Cardinality Estimation in Shared-Nothing Parallel Data Flows

IT4BI Master Thesis
Joint thesis between the Database and Information Management Systems Group (DIMA) of the Technische Universität Berlin and the Universitat Politècnica de Catalunya
by
Tamara Mendt

Advised by:

Alexander Alexandrov
alexander.alexandrov@tu-berlin.de

Asterios Katsifodimos
asterios.katsifodimos@tu-berlin.de

Alberto Abelló
aabello@essi.upc.edu

Friday 31st July, 2015.

Cardinality Estimation in Shared-Nothing Parallel Data Flows

Tamara Mendt

Universitat Politecnica de Catalunya
tamara.desiree.mendt@est.fib.upc.edu

Abstract. Shared nothing parallel dataflow systems aim to bridge the gap between MapReduce and RDBMSs by combining parallel execution of second order functions with operator based optimizations. In parallel systems, job latency is strongly affected by data shuffling and unbalanced data across nodes, thus the degree of parallelism and the data partitioning functions must be carefully considered when choosing optimization strategies. However, it is hard to make good optimization choices without any information about the distribution of the data. We attempt to overcome this challenge in shared nothing parallel dataflows by tracking statistics of data sets during query runtime. We use data streaming algorithms to track statistics so as to affect job latency as little as possible. We discuss how collected statistics can potentially be used to improve execution plans during runtime.

1 Introduction

The shift from the relational model. In the past, database management systems (DBMSs) were designed and optimized to store data following a relational model and to query it using relational algebra. However, this model requires fitting data into predefined schemas and is not well suited for performing complex transformations over heterogeneous data. In 2004, Google Inc. presented a parallel programming model and massively-scalable execution framework under the common name MapReduce [11] (Appendix A.2). The programming model, based on second-order functions, allows developers to execute a single data processing job, on many machines running in parallel, sharing no hardware, and connected through a high-speed network (shared nothing architecture).
MapReduce shortcomings. MapReduce achieves fast query processing by data-parallel application of user defined functions (UDFs). Users have the flexibility to perform per element and per group data transformations that are executed in a massively parallel way, using second order *map* and *reduce* functions. However, MapReduce fails when a user tries to fit a more complex processing pattern into map and reduce functions (e.g. multi input operators like join or cross), or when map and reduce functions are fit to perform transformations for which a high-level language exists (e.g. SQL single input aggregates like projections). These limitations of the MapReduce framework have driven recent developments in database technologies to bridge the gap between the ability to

parallelize tasks offered by the MapReduce framework and the expressiveness and ability to make operator based optimizations offered by RDBMSs[19]. Examples of these initiatives are Microsoft SCOPE [24] (from now on SCOPE), Apache Flink (from now on Flink) [3], and Apache Spark (from now on Spark) [22]. In particular, SCOPE and Flink have built-in cost based query optimizers which use similar reasoning techniques as RDBMS optimizers in order to create improved distributed execution plans.

Optimization in shared-nothing architectures. In shared-nothing architectures, data shuffling through the network is one of the costliest operations. Ideally, data transfer between nodes should be minimized. However, key-based operators such as join and group-by, often require data to be re-partitioned based on the operator key (so shuffling becomes inevitable). In addition to minimizing data shuffling, a crucial optimization goal in parallel executions is to keep data balanced across work nodes. This is because the overall execution is bounded by the slowest task, i.e. the worker processing the largest partition. However, in the presence of data skew, optimizers often fail to balance data evenly. In order to make smarter choices, optimizers require information about the distribution of the data flowing through operators.

Cardinality estimation and cost based optimization. Cost based query optimizers rely on estimations of the size of intermediate data sets in order to choose efficient execution plans. The cost of a single operator in a query is mainly determined by the size of the data input to the operator and the value distribution on the active key domain. Wrong statistics or oversimplifying assumptions lead to inaccurate estimations and hence to the selection of a highly sub-optimal execution plans.

Statistics collection in shared-nothing parallel dataflows Obtaining accurate statistics of intermediate data sets in a query is not trivial because data is transformed at every operator in the query. RDBMSs estimate statistics of intermediate data sets by propagating base statistics through operators (Appendix A.1). However, in parallel data flow systems, statistics propagation is very challenging given the unexpected behavior that user defined code (in the form of UDFs) introduces. Since parallel dataflow systems often materialize data at points in the dataflow right before re-partitioning, it would make sense to track statistics of intermediate results as a side effect of the data scan performed for materializing. Statistics that are collected during query runtime will be much more accurate than what can be obtained by trying to propagate base statistics through UDFs. This sets the motivation for our work.

Contributions. The contributions of our work are twofold.

(1) We perform cardinality estimation for parallel dataflows running in Apache Flink. For this purpose we collect statistical information (min, max, count distinct, heavy hitters) representing the distribution of the data flowing through operators. Since we do not wish to impact job latency or increase memory requirements of the query execution, we rely on the use of streaming algorithms to collect statistics. Though streaming algorithms sacrifice some accuracy, they obtain good estimates while requiring only one scan of the data and little mem-

ory and processing resources. We incorporate two streaming algorithms into the Apache Flink runtime to estimate count distinct (Linear Counting and HyperLogLog), and two streaming algorithms to detect heavy hitters (Lossy Counting and Count Min Sketch). We measure the overhead associated to the statistics collection and compare the two algorithms for each statistic against each other.

(2) We present proposals for leveraging the collected statistics for query optimization. We focus mainly on the degree of parallelism and data partitioning strategies.

2 Related Work

Query re-optimization in RDBMSs Kabra and Dewitt [15] were the first to propose re-optimizing queries in RDBMSs, based on actual statistics collected at query runtime (in 1998). The authors propose to re-optimize complex queries by pausing query execution before a given operator, materializing the intermediate results, and optimizing the remaining subquery with the collected statistics. In this approach, pre-defined rules determine the *innacuracy potential* of query operators which is then used to assess whether statistics collection should be be