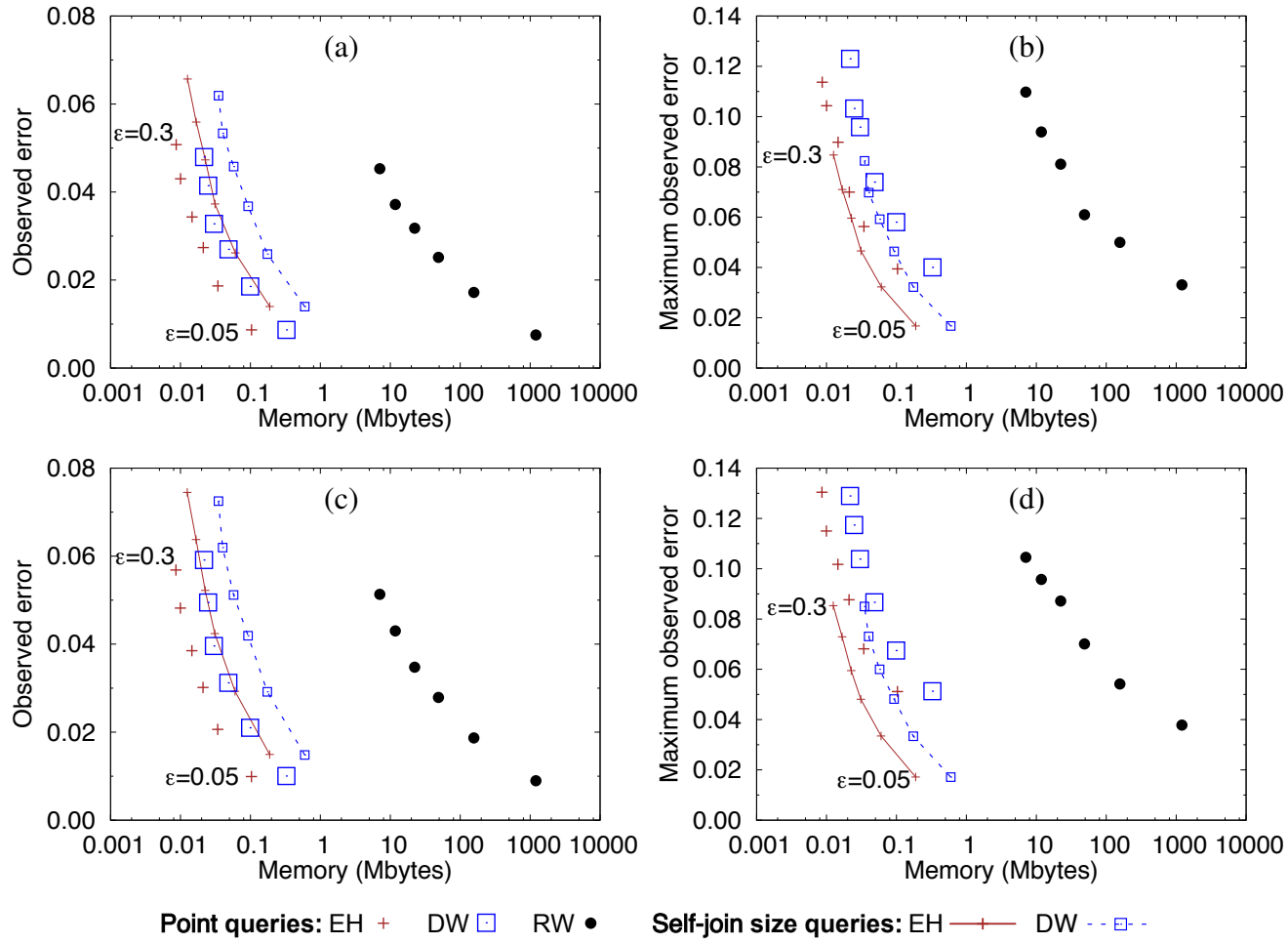

Fig. 5 Average and maximum observed error in correlation to memory requirements for a centralized setup: (a)-(b) wc98 data set, (c)-(d) caida data set. The points correspond to $\epsilon \in \{0.05, 0.1, 0.15, 0.2, 0.25, 0.3\}$.

of $\epsilon = 0.15$, the cost of maintaining the ECM-RW sketch is 48.8 Mbytes, whereas ECM-sketches based on exponential histograms and deterministic waves require 22 Kbytes and 50 Kbytes respectively. This happens because the memory requirements of randomized waves grow quadratically with $1/\epsilon$, whereas the two deterministic sliding window algorithms scale linearly. Note that this negative result applies to all known randomized sliding window algorithms, e.g., [35, 14], since they all scale quadratically with $1/\epsilon$. As such, ECM-sketches based on deterministic structures are more applicable for scenarios with hardware with less memory, like sensor networks and network devices. Also note that ECM-RW are substantially slower than ECM-EH and ECM-DW, supporting two orders of magnitude lower update and query rates (cf. Table 3).

Focusing on the two structures with deterministic sliding windows, we see that ECM-EH sketches are substantially more compact, requiring around one third of the space of ECM-DW for the same ϵ value. Concerning computational performance, both structures can support comparable update and query execution rates (ECM-DW is slightly faster than ECM-EH, mainly due to its $O(d)$ worst-case complexity per update, compared to $O(d \log N)$ for ECM-EH). The results are consistent for both data sets.

Summarizing, these first results demonstrate the superiority of ECM-EH and ECM-DW compared to ECM-RW, both in terms of compactness and computational performance. ECM-EH and ECM-DW have comparable computational performance, whereas in terms of compactness ECM-EH substantially outperforms ECM-DW.

7.3 Distributed Setup

The second series of experiments was designed to evaluate the suitability of ECM-sketches for distributed setups, and precisely: (a) for setups requiring one-time merging of ECM-sketches, possibly even in a hierarchical fashion, and

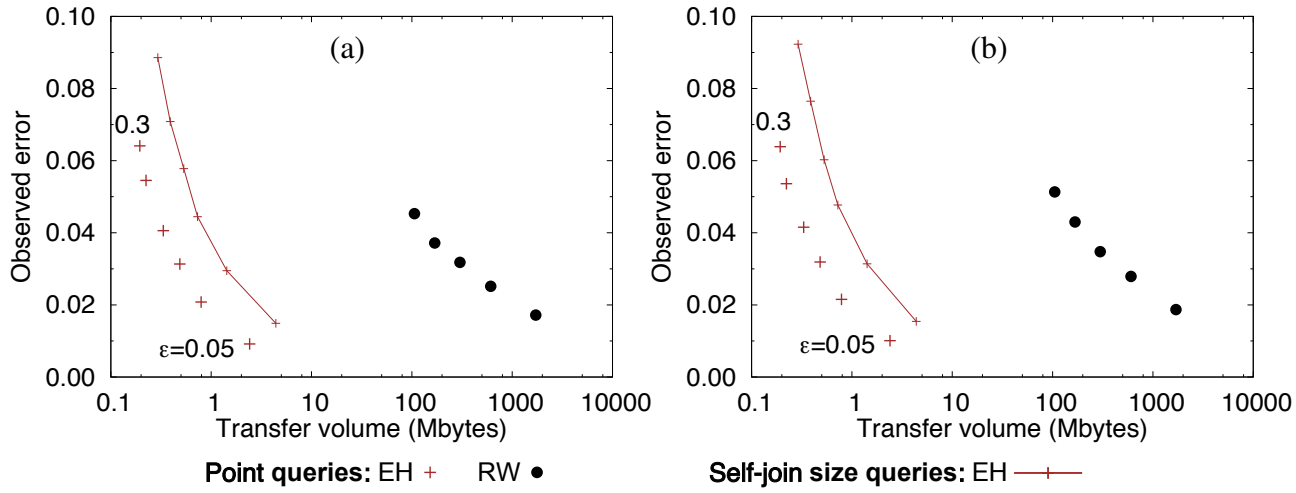

Fig. 6 Observed error in correlation to the network cost, for varying ϵ: (a) wc98 data set, (b) caida data set.

(b) for setups requiring continuous monitoring of functions through distributed ECM-sketches.

7.3.1 One-time Merging

These experiments focused on studying the influence of the network size and ϵ on the network cost. We have simulated a fixed network of $n \in [2, 256]$ sites, organized in an architecture resembling a balanced binary tree of height $\lceil \log_2(n) \rceil$. All sites resided at the leaf nodes of the tree, and were assigned the task of summarizing disjoint streams with ECM-sketches. Some of the sites were also randomly placed at the internal tree nodes, and were responsible for merging the sketches coming from their children. After completion of the streams, the sites pushed the resulting ECM-sketches to the root through the hierarchy, with merging at each intermediary node. At the end of this process, the root node of the hierarchy was holding a single ECM-sketch that represented the order-preserving merging of the n streams. ECM-DW sketches are not considered in this set of experiments, since they do not offer advantages compared to ECM-EH sketches with respect to compactness or accuracy.

Figures 6(a)-(b) plot the average observed error for point and self-join size queries in correlation to the network requirements for the whole merging process to be completed. The results correspond to a fixed network of 16 sites, with $\epsilon \in [0.05, 0.3]$ and $\delta = 0.15$. (Note that the simulation with ECM-RW sketches did not complete for $\epsilon = 0.05$ values, due to insufficient memory resources at the machine simulating the sites.) To illustrate the accuracy loss due to this merging, Table 4 presents a comparison between the observed error of the centralized and the distributed ECM-sketches.

As expected, the process of iterative mergings causes an inflation of the observed error for ECM-EH sketches. This inflation, however, is very small, and substantially lower than the theoretical worst-case bound derived by the analysis. For example, for the wc98 dataset with $\epsilon = 0.1$, the average observed error after all mergings is 0.020, whereas the centralized ECM-EH has an observed error of 0.018, i.e., the error inflation caused by the iterative ECM-EH mergings is less than 1/8 of the experimentally derived error of the centralized sketch. Concerning ECM-RW sketches, there is no systemic variation of the error, since randomized waves enable lossless merging at the expense of a larger memory footprint. However, the network required for performing this merging using ECM-RW is at least three orders of magnitude higher. This requirement is prohibitive for a large set of application scenarios, like sensor and mobile networks, where high network usage causes severe battery drainage.

To explore the influence of the network size on the estimation accuracy and network cost, we have also simulated networks of n sites, with $n = \{2, 4, \ldots, 256\}$. (For the case of ECM-RW, the number of sites reached only up to 64 due to memory constraints at the machine executing the simulation.) The sites were again placed as leaf nodes on a balanced binary tree, and updates were assigned to the sites randomly, with equal probability. Figures 7(a) and (c) plot the average observed error in correlation to the network size, for $\epsilon = \delta = 0.15$. As expected, for ECM-EH sketches, increasing the number of sites leads to a small increase on the observed estimation error, whereas the accuracy of ECM-RW sketches remains unaffected. However, similar to the previous experiment, the network cost for merging the sketches based on randomized waves (Figures 7(b) and (d)) is three orders of magnitude higher compared to ECM-EH. This limits the applicability of ECM-RW to cases where fast, fixed network is available, and makes the ability to merge deterministic sliding windows, e.g., based on exponential histograms, a very important contribution of this work.

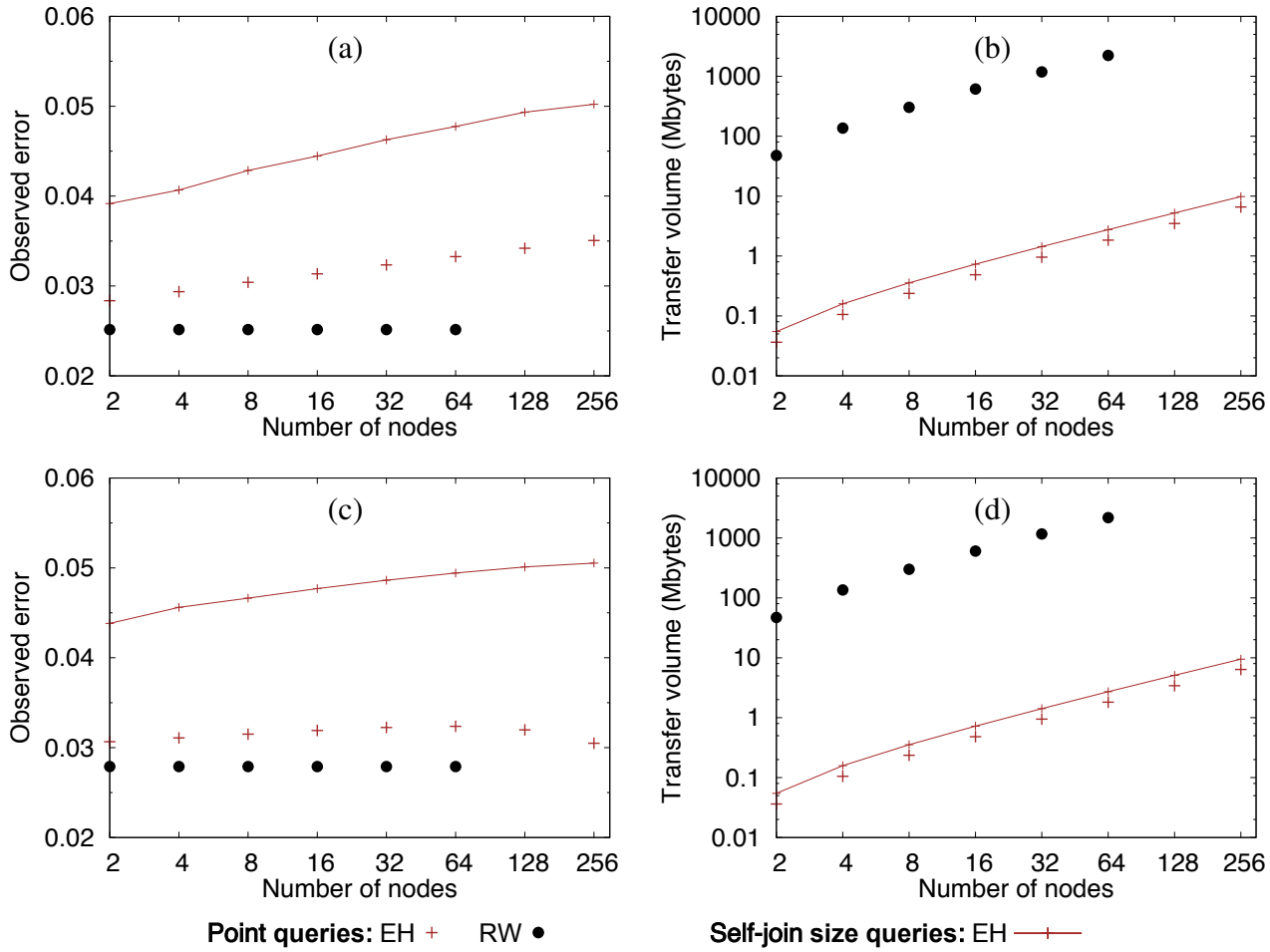

Fig. 7 Observed error and network cost for different network sizes: (a)-(b) wc98, (c)-(d) caida.

Summarizing, this set of experiments showed that ECM-sketches based on exponential histograms can be merged with very small information loss. Compared to the lossless merging of ECM-sketches based on randomized waves, ECM-EH are substantially more compact, and are therefore applicable for a wider range of application scenarios, where network cost and/or memory is of the essence, such as P2P networks, sensor networks, and network routers.

7.3.2 Continuous Monitoring

The final set of experiments investigates the suitability of ECM-sketches in combination with the geometric method for distributed continuous monitoring, as discussed in Section 6 (denoted as $\mathcal{A}_{ecm}$ hereafter). Particularly, we consider monitoring of the self-join size of a high-dimensional vector $\mathbf{v}$ that corresponds to sliding window statistics (i.e., item frequencies) aggregated over n data streams $S_1, S_2, \ldots, S_n$. Each stream S_i is monitored by site p_i, and all sites are enabled direct communication with a coordinator. Estimating the self-join size of such high-dimensional vectors is frequently useful, e.g., for query optimization in distributed databases, data partitioning, and computing a variety of useful indexes for streams (see [1] for a discussion). We only consider ECM-sketches constructed with exponential histograms, since these offer the best trade-off between memory and accuracy. As a baseline, we use the centralized algorithm (denoted as $\mathcal{A}_{cen}$), which relies on a central coordinator for collecting all updates from the remote sites and maintaining the accurate self-join size. Notice that $\mathcal{A}_{cen}$ has several practical considerations besides the high network cost, i.e., the coordinator still needs to efficiently maintain the high-dimensional statistics over a sliding window, which is challenging to achieve without ECM-sketches. Yet, we ignore this issue for our experiments. Both algorithms were allowed a warm-up phase (until the sliding window filled up for first time) before starting to measure cost and quality.

Figure 8(a) presents the transfer volume required by $\mathcal{A}_{ecm}$ for different network sizes as a ratio of the corresponding transfer volume of $\mathcal{A}_{cen}$. The results correspond to a configuration of $\mathcal{A}_{ecm}$ with $\delta = \epsilon = \theta = 0.15$. Clearly, $\mathcal{A}_{ecm}$ is substantially more efficient than the baseline, enabling net-

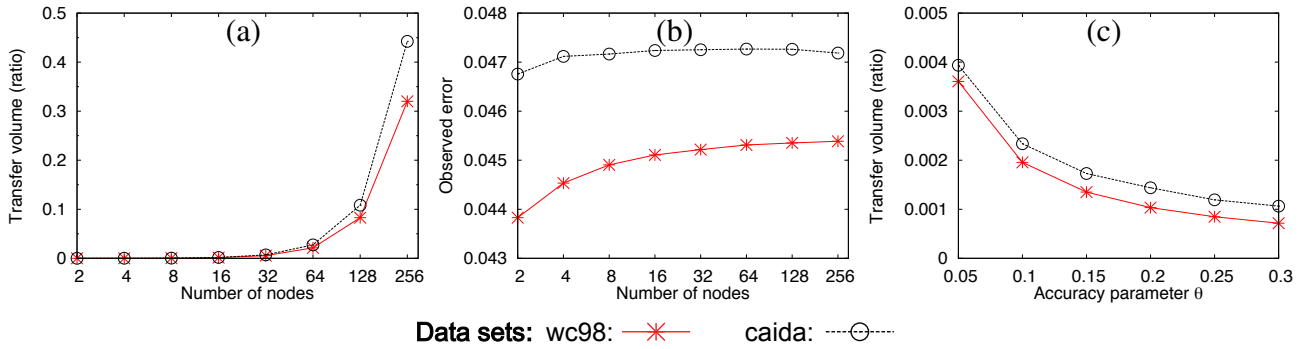

Fig. 8 (a)-(b) Network cost and observed error for different network sizes, (c) Effect of the value of θ.

work savings of up to two orders of magnitude for networks up to 32 sites. As expected, increasing the network size leads to an increase of the communication cost of $\mathcal{A}_{ecm}$ (the cost of $\mathcal{A}_{cen}$ does not change). This is a known characteristic of the geometric method. Nevertheless, even for the network of 256 sites, $\mathcal{A}_{ecm}$ still requires less than half the cost of $\mathcal{A}_{cen}$. We also see that caida is slightly more difficult to monitor compared to wc98. This is because wc98 is more stable than caida, i.e., as soon as the sliding window is filled, self-join size changes very slowly. Caida data set, on the other hand, is by nature more dynamic, causing more frequent threshold violations, and a higher network cost.

The average observed error for the same runs is shown in Figure 8(b). Even though error slightly increases with network size, the increase is negligible, and the error always remains smaller than the value of parameter θ, i.e., the error tolerance of the geometric method. All results are consistent for both data sets.

We have also tested the sensitivity of $\mathcal{A}_{ecm}$ on parameter θ. The results in Figure 8(c) correspond to a fixed network of 16 sites, with δ and ϵ set to 0.15. As expected, increasing θ drastically reduces the network cost of the algorithm: for a higher θ, $\mathcal{A}_{ecm}$ causes less threshold crossings, requiring less synchronizations of both phases. As an indication, for the caida dataset and for $\theta = 0.05$, $\mathcal{A}_{ecm}$ required 7133 first-phase synchronizations (i.e., synchronizations on $d_u|d_l$ only) and 2694 second-phase synchronizations (on the full sketches). For $\theta = 0.3$, these synchronizations were reduced to 5637 for the first phase, and only 457 for the second phase.

Summarizing, the experiments have shown that the combination of ECM-sketches with the geometric method can be used for efficiently monitoring of non-linear functions, such as the self-join size, in distributed settings. The network savings are substantial compared to the baseline algorithm that forwards all updates to a central coordinator, and typically exceed two orders of magnitude for small networks, whereas the observed error is negligible.

8 Conclusions

In this work we considered the problem of answering complex queries over distributed and high dimensional data streams, in the sliding window model. Our proposal, ECM-sketches, is a compact structure combining the state-of-the-art sketching technique for data stream summarization with deterministic sliding window synopses. The structure provides probabilistic accuracy guarantees for the quality of the estimation, for point queries and self-join size queries, and can enable a broad range of problems, such as finding heavy hitters, computing quantiles, and answering range queries over sliding windows.

Focusing on distributed applications, we also showed how a set of ECM-sketches, each one representing an individual stream, can be merged to generate a single ECM-sketch that summarizes the stream produced by the order-sensitive merging of all individual streams. Interestingly, this is the first result in the literature enabling such merging for deterministic sliding window synopses (or sketches based on these), and it is of high importance since deterministic synopses are generally a factor of $O(\log(1/\delta)/\epsilon)$ more compact than the best-known randomized synopsis for delivering an ϵ-accurate approximation. In the same context, we demonstrated how ECM-sketches can be exploited within the geometric method for answering continuous queries defined over sliding windows.

ECM-sketches were thoroughly evaluated with a set of extensive experiments, using two massive real-world datasets, and considering both centralized and distributed setups. The results verified the high performance of the structure. Compared to structures based on randomized sliding window synopses, ECM-sketches improve the memory and computational complexity by at least one order of magnitude. The same magnitude of improvement is observed with respect to the network requirements.

Our future work will focus on further optimizations for continuous distributed queries. Two interesting open problems include considering other query types, and concurrently

executing multiple continuous queries. In Section 6 we have already discussed initial optimizations for concurrent execution of many queries. We expect that both computation and network complexity can be reduced further by coordinating the synchronization process between the queries, and taking the accuracy requirements of each query into account during the synchronization process.

Acknowledgments. This work was supported by the European Commission under ICT-FP7-LEADS-318809 (Large-Scale Elastic Architecture for Data-as-a-Service) and ICT-FP7-FERARI-619491 (Flexible Event pRocessing for big dAta aRchItectures).

References

1. Alon, N., Matias, Y., Szegedy, M.: The space complexity of approximating the frequency moments. J. Comput. Syst. Sci. **58**(1), 137–147 (1999)
2. Arlitt, M., Jin, T.: A workload characterization study of the 1998 world cup web site. Network **14**(3), 30 –37 (2000)
3. Busch, C., Tirthapura, S.: A deterministic algorithm for summarizing asynchronous streams over a sliding window. In: STACS, pp. 465–476 (2007)
4. Chakrabarti, A., Cormode, G., Mcgregor, A.: A near-optimal algorithm for estimating the entropy of a stream. ACM Trans. Algorithms **6**(3), 51:1–51:21 (2010)
5. Chan, H.L., Lam, T.W., Lee, L.K., Ting, H.F.: Continuous monitoring of distributed data streams over a time-based sliding window. Algorithmica **62**(3-4), 1088–1111 (2012)
6. Charikar, M., Chen, K., Farach-Colton, M.: Finding frequent items in data streams. In: ICALP, pp. 693–703 (2002)
7. Cohen, E., Strauss, M.J.: Maintaining time-decaying stream aggregates. J. Algorithms **59**(1), 19–36 (2006)
8. Cormode, G., Garofalakis, M.: Approximate continuous querying over distributed streams. ACM Trans. Database Syst. **33**(2) (2008)
9. Cormode, G., Garofalakis, M., Muthukrishnan, S., Rastogi, R.: Holistic aggregates in a networked world: Distributed tracking of approximate quantiles. In: SIGMOD, pp. 25–36 (2005)
10. Cormode, G., Muthukrishnan, S.: What's hot and what's not: Tracking most frequent items dynamically. In: PODS, pp. 296–306 (2003)
11. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. J. Algorithms **55**(1), 58–75 (2005)
12. Cormode, G., Muthukrishnan, S., Yi, K., Zhang, Q.: Optimal sampling from distributed streams. In: PODS, pp. 77–86 (2010)
13. Cormode, G., Muthukrishnan, S., Yi, K., Zhang, Q.: Continuous sampling from distributed streams. J. ACM **59**(2), 10:1–10:25 (2012)
14. Cormode, G., Tirthapura, S., Xu, B.: Time-decaying sketches for robust aggregation of sensor data. SIAM J. Comput. **39**(4), 1309–1339 (2009)
15. Cormode, G., Yi, K.: Tracking distributed aggregates over time-based sliding windows. In: SSDBM, pp. 416–430 (2012)
16. Datar, M., Gionis, A., Indyk, P., Motwani, R.: Maintaining stream statistics over sliding windows. SIAM J. Comput. **31**(6), 1794–1813 (2002)
17. Dimitropoulos, X.A., Stoecklin, M.P., Hurley, P., Kind, A.: The eternal sunshine of the sketch data structure. Computer Networks **52**(17), 3248–3257 (2008)
18. Garofalakis, M.N., Keren, D., Samoladas, V.: Sketch-based geometric monitoring of distributed stream queries. PVLDB **6**(10), 937–948 (2013)
19. Gibbons, P.B.: Distinct sampling for highly-accurate answers to distinct values queries and event reports. In: VLDB, pp. 541–550 (2001)
20. Gibbons, P.B.: Distinct-values estimation over data streams. In: Data Stream Management: Processing High-Speed Data Streams. Springer (2007)
21. Gibbons, P.B., Tirthapura, S.: Distributed streams algorithms for sliding windows. In: SPAA, pp. 63–72 (2002)
22. Greenwald, M.B., Khanna, S.: Space-efficient online computation of quantile summaries. In: SIGMOD, pp. 58–66 (2001)
23. Huang, L., Garofalakis, M., Joseph, A., Taft, N.: Communication efficient tracking of distributed cumulative triggers. In: ICDCS (2007)
24. Huang, L., Nguyen, X., Garofalakis, M., Hellerstein, J., Jordan, M., Joseph, A., Taft, N.: Communication-efficient online detection of network-wide anomalies. In: INFOCOM, pp. 134–142 (2007)
25. Hung, R.Y.S., Ting, H.F.: Finding heavy hitters over the sliding window of a weighted data stream. In: LATIN, pp. 699–710 (2008)
26. Jain, A., Hellerstein, J.M., Ratnasamy, S., Wetherall, D.: A wakeup call for internet monitoring systems: The case for distributed triggers. In: SIGCOMM Workshop on Hot Topics in Networks (HotNets) (2004)
27. Keren, D., Sharfman, I., Schuster, A., Livne, A.: Shape sensitive geometric monitoring. TKDE **24**(8), 1520–1535 (2012)
28. Mirkovic, J., Prier, G., Reiher, P.L.: Attacking DDoS at the source. In: ICNP, pp. 312–321 (2002)
29. Muthukrishnan, S.: "Data Streams: Algorithms and Applications". Foundations and Trends in Theoretical Computer Science **1**(2) (2005)
30. Olston, C., Jiang, J., Widom, J.: Adaptive filters for continuous queries over distributed data streams. In: SIGMOD, pp. 563–574 (2003)
31. Papapetrou, O., Garofalakis, M.N., Deligiannakis, A.: Sketch-based querying of distributed sliding-window data streams. PVLDB **5**(10), 992–1003 (2012)
32. Qiao, L., Agrawal, D., El Abbadi, A.: Supporting sliding window queries for continuous data streams. In: SSDBM, pp. 85–96 (2003)
33. Sharfman, I., Schuster, A., Keren, D.: A geometric approach to monitoring threshold functions over distributed data streams. In: SIGMOD, pp. 301–312 (2006)
34. Tirthapura, S., Xu, B., Busch, C.: Sketching asynchronous streams over a sliding window. In: PODC, pp. 82–91 (2006)
35. Xu, B., Tirthapura, S., Busch, C.: Sketching asynchronous data streams over sliding windows. Distributed Computing **20**(5), 359–374 (2008)

Appendix

A Proofs for centralized queries

Lemmas 3 and 4 provide error guarantees for point and inner product queries on ECM-sketches, for any set of ϵ_{cm}, ϵ_{sw}, δ_{cm} and δ_{sw}. With Theorems 2 and 3 we derive the optimal values of these parameters (the ones that minimize the total cost), given only the acceptable total ϵ and δ.

Lemma 3 *With probability at least* $1 - \delta_{cm} - \delta_{sw}$,

$$|\hat{f}(x,r) - f(x,r)| \leq \begin{cases} (1+\epsilon_{sw})\epsilon_{cm}||a_r||_1 & \text{if } \epsilon_{sw} \leq \frac{\epsilon_{cm}}{1-\epsilon_{cm}}, \\ \epsilon_{sw}||a_r||_1 & \text{if } \epsilon_{sw} \geq \frac{\epsilon_{cm}}{1-\epsilon_{cm}}. \end{cases}$$

Proof We start with an overview of the proof. The ECM-sketch estimation is susceptible to two types of errors. The first is due to the hash collisions, i.e., two different items may hash to the same ECM-sketch cell. This error is relative to the L1 norm. The second is due to the sliding window counters, and is relative to the counter value, i.e., the number of items hashed to the particular counter. For the lemma, we derive a single error relative to the L1 norm by considering worst-case bounds (i.e., maximum possible values) for the combination of the two errors. We first bound the number of hash collisions that occur at counter $(j, h_j(x))$ for any row $j \leq d$ within the query range r, assuming that the sliding window algorithm offers perfect accuracy. The proof stems from the accuracy proof for Count-min sketches [11], differentiating on the estimation step of the number of hash collisions (by offering an error relative to $||a_r||_1 - f(x,r)$, instead of $||a_r||_1$). Then, we consider the error caused by the sliding window estimation.

Error due to hash collisions. Temporarily assume that the sliding window structure enables perfect accuracy (the assumption will be lifted later). With $I_{x,j,y}$ we denote the indicator variables which are 1 if $x \neq y \wedge h_j(x) = h_j(y)$, and 0 otherwise. We further define the variables $X_{x,j,r}$ to be $X_{x,j,r} = \sum_{y \in \mathcal{D}} I_{x,j,y} f(y,r)$. By our assumption that the sliding window estimation is accurate, $E(j, h_j(x), r) = f(x,r) + X_{x,j,r}$. Since the ECM-sketch will return $\hat{f}(x,r) = \min_j E(j, h_j(x), r) = f(x,r) + \min_j X_{x,j,r}$ as a frequency estimate, the estimation error will be $\hat{f}(x,r) - f(x,r) = \min_j X_{x,j,r}$, which can be bounded as follows.

By pairwise independence of the hash functions: $E(I_{x,j,y}) = Pr[h_j(x) = h_j(y)] \leq 1/w = \frac{\epsilon}{e}$. Therefore, the expected value of $X_{x,j,r}$ is $E(X_{x,j,r}) = E\left(\sum_{k=1}^{n} I_{x,j,y} f(k,r)\right) = \sum_{\forall k \in \mathcal{D} \setminus \{x\}} f(k,r) E(I_{x,j,y}) \leq (||a_r||_1 - f(x,r))\frac{\epsilon}{e}$. Furthermore, by Markov inequality:

$$
\begin{aligned}
&Pr[\min_j X_{x,j,r} > \epsilon(||a_r||_1 - f(x,r))] = \\
&Pr[\forall j : X_{x,j,r} > \epsilon(||a_r||_1 - f(x,r))] \leq \\
&Pr[\forall j : X_{x,j,r} > eE(X_{x,j,r})] < e^{-d} \leq \delta_{cm} \qquad (1)
\end{aligned}
$$

Error due to the sliding window estimation. In practice, the sliding window algorithm may introduce errors to the computation of $E(j, h_j(x), r)$. Let $R(j, h_j(x), r)$ denote the accurate number of bits contained within the query range at counter $(j, h_j(x))$. Then, an (ϵ, δ)-approximate sliding window algorithm guarantees that $Pr[E(j, h_j(x), r) - R(j, h_j(x), r) \leq \epsilon_{sw} R(j, h_j(x), r)] \geq 1 - \delta_{sw}$.

Consider row $j = \min\arg_j E(j, h_j(x), r)$, i.e., the row with $E(j, h_j(x), r) = \hat{f}(x,r)$. For the case that $\hat{f}(x,r) > R(j, h_j(x), r)$, we have:

$$
\begin{aligned}
&Pr[\hat{f}(x,r) \leq (1+\epsilon_{sw}) R(j, h_j(x), r)] \geq 1 - \delta_{sw} \Rightarrow \\
&Pr[\hat{f}(x,r) \leq (1+\epsilon_{sw})(f(x,r) + X_{x,j,r})] \\
&\qquad \geq 1 - \delta_{sw} \Rightarrow \\
&Pr[\hat{f}(x,r) - f(x,r) \leq X_{x,j,r} + \epsilon_{sw}(f(x,r) + X_{x,j,r}) \\
&\qquad \geq 1 - \delta_{sw}
\end{aligned}
$$

Furthermore, $X_{x,j,r}$ can be bounded by Inequality 1, giving: $Pr[\hat{f}(x,r) - f(x,r) \leq \epsilon_{cm}(||a_r||_1 - f(x,r)) + \epsilon_{sw}(f(x,r) + \epsilon_{cm}(||a_r||_1 - f(x,r)))] \geq 1 - \delta_{sw} - \delta_{cm}$. For convenience we define $c = f(x,r)/||a_r||_1$. Then, $Pr[\hat{f}(x,r) - f(x,r) \leq ||a_r||_1(\epsilon_{cm}(1-c) + \epsilon_{sw}(c + \epsilon_{cm}(1-c)))] = Pr[\hat{f}(x,r) - f(x,r) \leq ||a_r||_1(c(\epsilon_{sw} - \epsilon_{cm} - \epsilon_{sw}\epsilon_{cm}) + \epsilon_{cm} + \epsilon_{sw}\epsilon_{cm})] \geq 1 - \delta_{sw} - \delta_{cm}$.

Variable c takes values between 0 and 1 (inclusive). When $\epsilon_{sw} \leq \epsilon_{cm}/(1-\epsilon_{cm})$, the RHS of the inequality (the error) is maximized for $c = 0$. Otherwise, the RHS is maximized for $c = 1$. Therefore, with a probability of at least $1 - \delta_{cm} - \delta_{sw}$:

$$
\hat{f}(x,r) - f(x,r) \leq \begin{cases} ||a_r||_1 \epsilon_{cm}(1+\epsilon_{sw}) & \text{if } \epsilon_{sw} \leq \frac{\epsilon_{cm}}{1-\epsilon_{cm}}, \\ ||a_r||_1 \epsilon_{sw} & \text{if } \epsilon_{sw} \geq \frac{\epsilon_{cm}}{1-\epsilon_{cm}}. \end{cases} \qquad (2)
$$

With a similar analysis, the case of $\hat{f}(x,r) < R(j, h_j(x), r)$ gives a much tighter constraint:

$$
Pr[f(x,r) - \hat{f}(x,r) \leq \epsilon_{sw} f(x,r)] \geq 1 - \delta_{sw} \qquad (3)
$$

The lemma follows directly by inequalities 2 and 3. □

Theorem 3

Proof We first consider an ECM-sketch with a deterministic sliding window structure, e.g., an exponential histogram. We want to derive the combination of ϵ_{cm} and ϵ_{sw} that minimizes the space complexity of the sketch for a given ϵ, i.e., minimizes $C(\epsilon) = O(\frac{\ln(1/\delta_{cm})}{\epsilon_{cm}\epsilon_{sw}})$. We study the two cases of Lemma 3 separately:

Case 1 ($\epsilon_{sw} \leq \frac{\epsilon_{cm}}{1-\epsilon_{cm}}$): We first exploit the fact that

$$
\epsilon = (1+\epsilon_{sw})\epsilon_{cm} \qquad (4)
$$

to eliminate ϵ_{sw} from the space complexity of the sketch:

$$
C(\epsilon) = O\left(\frac{\ln(1/\delta_{cm})}{\epsilon_{sw}\epsilon_{cm}}\right) = O\left(\frac{\ln(1/\delta_{cm})}{\frac{\epsilon - \epsilon_{cm}}{\epsilon_{cm}}\epsilon_{cm}}\right) = O\left(\frac{\ln(1/\delta_{cm})}{\epsilon - \epsilon_{cm}}\right)
$$

The cost is minimized when the denominator is maximized. For a fixed ϵ chosen by the user, this happens when ϵ_{cm} is minimized. The smallest ϵ_{cm} satisfying Eqn. 4 and the precondition of Case 1 is $\epsilon_{cm} = \frac{\epsilon}{1+\epsilon}$, resulting to $\epsilon_{sw} = \epsilon$.

Case 2 ($\epsilon_{sw} \geq \frac{\epsilon_{cm}}{1-\epsilon_{cm}}$): By Lemma 3, we see that setting $\epsilon_{sw} = \epsilon$ we achieve the required accuracy. In order to derive ϵ_{cm}, notice that we want to minimize the complexity $C(\epsilon) = O(\frac{\ln(1/\delta_{cm})}{\epsilon_{cm}\epsilon_{sw}})$. This is achieved by maximizing ϵ_{cm}. The maximum value of ϵ_{cm} satisfying the precondition of Case 2 is $\epsilon_{cm} = \frac{\epsilon}{1+\epsilon}$.

Notice that both cases lead to the same combination for minimizing the cost, i.e., $\epsilon_{cm} = \frac{\epsilon}{1+\epsilon}$ and $\epsilon_{sw} = \epsilon$.

The case of randomized waves is similar. The cost function becomes $O(\ln(1/\delta_{cm})\ln(1/\delta_{sw})/(\epsilon_{cm}\epsilon_{sw}^2))$, with the constraint that $\delta = \delta_{cm} + \delta_{sw}$. The cost is minimized for $\delta_{cm} = \delta_{sw} = \delta/2$, $\epsilon_{cm} = \frac{\epsilon}{1+\epsilon}$ and $\epsilon_{sw} = \epsilon$. □

Lemma 4 *With probability at least* $1 - \delta_{cm}$,

$$
|\widehat{a_r \odot b_r} - a_r \odot b_r| \leq \begin{cases} ||a_r||_1||b_r||_1\epsilon_{cm}(1+\epsilon_{sw})^2 \\ \qquad \text{if } \epsilon_{cm} \geq \frac{\epsilon_{sw}^2 + 2\epsilon_{sw}}{(\epsilon_{sw}+1)^2}, \\ ||a_r||_1||b_r||_1(\epsilon_{sw}^2 + 2\epsilon_{sw}) \\ \qquad \text{if } \epsilon_{cm} \leq \frac{\epsilon_{sw}^2 + 2\epsilon_{sw}}{(\epsilon_{sw}+1)^2}. \end{cases}
$$

Proof We first examine the case that $\widehat{a_r \odot b_r} > a_r \odot b_r$. Consider the estimation derived by any single row j of the ECM-sketch. With $E_a(i,j,r)$ we denote the frequency estimation of the sliding window counter at position (i,j) for stream a and for query range r.

$$E(\widehat{(a_r \odot b_r)}_j - a_r \odot b_r) = \sum_{i=1}^{w} E_a(i,j,r)E_b(i,j,r) - a_r \odot b_r$$

$$\leq \sum_{i=1}^{w} \sum_{\substack{p\in\mathcal{D},\\ h_j(p)=i}} f_a(p,r) \sum_{\substack{q\in\mathcal{D},\\ h_j(q)=i}} f_b(q,r) * (1+\epsilon_{sw})^2 - a_r \odot b_r$$

$$= \sum_{i=1}^{w} \sum_{\substack{x\in\mathcal{D},\\ h_j(x)=i}} f_a(x,r)f_b(x,r) * (1+\epsilon_{sw})^2 +$$

$$\sum_{i=1}^{w} \sum_{\substack{p,q\in\mathcal{D},p\neq q,\\ h_j(p)=h_j(q)=i}} f_a(p,r)f_b(q,r) * (1+\epsilon_{sw})^2 - a_r \odot b_r$$

$$= (1+\epsilon_{sw})^2 \Big(\sum_{x\in\mathcal{D}} f_a(x,r)f_b(x,r) +$$

$$\sum_{\substack{p,q\in\mathcal{D},p\neq q,\\ h_j(p)=h_j(q)}} f_a(p,r)f_b(q,r)\Big) - a_r \odot b_r$$

$$= a_r \odot b_r(\epsilon_{sw}^2 + 2\epsilon_{sw}) +$$

$$\sum_{\substack{p,q\in\mathcal{D},p\neq q,\\ h_j(p)=h_j(q)}} f_a(p,r)f_b(q,r)(1+\epsilon_{sw})^2 \quad (5)$$

Our next step is to bound $\sum_{\substack{p,q\in\mathcal{D},p\neq q,\\ h_j(p)=h_j(q)}} f_a(p,r)f_b(q,r)$. For convenience we use $X_{i,j,r}$ as a shortcut for $\sum_{\substack{p,q\in\mathcal{D},p\neq q,\\ h_j(p)=h_j(q)}} f_a(p,r)f_b(q,r)$. Then,

$$E(X_{i,j,r}) = \sum_{p,q\in\mathcal{D},p\neq q} Pr[h_j(p)=h_j(q)] f_a(p,r)f_b(q,r)$$

$$= \frac{1}{w} \sum_{p,q\in\mathcal{D},p\neq q} f_a(p,r)f_b(q,r)$$

$$\leq \frac{\epsilon_{cm}}{e}\Big(\sum_{p,q\in\mathcal{D}} f_a(p,r)f_b(q,r) - a_r \odot b_r\Big)$$

$X_{i,j,r}$ can be bounded by Markov inequality:

$$Pr[\min_j X_{i,j,r} > \epsilon_{cm}(\sum_{p,q\in\mathcal{D}} f_a(p,r)f_b(q,r) - a_r \odot b_r)] =$$

$$Pr[\forall j : X_{i,j,r} > eE(X_{i,j,r})] < e^{-d} \leq \delta_{cm} \quad (6)$$

Let $c = \frac{a_r \odot b_r}{||a_r||_1||b_r||_1}$. Combining Equation 5 and Inequality 6:

$$\widehat{a_r \odot b_r} - a_r \odot b_r$$

$$\leq a_r \odot b_r(\epsilon_{sw}^2 + 2\epsilon_{sw}) + \sum_{\substack{p,q\in\mathcal{D},p\neq q,\\ h_j(p)=h_j(q)}} f_a(p,r)f_b(q,r)(1+\epsilon_{sw})^2$$

$$< a_r \odot b_r(\epsilon_{sw}^2 + 2\epsilon_{sw}) + (1+\epsilon_{sw})^2 \min_j X_{i,j,r}$$

$$\leq a_r \odot b_r(\epsilon_{sw}^2 + 2\epsilon_{sw}) +$$

$$(1+\epsilon_{sw})^2\epsilon_{cm}\left(\sum_{p,q\in\mathcal{D}} f_a(p,r)f_b(q,r) - a_r \odot b_r\right)$$

$$= a_r \odot b_r(\epsilon_{sw}^2 + 2\epsilon_{sw}) + (1+\epsilon_{sw})^2\epsilon_{cm}(||a_r||_1||b_r||_1 - a_r \odot b_r)$$

$$= c||a_r||_1||b_r||_1(\epsilon_{sw}^2 + 2\epsilon_{sw}) + \epsilon_{cm}(1+\epsilon_{sw})^2||a_r||_1||b_r||_1(1-c)$$

$$= ||a_r||_1||b_r||_1\left(c(\epsilon_{sw}^2 + 2\epsilon_{sw}) + \epsilon_{cm}(1+\epsilon_{sw})^2(1-c)\right)$$

with probability at least $1-\delta_{cm}$.

The values of c that maximize the error (the RHS) are $c=1$ when $\epsilon_{cm} < \frac{\epsilon_{sw}^2+2\epsilon_{sw}}{(\epsilon_{sw}+1)^2}$, and $c=0$ when $\epsilon_{cm} \geq \frac{\epsilon_{sw}^2+2\epsilon_{sw}}{(\epsilon_{sw}+1)^2}$. The corresponding maximum errors are $||a_r||_1||b_r||_1(\epsilon_{sw}^2+2\epsilon_{sw})$ (for $c=1$), and $||a_r||_1||b_r||_1\epsilon_{cm}(1+\epsilon_{sw})^2$ (for $c=0$).

With a similar analysis, the case of $\widehat{a_r \odot b_r} < a_r \odot b_r$ gives a tighter constraint: $Pr[a_r \odot b_r - \widehat{a_r \odot b_r} > (\epsilon_{sw}^2+2\epsilon_{sw})a_r \odot b_r] < \delta_{sw}$. The lemma follows directly. □

Theorem 2

Proof Similar to the analysis for point queries, we need to consider the two cases of Lemma 4 separately.

Case 1 ($\epsilon_{cm} \geq \frac{\epsilon_{sw}^2+2\epsilon_{sw}}{(\epsilon_{sw}+1)^2}$): By Lemma 4, we set $\epsilon_{cm}(1+\epsilon_{sw})^2 = \epsilon$ in order to achieve the required accuracy. The space complexity then becomes $C(\epsilon) = \frac{1}{\epsilon_{sw}\epsilon_{cm}} = \frac{(1+\epsilon_{sw})^2}{\epsilon\epsilon_{sw}}$. Since $\frac{(1+\epsilon_{sw})^2}{\epsilon\epsilon_{sw}}$ is strictly decreasing for ϵ_{sw} in the interval $[0,1]$, we can minimize the space complexity by setting the maximum value for ϵ_{sw} satisfying the case's precondition $\epsilon_{cm} \geq \frac{\epsilon_{sw}^2+2\epsilon_{sw}}{(\epsilon_{sw}+1)^2}$, i.e., $\epsilon_{sw} = \sqrt{\epsilon+1}-1$. Then, ϵ_{cm} becomes equal to $\frac{\epsilon}{\epsilon+1}$.

Case 2 ($\epsilon_{cm} \leq \frac{\epsilon_{sw}^2+2\epsilon_{sw}}{(\epsilon_{sw}+1)^2}$): By Lemma 4, in order to achieve the required accuracy we need to set $\epsilon_{sw}^2 + 2\epsilon_{sw}^2 = \epsilon \Rightarrow \epsilon_{sw} = \sqrt{\epsilon+1}-1$. Accordingly, $\epsilon_{cm} = \frac{\epsilon}{\epsilon+1}$.

Notice that, similar to the point queries analysis, the two cases lead to the same configuration for minimizing the cost, i.e., $\epsilon_{cm} = \frac{\epsilon}{\epsilon+1}$ and $\epsilon_{sw} = \sqrt{\epsilon+1}-1$. □

B Proofs for distributed setups

Theorem 4 derives worst-case error bounds for the merging of exponential histograms. Lemma 2 and Theorem 5 prove the correctness of the algorithm for continuous self-join size queries.

Theorem 4

Proof We argue that $EH_\oplus$ approximates the exponential histogram of the logical stream, with a maximum relative error of $\epsilon + \epsilon' + \epsilon\epsilon'$, where ϵ is the error parameter of the initial exponential histograms. Consider a query for the last q time units. With $s_q = t - q$ we denote the query starting time. Let Q denote the index of the bucket of $EH_\oplus$ which contains s_q in its range, i.e., $s(EH_\oplus^Q) \leq s_q \leq e(EH_\oplus^Q)$. With i and $\hat{i}$ we denote the accurate and estimated number of true bits in the query range. According to the estimation algorithm, the estimation for the number of true bits in the stream will be $\hat{i} = 1/2|EH_\oplus^Q| + \sum_{1\leq Y<Q} |EH_\oplus^Y|$. This estimation may be influenced by two types of approximation errors: (a) a possible approximation error of the overlap of bucket $EH_\oplus^Q$ with the query range, denoted as err_1, and, (b) a possible approximation error of i, denoted as err_2, because of the inclusion of data that arrived before s_q in buckets $Y \leq Q$, or data that arrived after s_q in buckets $Y > Q$. Let us now look into these two errors in more details.

With respect to err_2, recall that the contents of individual buckets are inserted to $EH_\oplus$ using the starting time and the ending time of the buckets. Therefore, it may happen that some bits arrive before s_q but are inserted to $EH_\oplus$ with a timestamp after s_q, creating 'false positives'. The opposite is also possible. These bits are called out-of-order bits with respect to s_q. Clearly, out-of-order bits may lead to underestimation or overestimation of the query answer. According to Lemma 5, the number of out-of-order bits originating from each exponential histogram EH_x is at most ϵi_x, with i_x denoting the accurate number of true bits that were inserted in EH_x at or after s_q. The number of out-of-order bits from all streams is then bounded as follows: $\text{err}_2 \leq \sum_{x=1}^{n} \epsilon i_x = \epsilon \sum_{x=1}^{n} i_x = \epsilon i$.

Underestimation or overestimation of the overlap may also happen because of the halving of the size of bucket $EH_\oplus^Q$ during query time

(err_1). As shown in [16], this process may introduce a maximum relative error of ϵr, where r is the sum of the sizes of all buckets in $EH_{\oplus}$ with an index lower than Q (i.e., with a starting time at least equal to s_q). Recall that r may also include bits that have arrived before s_q (the out-of-order bits), which is however upper-bounded by ϵi, as discussed before. Therefore, the maximum underestimation or overestimation error is $\text{err}_1 = \epsilon' r \leq \epsilon'(i + \epsilon i) = \epsilon' i + \epsilon\epsilon' i$, with $i = \sum_{x=1}^{n} i_x$.

Summing err_1 and err_2, we get a maximum relative error of $(\epsilon + \epsilon' + \epsilon\epsilon')$, which completes the proof. □

Lemma 5 *Consider an individual exponential histogram EH_x of stream X, configured with error parameter ϵ. The out-of-order bits with respect to the query starting time s_q that EH_x can generate are at most ϵi_x, with i_x denoting the number of true bits arriving at or after s_q in X.*

Proof Due to the non-decreasing nature of bucket timestamps, there can be only one bucket with a start time less than s_q and end time greater than or equal to s_q. Let this bucket be EH_x^j. All other buckets have both starting and ending time at the same side of s_q, and therefore their contents are always inserted with a timestamp at the correct side of s_q and do not create out-of-order bits.

Since the ending time of EH_x^j is at or after s_q, its most recent true bit has arrived at or after s_q, and should be included in the query range. Therefore, the number of true bits arriving at or after s_q in stream X is $i_x \geq 1 + \sum_{b=1}^{j-1} |EH_x^b|$. Furthermore, since half of the bits of EH_x^j are inserted using the ending time and half using the starting time of the bucket, the maximum number of out-of-order bits is $|EH_x^j|/2$. By construction (invariant 1):

$$\frac{|EH_x^j|}{2(1 + \sum_{b=1}^{j-1} |EH_x^b|)} \leq \epsilon \Rightarrow \frac{|EH_x^j|}{2} \leq \epsilon(1 + \sum_{b=1}^{j-1} |EH_x^b|) \leq \epsilon i_x$$

□

Lemma 2

Proof The proof relies on the following properties of the min:

Monotonicity: If $\mathbf{x}[i] \leq \mathbf{y}[i]$ for all i, then $\min_i\{x[i]\} \leq \min_i\{y[i]\}$.

Distributivity: For any monotonically increasing function f, $\min_i\{f(\mathbf{x}[i])\} = f(\min_i\{\mathbf{x}[i]\})$.

We want to derive sufficient conditions such that $(1-\theta)\boldsymbol{f}(\mathbf{v}(t)) \leq \boldsymbol{f}(\mathbf{v}(t_0)) \leq (1+\theta)\boldsymbol{f}(\mathbf{v}(t))$, with $\boldsymbol{f}(\mathbf{v}(t)) = \min_{row=1}^{d}\{||\mathbf{v}[row]||^2\}$. By the distributivity property of the min for monotonically increasing functions (i.e., the square root), it is sufficient to verify:

$$\sqrt{\frac{\boldsymbol{f}(\mathbf{v}(t_0))}{1+\theta}} \leq \min_{row=1}^{d}\{||\mathbf{v}(t)[row]||\} \leq \sqrt{\frac{\boldsymbol{f}(\mathbf{v}(t_0))}{1-\theta}}.$$

By the triangle inequality:

$$||\mathbf{v}(t)[row] - \mathbf{v}(t_0)[row]|| \leq \sum_{j=1}^{n} ||\mathbf{v}_j(t)[row] - \mathbf{v}_j(t_0)[row]||$$

$$= \sum_{j=1}^{n} \boldsymbol{d}_j[row] = n\boldsymbol{d}[row] \Rightarrow$$

$$||\mathbf{v}(t_0)[row]|| - n\boldsymbol{d}[row] \leq ||\mathbf{v}(t)[row]|| \leq ||\mathbf{v}(t_0)[row]|| + n\boldsymbol{d}[row] \quad (7)$$

Notice that $||\mathbf{v}(t_0)[row]||$ is constant per synchronization. Therefore, Inequality 7 bounds $||\mathbf{v}(t)[row]||$ by a linear relation of $\boldsymbol{d}$, i.e., it allows us to form threshold-crossing queries in the R^d space. By monotonicity of the min, it suffices to monitor the following conditions:

$$\min_{i=1}^{d}\{||\mathbf{v}(t_0)[i]|| + n\boldsymbol{d}[i]\} \leq \sqrt{\frac{\boldsymbol{f}(\mathbf{v}(t_0))}{1-\theta}}$$

and

$$\min_{i=1}^{d}\{||\mathbf{v}(t_0)[i]|| - n\boldsymbol{d}[i]\} \geq \sqrt{\frac{\boldsymbol{f}(\mathbf{v}(t_0))}{1+\theta}}.$$

The lemma follows directly, by dividing both sides of the conditions by n. □

Theorem 5

Proof Sketch: By construction, all counters of $\mathbf{v}_i^u(t)$ are at least equal to the corresponding counters of $\mathbf{v}_i(t)$. Therefore, the self-join size estimate for $\mathbf{v}_i^u(t)$ will be at least equal to the self-join size estimate for $\mathbf{v}_i(t)$ at all times. Using Lemma 2 to monitor $\mathbf{v}$ but only considering the shifts which increase the counters, we get that if $\min_{row=1}^{d}\{\frac{||\mathbf{v}(t_0)[row]||}{n} + \boldsymbol{d}^u[row]\} \leq \frac{1}{n}\sqrt{\frac{\boldsymbol{f}(\mathbf{v}(t_0))}{1-\theta}}$, then $\boldsymbol{f}(\mathbf{v}(t_0)) \leq (1+\theta)\boldsymbol{f}(\mathbf{v}(t))$. The lower bound is shown analogously. □

TOPiCo: Detecting Most Frequent Items from Multiple High-Rate Event Streams

Valerio Schiavoni*,
Etienne Rivière,
Pierre Sutra,
Pascal Felber
Université de Neuchâtel,
Switzerland

Miguel Matos,
Rui Oliveira
INESC TEC & University of
Minho, Portugal

ABSTRACT

Systems such as social networks, search engines or trading platforms operate geographically distant sites that continuously generate streams of events at high-rate. Such events can be access logs to web servers, feeds of messages from participants of a social network, or financial data, among others. The ability to timely detect trends and popularity variations is of paramount importance in such systems. In particular, determining what are the most popular events across all sites allows to capture the most relevant information in near real-time and quickly adapt the system to the load. This paper presents TOPiCo, a protocol that computes the most popular events across geo-distributed sites in a low cost, bandwidth-efficient and timely manner. TOPiCo starts by building the set of most popular events locally at each site. Then, it disseminates only events that have a chance to be among the most popular ones across all sites, significantly reducing the required bandwidth. We give a correctness proof of our algorithm and evaluate TOPiCo using a real-world trace of more than 240 million events spread across 32 sites. Our empirical results shows that (i) TOPiCo is timely and cost-efficient for detecting popular events in a large-scale setting, (ii) it adapts dynamically to the distribution of the events, and (iii) our protocol is particularly efficient for skewed distributions.

Keywords

top-k query, top-k frequency, distributed systems, event processing

Categories and Subject Descriptors

C.2.4 [**Computer-Communication networks**]: Distributed systems

*Corresponding author: valerio.schiavoni@unine.ch

DEBS'15, June 29 - July 3, 2015, OSLO, Norway.

http://dx.doi.org/10.1145/2675743.2771838.

1. INTRODUCTION

The collection of aggregates and statistical metrics over online data streams has attracted considerable attention from both academia and industry over the past decade. Mining the properties of such data streams can be used in various contexts, ranging from targeted advertisement [6], network analysis [7, 25], or automated virus detection [20]. Of the various statistical information that can be computed over a stream, identifying the distribution of the events is of practical interest. This information might help a content delivery network infrastructure to dimension its caches using the stream of web access logs. It can also detect sudden changes in popularity, e.g., flash crowds phenomena or denial of service attacks.

Practical streaming systems exhibit heavy-tailed distributions. Moreover, in a vast majority of use cases, only the most frequent items are required. As a consequence, collecting and storing the frequency of all events to compute only the k most frequent ones is a waste of resources. This observation is particularly acute in a geo-distributed setting, where distinct sites might receive distinct stream of events. Detecting the k most frequent items is a form of top-k query processing, where the query is simply the sum of occurrences grouped by item. As a consequence, we shall use the term *top-k frequent items* in the remainder of this paper.[1]

We are interested in the construction of the top-k frequent items from the union of multiple streams. These different streams are generated at multiple geographically distant locations. We consider the following motivating scenario. A set of geo-distributed servers located in different regions of the world support the information needed for a large-scale event, e.g., the Olympics or the FIFA World Cup. Servers receive and emit messages, similarly to the Twitter service, for the local area. Participants and spectators can comment and react while the events occur, online. Messages are tagged with keywords regarding the event, e.g., the names or moods of the participants. Each site maintains the top-k frequent keywords over a sliding window, allowing to observe the trends for a particular region. We are also interested in computing the most frequent keywords at the scale of the geo-distributed infrastructure. Such information is used

[1]As pointed out by [26], our problem differs from the simple top-k problem where unique items with the highest values for a given attribute are returned, which is merely a selection problem [2]. A top-k *query* problem on the opposite, e.g., top-k counting and top-k frequent items, requires to aggregate multiple instances of the same item from different sites.

locally, for instance, to see in near real-time the differences between local and global trends. A naive solution for such a scenario consists in redirecting all the streams towards all the sites. Obviously, this solution does not scale with the number of sites, nor with the geo-distribution. Besides, although solutions exist for speeding up data streams over multiple data centers [22], they have huge bandwidth costs, as well as higher-than-required computational power.

We are thus interested in maintaining the *global* top-k frequent items in a multi-site information system. The global top-k frequent items set is referred as $\mathcal{G}$ in the remainder of this paper. We assume that each site maintains, using an existing single-site algorithm, the *local* top-k frequent items for its own stream. This local top-k frequent items set for site i is denoted as $\mathcal{L}_i$. Our goal is to construct, at each site i, a version of the global top-k frequent items, denoted as $\mathcal{G}_i$. The construction of $\mathcal{G}_i$ is based on a combination of $\mathcal{L}_i$ and of information received from the other sites. The global top-k frequent items set $\mathcal{G}$ is never materialized as there is no centralized entity receiving all the streams without transmission delays. The objectives for the construction of $\mathcal{G}_i$ at each site are twofold. First, we want to minimize the amount of information that is exchanged between sites. Second, we want to minimize the deviation between $\mathcal{G}_i$ maintained at one site, and the hypothetical and ideal content of $\mathcal{G}$, as constructed by an omniscient observer collecting all the streams in real time.

Contributions. We make the following contributions to the top-k frequent items problem. We propose TOPICO, a novel protocol for computing an accurate view of the globally most frequent items at the scale of multiple geo-distributed sites. Our protocol leverages the presence of a continuously maintained local list of the top-k frequent items at each site. Sites exchange such lists up to a certain depth, while ensuring that the global view $\mathcal{G}_i$ computed at each site effectively contains the appropriate elements and their respective frequencies, within reasonable and practical delay. We provide a correctness proof of the protocol, as well as an extensive experimental evaluation using a prototype and fed with traces of 240 million events received on 32 sites, collected during the FIFA World Cup'98 [1]. The evaluation confirms that TOPICO is a lightweight approach and shows that it is able to dynamically adapt to the items distribution. The rest of the paper is organized as follows: Section 2 discusses related work. Section 3 precisely formulates the top-k frequent items problem and circumscribe the conditions under which this problem is solvable. Section 4 details our TOPICO protocol and covers its correctness proof. Section 5 presents our experimental evaluation of the TOPICO prototype. Finally, Section 6 concludes the paper.

2. RELATED WORK

The problem of efficiently computing the results of top-k frequent items from a stream has been considered both in centralized and distributed settings. Our focus is on the construction of the *global* top-k frequent items sets $\mathcal{G}_i$ at each sites of a distributed system rather than the construction of the *local* top-k frequent items sets $\mathcal{L}_i$ for each individual sites. The construction of $\mathcal{L}_i$ employs a centralized algorithm, which can be implemented using a stream processing engine [3, 12]. As this construction is not the focus in this paper, we use a simple counting-based approach and concentrate on the distributed aspects. We provide nonetheless a review of centralized solutions and alternatives that can be used to build $\mathcal{L}_i$ at each site, as well as similar and related problems that could benefit from our distributed algorithm. Then, we review distributed algorithms and protocols allowing to compute $\mathcal{G}$, by collecting information from multiple sites. Note that the problem of top-k frequent items is sometimes called the *heavy hitters* problem in the literature.

2.1 Centralized top-k frequent items

Ilyas et al. [13] present a survey of top-k query algorithms for centralized single site relational database systems. The queries that they consider include *counting* queries, which themselves include *frequent items* queries. The survey however does not consider the data streaming model but only instantaneous queries.

The FREQUENT algorithm of Misra and Gries [18] was an early approach proposed for the detection of frequently occurring items in an infinite stream. It allows outputting the set of elements that account for more than a fraction $\frac{1}{f}$ of the total stream size, i.e., to return the set $j : c_j > \frac{c}{f}$ where c is the size of the stream seen so far, and c_j is the number of occurrences of element j. This is a different problem than ours, as top-k frequent items may be less present in the stream than $\frac{c}{f}$ even for $f \geq k+1$. The algorithm maintains $f-1$ counters, associated with unique items. For each item seen in the stream, the corresponding counter is incremented if it exists (or created if there are unused counter). Otherwise, all counters are decremented and freed when they reach zero. The counters are eventually associated with up to $f-1$ items that are present in more than a fraction $\frac{1}{f}$ of the stream.

The *Count-Min* sketch of Cormode and Muthukrishnan [5] is a data structure dedicated to the summarization of data streams. It can allow detecting the most frequent items, among several other operations. It would be a possible alternative for the computation of the $\mathcal{L}_i$ at each site.

Lahiri et al. [14] consider the problem of tracking persistent items in a stream. This is a different, but complementary problem to top-k frequent items. Computing both sets can allow indicating trends over time or detect sudden changes in popularity of items.

Wang et al. [24] propose a framework to execute top-k *pattern* queries. Such queries allow recognizing the most frequent sequences of events for which the patterns corresponding to the interdependency conditions apply. This technique uses adaptive join scheduling strategies and stratified stream graphs. The output could be used as the local top-k frequent items set $\mathcal{L}_i$ used in our algorithm, allowing to compute the result of a global top-k *pattern* query.

Wong and Fu [26] present a probabilistic solution for top-k frequent *item sets* over a stream. The problem they consider is more general than the one we consider in this paper. Their goal is to detect the most frequent *item sets*, i.e., set of l items that frequently appear together from a stream of transactions. Note that the two problems are equivalent if $l = 1$, i.e., when considering individual items. The authors propose two algorithms that can derive the top-k frequent *item sets* from a complete stream without prior knowledge of its statistical characteristics: an algorithm based on Chernoff bounds and the Top-k *Lossy Counting* algorithm. These algorithms could also be candidate for computing $\mathcal{L}_i$ at each of site. Furthermore, when combined with our contribution, they allow to compute the global top-k frequent *item sets* at each site.

The Vitter's reservoir [23] samples uniformly a complete stream using a fixed-size reservoir, which is simply an array of samples. The reservoir allows estimating the distribution of events from the beginning of the stream. If this distribution of events presents enough skew (e.g., the popularity follows a power law), a large enough reservoir may allow answering top-k frequent items. However, there is no guarantee: Items belonging to the most frequent ones may simply be missed if there are not sampled enough in the reservoir.

2.2 Distributed top-k frequent items

The *threshold algorithm* (TA) of Fagin et al. [9] computes the top-k frequent items from a collection of items in independent sets. The algorithm does not consider data streams but static sets, which correspond to independent disk drives in a relational database setting. They can be considered as sites in our system model. A coordinator process computes $\mathcal{G}$ through several rounds of interaction with the sites. Each of the site maintains its list of items in decreasing frequency order. The algorithm goes down the list at all sites in parallel. In its TA-random variation, the algorithm computes for each new item its aggregate value. This requires being able to perform random accesses to the lists. A variant for systems, where only sequential access is possible (or preferred), is named the TA-sorted or NRA in [9], as well as Stream-Combine in an independent work by Guntzer et al. [11]. TA-sorted first computes the set of items that are part of the top-k frequent items, without calculating their exact aggregate count values. It requires an additional final pass to perform these aggregations. For both TA-random and TA-sorted, thresholds are computed to allow determining when elements that are down the lists at each site will not be included in the top-k frequent items, that is, their aggregate value cannot be higher than the k^{th} element in $\mathcal{G}$. The threshold can be exact in the TA-random case or approximated by bounds of worst and best possible scores for TA-sorted. Since lists are sorted in decreasing frequency order, the algorithm can stop when new elements have frequency lower than the (worst case) threshold.

Both threshold algorithms require $O(N^2)$ operations, where N is the number of sites. The number of iterations is also unbounded, and depends on the similarity of rankings between the different $\mathcal{L}_i$ of the different sites . In a distributed setting, this prevents from giving any guarantee on the delay of calculation of $\mathcal{G}$ at the coordinator. While this might be acceptable in a relational database setting, it is not adapted in a distributed data streaming model. Such an arbitrary delay may incur an important and uncontrollable drift between the content of $\mathcal{G}_i$ at the coordinator and the actual $\mathcal{G}$ an omniscient observer would obtain. Furthermore, the number of iterations at a site depends on the popularity distribution of items at that particular site. This may lead to the content of $\mathcal{G}_i$ being based on sliding windows for uncorrelated time periods at the different sites. As a consequence, in both cases, the final content of $\mathcal{G}_i$ might be inadequate to effectively detect trends across all sites in a timely manner.

Probabilistic versions of the threshold algorithms named the Prob-sorted algorithms family, were proposed by Theobald et al. [21]. They produce estimations of $\mathcal{G}$ based on probabilistic score predictions. This allows reducing the number of accesses to the lists at each site, in particular when the order of elements highly differ from one site to another, but does not guarantee that the exact scores for the aggregate count values are computed.

The TPUT algorithm of Cao and Wang [4] addresses the limitation of the threshold algorithm for static data sets. It explicitly targets a distributed setting with a coordinator site that computes the value $\mathcal{G}$ through interaction with the other sites. Unlike the threshold algorithm, TPUT requires 3 rounds between the coordinator and the sites. In the first round, the algorithm computes a lower bound τ on the value associated with the k^{th} element in $\mathcal{G}$. Each site sends its top-k elements from $\mathcal{L}_i$ to the coordinator, which uses the bottom value computed by the aggregation of these lists as τ. In the second round, the coordinator selects the threshold $T = \frac{\tau}{N}$. All sites then send back the elements not previously sent with a frequency of at least T. At this point, the coordinator can determine an overset S of the elements that form $\mathcal{G}$. A third round is required to collect the actual values associated with the elements in S, in order to determine the final and exact content of $\mathcal{G}$.

TPUT operates on a static set; it does not consider the data streaming model, nor computations on sliding windows. Furthermore, it uses a single coordinator site, while our problem is to compute $\mathcal{G}_i$ at each site. TPUT could be instantiated with a coordinator at each site: In this case, each site would have to perform the three phases and pull information from all the other nodes. Our approach is the opposite. Each site decides on its own using only local information what it needs to send to the others. This allows a single, one way communication to inform about the evolution of $\mathcal{L}_i$, and therefore the changes to the $\mathcal{G}_i$ on other sites, in comparison with the three two ways communication and the resulting higher delays and load with TPUT. We also exploit the evolution of $\mathcal{L}_i$ through time whereas TPUT is essentially a one-shot algorithm that recomputes $\mathcal{G}$ from scratch for each new query.

Manjhi et al. [16] consider the top-k frequent items problem with multiple sites where sites are organized in a tree structure. Their approach is to build an approximate version of $\mathcal{G}$. They allow nodes to define the degree of precision, and introduce the notion of precision gradient that captures the precision of the combination of approximate frequency counts along the tree, with the goal of minimizing the bandwidth cost as much as possible. We do not consider the use of a rigid overlay between nodes, and we target a complete computation of the top-k frequent items at all nodes.

Michel et. al [17] present the KLEE system which target a range of top-k queries including counting, and thus cover top-k frequent items. This solution works for P2P networks, and it provides an approximate computation of $\mathcal{G}$. It is unclear how the solution can be evolved to support a data streaming model, or a model that supports dynamic data sets in general.

Some solutions are based on gossip-based protocols, where a periodic interaction takes place between pairs of nodes in a probabilistic manner, and with no complete membership information maintained at each node. Lahiri and Tirthapura [15] present such a gossip-based algorithm using adaptive sampling techniques. Their algorithm can consider absolute thresholds (more than a certain quantity of items are present in the system, regardless of its size), and relative thresholds similar to the model used by Misra-Gries [18]. The set of the *probabilistically most frequent items* is available at all peers after convergence. Sacha and Montresor [19]

present another gossip-based protocol that targets the same problem but achieves a higher efficiency. Guerrieri et al. [10] present an evolution of the algorithm of Sacha and Montresor [19] to account for the addition and deletion of items. It is nonetheless unclear if any of these approaches could be adapted to sets of events that evolve very rapidly over time, and in particular to the data streaming model. Furthermore, even the addition and removal of items may be reflected after several rounds of gossiping as it is difficult to track and remove items that rapidly leave the window of interest. This indicates that such decentralized solutions are more adapted to static sets with complete periodic re-computation, or to slowly evolving sets where the delay of propagation is less an issue.

3. TOP-K FREQUENT ITEMS PROBLEM

This section defines the elements of our system model, then introduces the problem of computing the top-k frequent items over a distributed set of streams. Further, we state two results regarding the space complexity of every solution. These results serve as guidelines to our TOPICO protocol.

3.1 System model

We consider a system composed of N distributed sites that communicate through message-passing. For the sake of simplicity, we assume that the communication graph is complete, and that sites are able to send/receive messages via a reliable communication medium. We shall precise our synchrony assumptions in the following.

Every site i receives a continuous stream s_i of events at some rate λ_i. Each event in the streams refers to some uniquely identified item (e.g., a topic, a keyword). We note *Items* the countable set of items, and we assume some ordering $<$ over *Items*. Every time a site i receives an event e, i tags e with the reception time. Based on this timestamping mechanism, every site i continuously keeps track of the events received within a time-based sliding window W_i of length τ.

The *top-k frequent items* problem requires to compute at each site a view of the most frequent items received globally, across all sites, within the last τ units of time. To model this problem, we consider that each site i holds two data structures $\mathcal{L}_i$ and $\mathcal{G}_i$, as described next:

- Variable $\mathcal{L}_i$ maintains the *local top-k frequent items*. This variables stores in order the items received at site i, together with their respective number of occurrences in the time window W_i (ties are broken according to $<$). For the sake of simplicity, we shall be assuming hereafter that $\mathcal{L}_i$ contains in fact *all* the items in W_i with their number of occurrences in a sound order.
- Variable $\mathcal{G}_i$ stores the *global top-k frequent items* as collected by site i. This means that $\mathcal{G}_i$ contains the local view at site i of the k most frequent items received globally in the streams $(\lambda_i)_i$ during the last τ units of time.

To illustrate the above data structures, consider that site i received items "x" at times 1 and 2, and respectively "y" and "z" at times 3 and 4. Further consider that the sliding window length equals 3. In such a case, at time $t = 3$, we have $\mathcal{L}_{i,3} = \{(\text{"x"}, 2), (\text{"y"}, 1)\}$, while at time $t = 4$, $\mathcal{L}_{i,4} = \{(\text{"x"}, 1), (\text{"y"}, 1), (\text{"z"}, 1)\}$ holds. Then, if we consider that $k = 1$ and $<$ is the Lexicographical order, we have $\mathcal{G}_{i,3} = \{(\text{"x"}, 2)\}$ and $\mathcal{L}_{i,4} = \{(\text{"z"}, 1)\}$.

The core task of any top-k frequent items protocol is to maintain at each site i a value of $\mathcal{G}_i$ consistent with the content of the streams $(s_i)_i$. Intuitively, we want to compare the local computation of $\mathcal{G}_i$ against an omniscient computation of the globally most frequent items. Next, we state such a notion in more formal terms.

3.2 Problem statement

Let us note $\mathcal{G}$ the top-k most frequent items in $(s_i)_i$ over the last period of τ units of time. In the top-k frequent items problem, the key information is the ordering of the items. We capture this by measuring the distance d between the ordering of the items in $\mathcal{G}$ and $\mathcal{G}_i$ at some site i.[2] For some site i, we shall note $d(\mathcal{G}_{i,t}, \mathcal{G}_t)$ the distance between $\mathcal{G}_i$ and $\mathcal{G}$ at time t.

DEFINITION 1. *A top-k frequency protocol is* perfect *when, for any ϵ, $lim_{t\to\infty} d(\mathcal{G}_{i,t}, \mathcal{G}_t) < \epsilon$ holds.*

Clearly, constructing such a perfect algorithm is not always possible. This might be for instance the case when the stream rate times the message delay between sites is higher than one. Indeed, whenever a site i received some information about the most frequent items at site j, such an information can be outdated by the arrival of a novel item at site j. The two results that follow further circumscribe the conditions under which a perfect solution is constructible for the top-k frequency problem. With more details, Lemma 1 proves that the problem requires a bound on the message delay between sites. Then, we show in Lemma 2 that the existence of a greatest element in *Items* for the order $<$ is necessary.

LEMMA 1. *No top-k frequent items protocol is perfect in an asynchronous distributed system.*

PROOF. (By contradiction.) Consider a system with two sites, i and j, and assume that $k = 1$ holds. In addition, suppose that x and y are the two sole items received respectively at sites i and j at rate $\frac{1}{\tau}$, starting from time 0. Furthermore, consider that $x > y$ holds for the arbitrary order defined to break ties. It follows that at any time t, x is the only item in $\mathcal{G}_t$. Since the system is asynchronous, there is no bound on the message delay between the two sites. In particular, for any value of τ, we might consider repeated arbitrary asynchrony periods longer than τ during which site j does not receive the messages from site i. During such a period, $\mathcal{G}_j$ cannot contain only item y as it would differ from $\mathcal{G}$. However, by a simple undistinguishability argument, we might also consider the exact same run up to that point, and consider now that site i does not received the events associated to x during the asynchrony period. Hence, during such a run, $\mathcal{G}_j$ should only contain y; a contradiction. □

LEMMA 2. *Finding a perfect solution to the top-k frequent items problem requires a greatest element in Items for the order $<$.*

PROOF. (By contradiction.) Let us consider again two sites i and j and that $k = 1$. Stream s_j is empty, while stream s_i is a continuous sequence of distinct items $x_0, x_1, \ldots$, growing for the order $<$, and received at the fixed rate λ from

[2]Our problem definition does not depend on a specific distance function d. Meaningful distance functions include Levenshtein's or the Kendall-Tau rank distance [8]. We use Kendall-Tau in Section 5.

time 0. In addition, consider some non-null message delay from site i to site j. Clearly at time t, $\mathcal{G}_t$ equals $\{(x_{\lfloor \frac{t}{\lambda} \rfloor}, 1)\}$. On the other hand, observe that for any $k \geq 1$, at time $t = k \times \delta$, site j never received the item $x_{\lfloor \frac{t}{\lambda} \rfloor}$. Hence, the distance between $\mathcal{G}_{j,t}$ and $\mathcal{G}_t$ never converges toward 0. □

To accommodate with these results, we introduce two additional properties for top-k frequent items protocols.

DEFINITION 2. *A protocol shall be* ϵ-good *when at all time* t*, for every site* s_i*,* $d(\mathcal{G}_{i,t}, \mathcal{G}_t) < \epsilon$ *holds.*

DEFINITION 3. *A top-k frequent items protocol is* eventually perfect *when,* $(\mathcal{G}_t)_t$ *convergent implies* $lim_{t\to\infty} d(\mathcal{G}_{i,t}, \mathcal{G}_t) = 0$.

Section 5 shows that the TOPiCo protocol exhibits on average a 0.24-goodness factor for the Kendall tau rank distance during our experiments on the given dataset. Section 4.3 proves that TOPiCo is eventually perfect.

3.3 Resolvability

As pointed out previously, the goal of any top-k frequent items protocol is to exchange the minimal amount of information about the local top-k to ensure that $d(\mathcal{G}_{i,t}, \mathcal{G}_t)$ is minimal at each site. This means that a site should send only the items in $\mathcal{L}_i$ that are likely to enter in $\mathcal{G}$. Of course, at least the top-k items in $\mathcal{L}_i$ have to be sent to the other sites. However, it is also obvious that just sending only those entries is not enough to correctly compute $\mathcal{G}$. We state this simple observation in the lemma that follows.

LEMMA 3. *Exchanging the top-k items from* $\mathcal{L}_i$ *among all sites is not sufficient to be eventually perfect.*

PROOF. To prove the above claim, we exhibit a simple counter-example. We consider two sites i and j such that at some point in time we have: $\mathcal{L}_i = \{(a, 10), (x, 6)\}$ and $\mathcal{L}_j = \{(d, 9), (x, 5)\}$. In addition, let us consider that $k = 1$. If site i and j only exchange their first entries, i.e., respectively $(a, 10)$ and $(d, 9)$, we shall have at both sites $\mathcal{G}_i = \mathcal{G}_j = \{(a, 10)\}$. However, we clearly have that $\mathcal{G} = \{(x, 11)\}$. Hence, the previous approach never converges toward the correct solution. □

At a consequence of the previous result, we need to determine locally a value $l \geq k$ such that sending the top l items is sufficient to converge toward $\mathcal{G}$. Our next result shows that such a l exists by proving that a full information protocol is eventually perfect.

LEMMA 4. *A full information protocol is an eventually perfect top-k frequent items protocol.*

PROOF. Let us first recall that a full information protocol consists at each site in sending the local state every time this state changes and storing all historical data. Then consider some run of this protocol. Since $\mathcal{G}_t$ is convergent, $lim_{t\to\infty}\mathcal{G}_t$ exists. We note G such limit and T the time after which $\mathcal{G}_{t>T} = G$. Consider some tuple (x, ω) in G. From the definition of G, ω is the aggregated value of the number of occurrences of x in $(\mathcal{L}_{i,t})_i$ for every $t > T$. Hence after time T, computing the top-k frequent items on $(\mathcal{L}_{i,t})_i$ for any $t > T$ leads to G. Since we make use of a full information protocol, once every sites i broadcasts $\mathcal{L}_i$ after time t, we have eventually $\mathcal{G}_i = G$ at all sites. □

Despite a full information protocol is a correct solution, it requires to broadcast an information every time a novel event is received. This is not practical. On the contrary, the goal of our TOPiCo protocol is to allow sites constructing $\mathcal{G}$ efficiently, by forwarding as few information as possible. In the next section, we detail the internals of our approach, and prove that TOPiCo is eventually perfect.

4. THE TOPICO PROTOCOL

In this section, we describe the TOPiCo protocol in detail. We first start with an overview of our approach, while providing key insights on the internals of our solution. The concluding part of this section provides a formal proof that TOPiCo is eventually perfect.

4.1 Overview of the protocol

Each TOPiCo site i executes the following two tasks: (Update) Site i computes locally a list of *candidates* items that it broadcasts together with their local number of occurrences to all sites. (Disseminate) Upon receiving a list of candidates from some distant site j, a site i updates its global view of the most frequent items $\mathcal{G}_i$. To that end, i first sums-up for each item the contribution received in the candidate lists from the other sites (such contribution equals 0, if the item was not received). Then, site i sorts the global contributions and outputs $\mathcal{G}_i$. We consider that the system is synchronous, and that the update task occurs at frequency $1/\delta$.

TOPiCo constructs a list of candidates by determining at each site a value $l \geq k$ for which the top-l ranked items in $\mathcal{L}_i$ have a chance to enter in $\mathcal{G}$. Such an estimation is based on (i) the global number of occurrences of each item among the top-k in $\mathcal{G}_i$, (ii) the number of occurrences of each items in $\mathcal{L}_i$, and (iii) the candidates received by remote sites. In what follows, we cover with more details the internals of our approach, and how this estimation is computed.

4.2 TOPiCo in detail

Our first key observation in the design of TOPiCo is the following:

> (**Observation 1**) Consider an item x at some position lower than k in $\mathcal{G}$. If x enters in the top-k most ranked items in $\mathcal{G}$, then there exists a site i for which the number of occurrences of x at i times N is greater than the number of global occurrence of the item at position k in $\mathcal{G}$. Thus, item x is at site i a candidate to enter $\mathcal{G}$.

To actually transform the above observation into an algorithm, we must then accommodate with the fact that no site has access to $\mathcal{G}$. First, every site i executes the above *candidacy test* on the items in $\mathcal{L}_i$, using $\mathcal{G}_i$. Then, we make a second observation:

> (**Observation 2**) Consider that some item x passes the candidacy test at site i, i.e., denoting ω the number of occurrences of x in W_j, we have $\omega \times N > \mathcal{G}_i[k-1]$. Item x might fail the candidacy test at some other site j, for instance if j never receives x. Hence for every candidate it receives, site j must piggyback its local number of occurences.

We base our TOPiCo protocol on the above two observations. Algorithm 1 presents its pseudo-code. In addition

Algorithm 1: TOPiCo at process i

```
1  variables
2      candidates_i ;                          // the candidates
3      L_i ;                                   // local top-k
4      G_i ;                                   // global top-k
5      δ ;                                     // update period
6  task update
7      upon receive ⟨UPDATE, C⟩ from j
8          candidates_i[j] ← C
9          foreach (x, _) ∈ C do
10             G_i ← G_i \ {(x, _)}
11             Ω ← Σ_{(x,ω)∈candidates_i} ω
12             G_i ← G_i ∪ {(x, Ω)}
13         G_i ← {G_i[0], ..., G_i[l − 1]} ;   // keep top-k
           items
14 task disseminate (every δ second)
15     let (_, Γ) = G_i[k − 1] ;   // lowest score in top-k
16     l ← k
17     while l < |L_i| do
18         let (x, ω) = L_i[l]
19         if ω × N ≥ Γ then
20             l ← l + 1 ;          // candidacy test passed
21         else
22             break
23     C ← ∅ n ← 0
24     while n < l do
25         C ← C ∪ L_i[n]
26         n ← n + 1
27     foreach (x, _) ∈ candidates_i do
28         if (x, _) ∉ C ∧ (x, ω) ∈ L_i then
29             C ← C ∪ {(x, ω)}
30     broadcast ⟨UPDATE, C⟩ to all sites
```

to $\mathcal{L}_i$ and $\mathcal{G}_i$, the protocol uses two additional local variables: δ defines the periodicity of the dissemination task, and $candidates_i$ is an array that contains for each site i, the last candidates received at site i from j.

In details, TOPiCo works as follows. The *update* task is in charge of maintaining the candidates at every site i. Upon the reception of a new set of candidates C from some site j (which might be i), the update task assigns C to $candidates_i[j]$ (line 8). Then, the computation of $\mathcal{G}_i$ takes place. For every item x in C, site i computes the aggregated value of x over all the candidates in $candidates_i$ and updates variable $\mathcal{G}_i$ accordingly (lines 10 to 12). Notice that at line 11, we write for simplicity $(x, \omega) \in candidates_i$ instead of considering (x, ω) in the multiset $\bigcup_j candidates_i[j]$. The update task ends by truncating $\mathcal{G}_i$ to only keep the first l entries (line 13).

The core routine of TOPiCo is the *disseminate* task. Its goal is to broadcast the candidates computed at site i with a periodicity of δ units of time. The candidates are the $l \geq k$ most frequent items in $\mathcal{L}_i$. To determine the value of l, the disseminate task first computes Γ, the number of occurrence of the last item in $\mathcal{G}_i$ (line 15). Then, it traverses $\mathcal{L}_i$ starting from position k (lines 17 to 22). Every time an item x successes the candidacy test, l is incremented (line 20); otherwise the loop ends (line 22). Site i collects the number of occurrences of items that successfully passed the candidacy test to form the candidates list C (lines 23 to 26). Then, site i appends to this list the candidates that were sent by other sites. (lines 27 to 29), and broadcasts the final content of C to all sites (line 30).

4.3 Proof of Correctness

This section is devoted to a formal proof of the correctness of TOPiCo. We formulate this result below then detail how to achieve it.

THEOREM 1. *The* TOPiCo *protocol is eventually perfect.*

PROOF. (By contradiction.) Let us consider some run ρ of the TOPiCo protocol. As $\mathcal{G}_t$ is convergent, we know that $lim_{t\to\infty}\mathcal{G}_t$ exists. Let G be that limit and T the time after which $\mathcal{G}_{t>T} = G$ holds. At some site j, we note $\mathcal{L}_j[l].\omega$ (respectively $\mathcal{G}_j[l].\omega$) the number of occurrences of the item at rank l in $\mathcal{L}_j$ (resp. $\mathcal{G}_j$), and for some item x, $\omega_{x,j}$ the number of occurrences of x in W. At every site j, recall that for every item z at position l in $\mathcal{L}_j$, we have $\mathcal{L}_j[l].\omega = \omega_{z,j}$.

Consider a time t after T in the run ρ. For the sake of contradiction, assume that an item x is in $\mathcal{G}$, but not in $\mathcal{G}_i$ at some site i. Name y the last item in $\mathcal{G}_i$. Since item y belongs to $\mathcal{G}_i$ and the system is synchronous, y also appears in every $\mathcal{G}_j$ at the same position $l_j \leq k$. Moreover, as item x appears in $\mathcal{G}$ and not y after time T. there must exist some site j_0 for which $\omega_{x,j_0} \times N > \mathcal{G}_{j_0}[l_j]$.

From the above analysis, it follows that for every item z higher than x (and including it) in $\mathcal{L}_{j_0}$. we have $\omega_{z,j_0} \times N > \mathcal{G}_{j_0}[k-1].\omega$, Hence, x passes the candidacy test (line 20) at site j_0. From which we deduce that x is among the candidates at that site, and is broadcast to all sites (lines 23 to 30). Then, every site receiving (x, ω_{x,j_0}), adds x to its candidates list, if it was not the case previously (lines 27 to 29). It follows that at the end of the computation round, x precedes y in $\mathcal{G}_i$. □

5. EVALUATION

We evaluate TOPiCo using a real workload and a real implementation. The prototype is implemented using a combination of C and the Lua programming languages.The experiments are run on a cluster of 29 bi-quad-core Xeon machines, each with 8 GB of RAM and interconnected using a switched 1 Gbps network.

The workload consists of a trace of HTTP requests to the sites hosting the FIFA World Cup'98 website [1]. The data was collected for more than 80 days, including during the competition finale, and contains 240 million requests (events) over 32 different sites spread non-uniformely across the globe.

We start by analyzing the properties of the trace. Instead of fully characterizing the trace, which was done in detail in [1], here we focus just on the metrics pertinent to this paper. Next, we evaluate TOPiCo in terms of resource efficiency and closeness to the ideal $\mathcal{G}$ as computed by an omniscient entity.

5.1 Workload

Figure 1 shows the evolution of the number of sites and events during the competition. From the 80 days available in the trace, in the rest of the evaluation we focus just on days

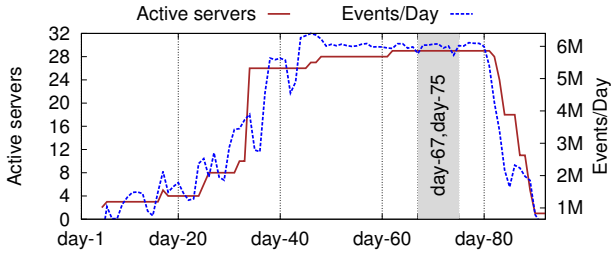

Figure 1: Active servers vs total events per day across active the servers. The gray area highlights the period considered in the rest of the evaluation.

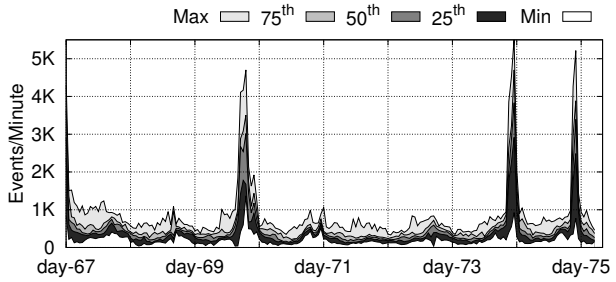

Figure 2: Events per minute across all active 29 servers between day-67 and day-75. There are more than 48 million events during this period.

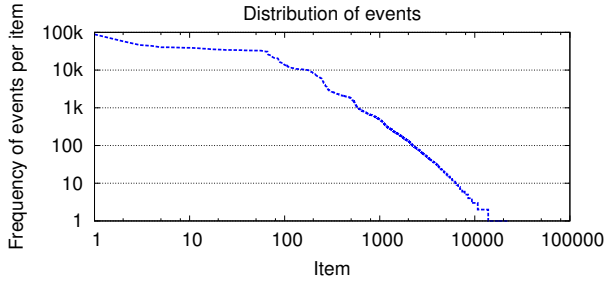

Figure 3: Distribution of event popularity across several days (the days not shown follow the same pattern). Note that the plot is log-log.

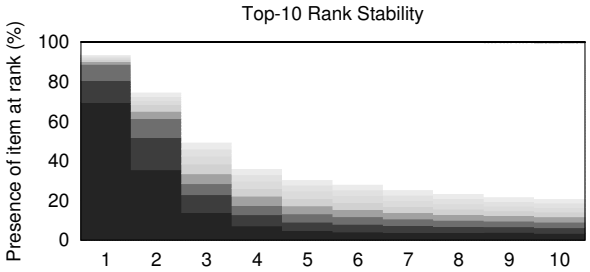

Figure 5: Rank stability showing the percentage of time a position in the rank is occupied by a given item.

67 to 75 which correspond to the competition finale. The reason for this is not only to run the experiments in a more reasonable timeframe, but also because this is where the highest load happens, and hence where TOPiCo becomes more interesting. During this peak period there are 29 active sites and around 6 million events per day.

In Figure 2, we show the overall number of events per minute across all sites for this peak period. We use a representation based on stacked percentiles throughout this section. The white bar at the bottom represents the minimum value, the pale grey on top the maximal value. Intermediate shades of grey represent the 25th, 50th -the median-, and 75th percentiles. For instance, the median number of events per minute around day-70 (during the peak) is 3,000. Clearly, there are peaks in load which will affect not only resource consumption but also the closeness of the computed $\mathcal{G}_i$ to the ideal $\mathcal{G}$.

The efficiency of TOPiCo, and in general of any top-k frequent items algorithm, depends on the distribution skew of events popularity. In fact, if all events where equally popular, computing the most popular ones would be not only impractical but also useless. Figure 3 depicts the distribution of event popularity for several days. as it is possible to observe, the distribution is mildly skewed meaning that the difference in popularity on the most popular items is only moderate. This implies that, sometimes, the list of candidates to enter the top-k might grow large thus affecting the efficiency of TOPiCo. Our experiments evaluating the efficiency of TOPiCo confirm this observation.

We complete the characterization of the workload by assessing how the composition of the top-k evolves over time. For the sake of visualization, we only show results for K=10 and a sliding window of 20 seconds. A total of 311 different items appear in the top-10 during the time period between day-67 and day-75. Different perspectives of the same data are shown in Figure 4 and Figure 5. Each unique item is associated with a unique color. Figure 4 shows which items made it to the top-10 over time. Clearly some items are close to the topmost popular, while others stay mostly close to the bottom. This indicates that there is some stability on the most popular items. Figure 5 confirms this observation by showing the distribution of the time a given position in the top-k is occupied by a given item. For instance, we can see that a given item is the most popular (top-1) more than 60% of the time.

5.2 TOPiCo

We now focus on assessing TOPiCo when subject to the workload described above. Unless stated otherwise, presented results are the average over all sites with the following parameters: k=20, the size of the window is 15 seconds and the update period δ is 5 seconds. We start by observing how effective TOPiCo is in reducing the number of entries exchanged among sites. A naive approach would always broadcast the full $\mathcal{L}_i$ regardless of the workload. TOPiCo on the other hand exchanges just the minimal number of items necessary to compute correctly $\mathcal{G}_i$. These results are show in Figure 6. At Figure 6(top), we show the number of items sent as a fraction of the total number of items in $\mathcal{L}_i$. Clearly, TOPiCo is able to adapt the number of items sent accordingly to the workload. Still, sometimes up to 80%

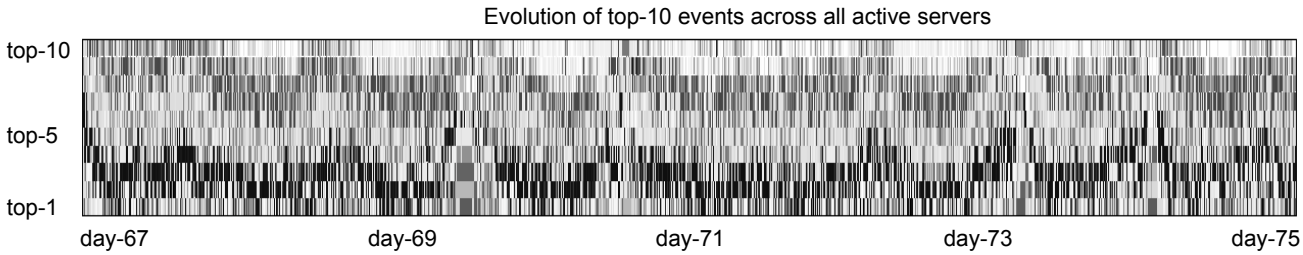

Figure 4: Trends of top-10 ranks computed by an omniscient observer over the 29 active servers. The sliding window size is 20 seconds. 311 unique events compete for a spot in the top-10 ranks. We notice clear trends and the extreme volatility of the spots at lower ranks.

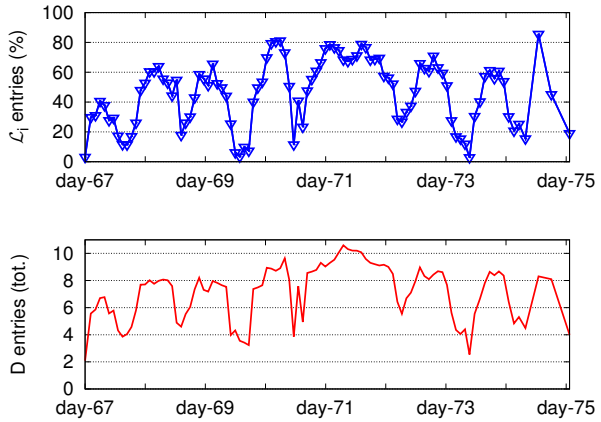

Figure 6: Cost of TOPiCo in terms of entries sent to other nodes as a function of the entries available in $\mathcal{L}_i$ (above) and as D entries (below).

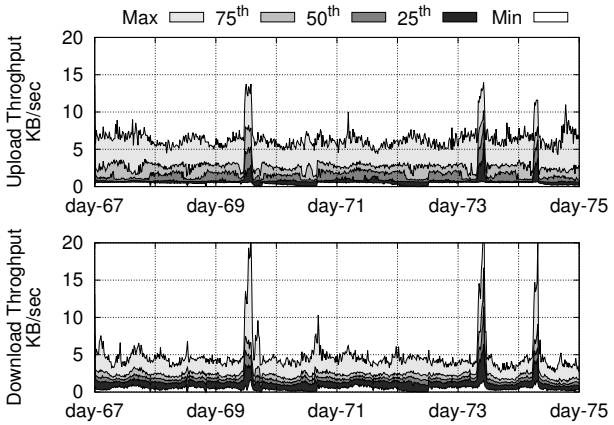

Figure 7: Throughput: upload (top) and download (bottom).

of the items in $\mathcal{L}_i$ need to be exchanged. Note that this is not a limitation of TOPiCo itself, but due to the intrinsic nature of the workload. In fact, as one can confirm in Figure 3, the difference in the frequency of the most popular items is small, implying that the number of candidates for $\mathcal{G}_i$ becomes potentially large. We expect TOPiCo to be able to significantly reduce the number of items sent when faced with more skewed workloads. Figure 6(bottom) complements this by showing, the number of additional items (D) that need to be sent. Even with this workload, the largest number of items exchanged is just 31 ($k + D = 20 + 11$) around day 71. Considering that for each item, we just send the item identifier and its frequency, the size of the exchanged list is still very small.

We confirm this by observing the download and upload throughput of sites, as shown in Figure 7. As expected, this is mostly affected by the arrival of events depicted in Figure 2. Both upload and download bandwidth usage remain under 10KB/sec most of the time which is quite small on modern infrastructures. The fact that few sites upload significantly more than the majority (notice the difference between the max and 75^{th} percentile on Figure 7) is because the reception of events is not uniformly spread across sites, causing some sites to maintain larger $\mathcal{L}_i$ and with more similar frequencies near the top, hence requiring to transmit more data.

We now focus on the closeness between $\mathcal{G}_i$, computed locally at each site, and $\mathcal{G}$ computed by an hypothetical omniscient observer. To this end we use the normalized Kendall-Tau rank distance which gives the pairwise differences between two ranked lists. A distance of zero means the lists are equal, while a distance of one means complete disagreement in the rankings. Figure 8 shows the normalized Kendall-Tau distance between $\mathcal{G}$ and $\mathcal{G}_i$. Here we split the servers according to their geographical location: Europe, US-EastCoast, US-Central and US-WestCoast. The reason for this is to observe how the geographical location, and hence, different access patterns affect the distance to $\mathcal{G}$. As shown, the distance to the omniscient $\mathcal{G}$ is roughly the same across all regions meaning TOPiCo achieves good results regardless of the composition of $\mathcal{L}_i$ s in a particular region.

Then, we investigate how the Kendal-Tau distance distance between $\mathcal{G}$ and $\mathcal{G}_i$ performs under peak hours. We choose the peak hours between day-69 and day-70 (Figure 2). In this time window, there are as many as 4,500 messages per minute. Figure 9 reports our result. We observe how TOPiCo performs similarly to the previous scenario, stating the benefits of our approach even under heavy load.

Finally, we show the delay between $\mathcal{G}_i$, computed locally at each site, and the real instantaneous $\mathcal{G}$. As before, we consider sites in different geographical regions. Results are presented at Figure 10 from day-67 to day-75. Again, the

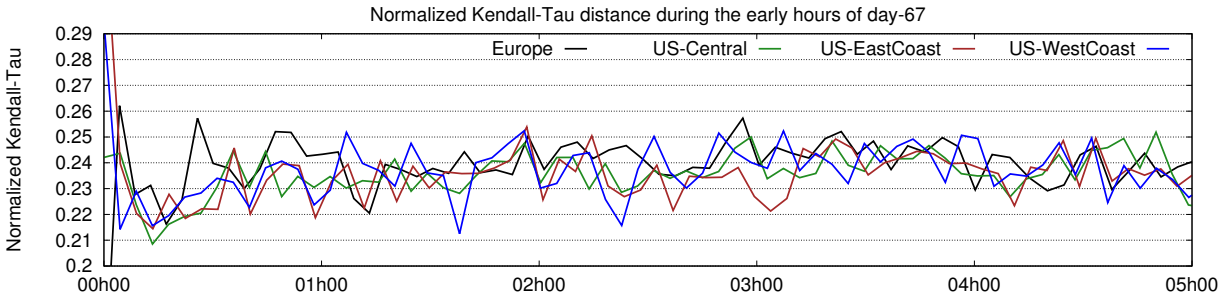

Figure 8: Kendall-Tau rank distance between $\mathcal{G}_i$ and $\mathcal{G}$ for 4 nodes in different regions. Results focus on the first 5 hours of day-67. A distance of 0 indicates identical rankings.

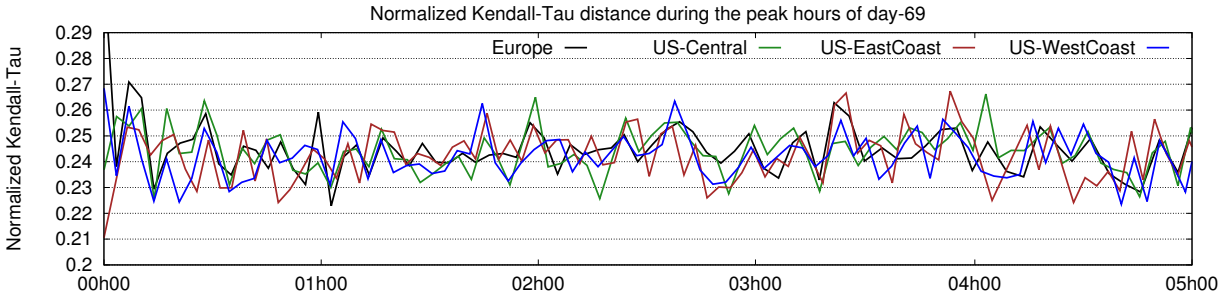

Figure 9: Kendall-Tau rank distance between $\mathcal{G}_i$ and $\mathcal{G}$ for 4 nodes in different regions. Results focus on the peak hours between day-69 and day-70.

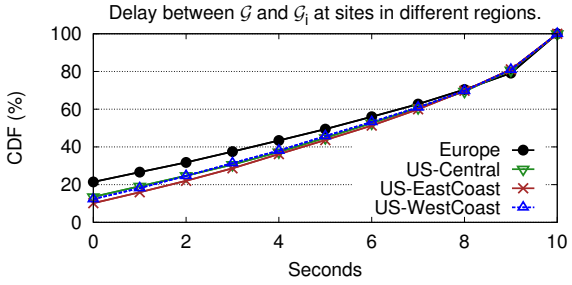

Figure 10: Distribution of delay between $\mathcal{G}_i$ and $\mathcal{G}$ for 4 nodes in different regions.

difference across regions is negligible meaning TOPiCo is able to dilute the local differences when computing $\mathcal{G}$. Half of the time, sites experience a latency smaller than the update period δ. Naturally, by decreasing δ we would observe a reduction in the latency at the expense of bandwidth.

Overall, the evaluation conducted allows us to conclude that TOPiCo delivers its promises: a lightweight, adaptable algorithm for computing the top-k most frequent items across a set of geographically distributed sites.

6. DISCUSSION AND CONCLUSION

In this paper we presented TOPiCo, a lightweight protocol for computing the top-k frequent items over several geographically dispersed event streams. TOPiCo works by selecting, locally at each site, the minimal amount of items that need to be exchanged such that each site is able to build a $\mathcal{G}_i$ close to the ideal $\mathcal{G}$ as observed by an omniscient observer. We provide a correctness proof of the algorithm and evaluate it in a real implementation with a real workload. The results confirm TOPiCo as a resource efficient approach able to adapt to variations in the workload.

In the present work we have not considered site failures. However, as TOPiCo relies only on local knowledge, and does not require any form of coordination among site, it is suited to work on an environment where sites may fail. Assessing TOPiCo behavior and any potential adjustments required to tolerate faults is part of our future plans.

Besides, we assume the existence of a broadcast primitive to disseminate information to all sites. The absence of an underlying multicast primitive results often in broadcast being implemented as a series of unicast calls, which limits the scalability of the system. We plan to overcome these limitations by extending TOPiCo to disseminate information using epidemic/gossip based protocols. Such protocols are known to be highly scalable but also robust to failures which aligns very well with the goals of TOPiCo we have in mind.

7. ACKNOWLEDGEMENTS

The authors are grateful to Flavio Junqueira for the initial discussions around this work. The research leading to this publication was partly funded by the European Commission's FP7 under grant agreement number 318809, LEADS project and 619606, LeanBigData, Ultra-Scalable and Ultra-Efficient Integrated and Visual Big Data Analytics project.

8. REFERENCES

[1] Arlitt, M., and Jin, T. A workload characterization study of the 1998 world cup web site. *Network, IEEE 14*, 3 (2000), 30–37.

[2] Babcock, B., and Olston, C. Distributed top-k monitoring. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2003), SIGMOD '03, ACM, pp. 28–39.

[3] Brenna, L., Gehrke, J., Hong, M., and Johansen, D. Distributed event stream processing with non-deterministic finite automata. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems* (New York, NY, USA, 2009), DEBS '09, ACM, pp. 3:1–3:12.

[4] Cao, P., and Wang, Z. Efficient top-k query calculation in distributed networks. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2004), PODC '04, ACM, pp. 206–215.

[5] Cormode, G., and Muthukrishnan, S. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms 55*, 1 (Apr. 2005), 58–75.

[6] Culhane, W., Jayaram, K. R., and Eugster, P. Fast, expressive top-k matching. In *Proceedings of the 15th International Middleware Conference* (New York, NY, USA, 2014), Middleware '14, ACM, pp. 73–84.

[7] Demaine, E. D., López-Ortiz, A., and Munro, J. I. Frequency estimation of internet packet streams with limited space. In *Proceedings of the 10th Annual European Symposium on Algorithms* (London, UK, UK, 2002), ESA '02, Springer-Verlag, pp. 348–360.

[8] Fagin, R., Kumar, R., and Sivakumar, D. Comparing top k lists. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2003), SODA '03, Society for Industrial and Applied Mathematics, pp. 28–36.

[9] Fagin, R., Lotem, A., and Naor, M. Optimal aggregation algorithms for middleware. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (New York, NY, USA, 2001), PODS '01, ACM, pp. 102–113.

[10] Guerrieri, A., Montresor, A., and Velegrakis, Y. Top-k item identification on dynamic and distributed datasets. In *Euro-Par 2014 Parallel Processing*, F. Silva, I. Dutra, and V. Santos Costa, Eds., vol. 8632 of *Lecture Notes in Computer Science*. Springer International Publishing, 2014, pp. 270–281.

[11] Guntzer, J., Balke, W.-T., and Kiessling, W. Towards efficient multi-feature queries in heterogeneous environments. In *Proceedings of the International Conference on Information Technology: Coding and Computing* (Washington, DC, USA, 2001), ITCC '01, IEEE Computer Society, pp. 622–.

[12] Hirzel, M. Partition and compose: Parallel complex event processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems* (New York, NY, USA, 2012), DEBS '12, ACM, pp. 191–200.

[13] Ilyas, I. F., Beskales, G., and Soliman, M. A. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv. 40*, 4 (Oct. 2008), 11:1–11:58.

[14] Lahiri, B., Chandrashekar, J., and Tirthapura, S. Space-efficient tracking of persistent items in a massive data stream. In *Proceedings of the 5th ACM International Conference on Distributed Event-based System* (New York, NY, USA, 2011), DEBS '11, ACM, pp. 255–266.

[15] Lahiri, B., and Tirthapura, S. Identifying frequent items in a network using gossip. *Journal of Parallel and Distributed Computing 70*, 12 (2010), 1241 – 1253.

[16] Manjhi, A., Shkapenyuk, V., Dhamdhere, K., and Olston, C. Finding (recently) frequent items in distributed data streams. In *Proceedings of the 21st International Conference on Data Engineering* (Washington, DC, USA, 2005), ICDE '05, IEEE Computer Society, pp. 767–778.

[17] Michel, S., Triantafillou, P., and Weikum, G. KLEE : A Framework for Distributed Top-k Query Algorithms. *VLDB '05 - Proceedings of the 31st VLDB conference* (2005), 637–648.

[18] Misra, J., and Gries, D. Finding repeated elements. *Sci. Comput. Program. 2*, 2 (1982), 143–152.

[19] Sacha, J., and Montresor, A. Identifying frequent items in distributed data sets. *Computing 95*, 4 (Apr. 2013), 289–307.

[20] Singh, S., Estan, C., Varghese, G., and Savage, S. Automated worm fingerprinting. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), OSDI'04, USENIX Association, pp. 4–4.

[21] Theobald, M., Weikum, G., and Schenkel, R. Top-k query evaluation with probabilistic guarantees. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (2004), VLDB '04, VLDB Endowment, pp. 648–659.

[22] Tudoran, R., Nano, O., Santos, I., Costan, A., Soncu, H., Bougé, L., and Antoniu, G. Jetstream: Enabling high performance event streaming across cloud data-centers. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems* (New York, NY, USA, 2014), DEBS '14, ACM, pp. 23–34.

[23] Vitter, J. S. Random sampling with a reservoir. *ACM Transactions on Mathematical Software 11*, 1 (1985).

[24] Wang, X., Candan, K. S., and Song, J. Complex pattern ranking (cpr): Evaluating top-k pattern queries over event streams. In *Proceedings of the 5th ACM International Conference on Distributed Event-based System* (New York, NY, USA, 2011), DEBS '11, ACM, pp. 279–290.

[25] Weigert, S., Hiltunen, M. A., and Fetzer, C. Community-based analysis of netflow for early detection of security incidents. In *Proceedings of the 25th International Conference on Large Installation System Administration* (Berkeley, CA, USA, 2011), LISA'11, USENIX Association.

[26] Wong, R. C.-W., and Fu, A. W.-C. Mining top-k frequent itemset from data streams. *Journal of Data Mining and Knowledge Discovery 13*, 2 (2006), 193–217.

Appendix B. Deployment and execution instructions

B.1. Deploying new virtual machines and micro-clouds

LEADS real-time processing platform and query engine are already deployed at several micro-clouds offered by Cloud&Heat, and at two micro-clouds under the control of two academic partners (University of Neuchâtel and TSI). For the sake of completeness, we now present the detailed steps of adding new virtual machines and micro-clouds to the platform. This process requires limited technical knowledge (basic knowledge of GNU/Linux).

Preparing a virtual machine. The step-by-step process is as follows:

1. First, install java 1.7 and vert.x 2.1.5 in the machine. Vert.x needs to be installed under $HOME/bin. Java can be installed anywhere as long as it is included in the path.
2. Copy file $HOME/bin/vert.x-2.1.5/bin/vertx to vertx1g, vertx2g, vertx500m.
3. Add the following line near the beginning of $HOME/bin/vert.x-2.1.5/bin/vertx1g:
   `JVM_OPTS="-XX:+CMSClassUnloadingEnabled -XX:-UseGCOverheadLimit -XX:PermSize=128m -Xmx1g -XX:+UseG1GC -server"`
4. Add the following line near the beginning of $HOME/bin/vert.x-2.1.5/bin/vertx2g:
   `JVM_OPTS="-XX:+CMSClassUnloadingEnabled -XX:-UseGCOverheadLimit -XX:PermSize=128m -Xmx2g -XX:+UseG1GC -server"`
5. Add the following line near the beginning of $HOME/bin/vert.x-2.1.5/bin/vertx500m:
   `JVM_OPTS="-XX:+CMSClassUnloadingEnabled -XX:-UseGCOverheadLimit -XX:PermSize=128m -Xmx500m -XX:+UseG1GC -server"`
6. Append the following line to the end of .bashrc
   `export VERTX_MODS=~/.vertx_mods`
7. Append the following line to the end of .profile
   `export PATH=$PATH:$HOME/bin/vert.x-2.1.5/bin/`
8. Place all jar files under $VERTX_MODS
9. Place all configuration files under $VERTX_MODS/conf
10. Add the following lines in /etc/security/limits.conf (requires sudo)
    ```
    *                soft  nofile           65000
    *                hard  nofile           65000
    ```

The platform is ready to start.

Starting the LEADS platform. The platform can be started automatically at all machines by executing the bootstrapper with the correct bootconf.xml (see Section B.3).

Alternatively, the platform can be started manually, as follows:

1. Create the configuration json file (denoted as <file.json> in the following)
2. Clear folders /tmp/leads* using: `rm -rf /tmp/leads*`
3. Change to folder ~/.vertx_mods
4. Run the following commands to start the engine (please run them in different terminals):
   ```
   vertx runMod gr.tuc.softnet~processor-webservice~1.0-SNAPSHOT -conf <file.json> -cluster
   vertx runMod gr.tuc.softnet~imanager-comp-mod~1.0-SNAPSHOT -conf <file.json> -cluster
   vertx runMod gr.tuc.softnet~log-sink-module~1.0-SNAPSHOT -conf log-sink.json  -cluster
   vertx runMod gr.tuc.softnet~deployer-comp-mod~1.0-SNAPSHOT -conf <file.json> -cluster
   vertx runMod gr.tuc.softnet~nqe-comp-mod~1.0-SNAPSHOT -conf <file.json> -cluster
   vertx runMod gr.tuc.softnet~planner-comp-mod~1.0-SNAPSHOT -conf <file.json> -cluster
   ```

Stopping the platform. The engine can be stopped with the following command: jps | grep Start | awk {'print $1'} | xargs kill -9

B.2. Loading data in the query engine

Populating the query engine with csv files.
We provide the load-csv utility, which can be executed as follows:

```
java -cp load-csv-1.0-SNAPSHOT-jar-with-dependencies.jar data.LoadCsv
[loadRemote|loadEnsembleMulti] pathToDirectoryWithCSVfiles
[IP:Port|EnsembleString] delayInMs
```

For example:

```
java -cp load-csv-1.0-SNAPSHOT-jar-with-dependencies.jar data.LoadCsv
loadRemote /home/ubuntu/cassandra_export_tuc 80.156.73.113:11222 0
```

will load all records from the CSV files in folder /home/ubuntu/cassandra_export_tuc, to a remote micro-cloud accessible at 80.156.73.113:11222. Each subsequent insert will delay 0 msec. Delaying could be necessary to not overload the machines.

EnsembleString can be used to load data on more than one micro-clouds. The string contains the list of pairs of IP addresses and ports for all Infinispan nodes participating in the Ensemble. A semicolon ';' is used as a delimiter between the IP addresses inside each micro-cloud, and '|' is used as a delimiter between the micro-clouds. For example, the following EnsembleString contains two micro-clouds, each with two VMs:

```
'5.147.254.199:11222;5.147.254.195:11222|80.156.73.113:11222;80.156.73.116:11222'
```

Populating the query engine with AMPLab data. The tool load-AMPLab can be used as follows:

```
java -XX:+UseG1GC -cp load-AMPLab.jar data.LoadAMPLab
[loadRemote|loadEnsembleMulti] pathToDirectoryWithKeyfiles
[IP:Port|EnsembleString] delayInMs numberOfRecords sizeOfRecords
```

As an example, the following command will create and load 1 million records of size 4000, according to the keys stored in ~/ dataAmpAll:

```
java -XX:+UseG1GC -server -cp load-AMPLab.jar data.LoadAMPLab2
loadEnsembleMulti ~/dataAmpAll/
'5.147.254.161:11222|5.147.254.199:11222|80.156.73.113:11222|80.156.222.4:1
1222' 0 1000000 4000
```

B.3. Using the Bootstrapper to configure and start the platform

To ease the process of configuring and starting the platform, the Bootstrapper tool automatically configures all platform components, and initiates/executes the components over all micro-clouds. The tool is provided as a compressed file, and includes the executable (a jar) and a set of required xml files.
The user needs to edit boot_configuration.xml, to include the micro-cloud IP addresses, micro-cloud credentials, and configuration related to HDFS, as follows.

HDFS configuration parameters

```
<hdfs>
        <uri>hdfs://address:port</uri> // address and port of the hdfs service
        <user>username</user> // hdfs username
        <prefix>file_system_prefix</prefix>
</hdfs>
```

Micro-cloud configuration

A micro-cloud includes several VMs. Each VM has as name a private IP (adress_1) and a public IP (address_2). The following example defines a micro-cloud with name dd1a that contains four VMs:

```
<MC name='dd1a' credentials="cloudandheat">
      <node name='leads-qe8' privateIp='10.130.0.16'>80.156.222.4</node>
      <node name='leads-qe14' privateIp='10.130.0.104'>80.156.222.23</node>
      <node name='leads-qe15' privateIp='10.130.0.122'>80.156.222.21</node>
      <node name='leads-qe22' privateIp='10.130.0.128'>80.156.222.31</node>
</MC>
```

To support automatic configuration of micro-clouds with different credentials, the tool requires authorization information for each micro-cloud. A pointer to the credentials is defined using the "credentials" attribute in the XML. Each credential is subsequently defined in the document, by including the full authorization details. Two types of credentials are supported: (a) with RSA files, and (b) with username/password.

```
<ssh>
      <credentials> // a set of credentials for RSA login
            <id>cloudandheat</id>
            <username>ubuntu</username>
            <rsa>/path/to/file.rsa</rsa>
      </credentials>
      <credentials> a set of credential for login using user/password
            <id>tsi</id>
            <username>ubuntu</username>
            <password>*********</password>
      </credentials>
</ssh>
```

The VM running the Bootstrapper needs to include in ~/.ssh/known_hosts the ssh keys for all VMs that will be running the engine. This is an inherent requirement of the ssh protocol, and even though bypassing it is possible (e.g., by setting StrictHostKeyChecking=no in ssh config file), it would pose security risks. Nevertheless, it is fairly straightforward to include all ssh keys in this file, by simply establishing a regular SSH connection once from the VM that runs the engine to each of the VMs that will participate in the engine. In this way, the known_hosts file is updated to include the ssh keys of all VMs.

Additional parameters

The configuration file includes additional parameters, for which the default values can be kept.

B.4. Running queries with the command-line interface

SQL queries can be executed using the LEADS command-line interface, which can be started as follows

```
java -jar leadscli.jar http://WebserviceIP:Port
```

with WebserviceIP:Port denoting the IP:Port of the web service, as given by the bootstrapper.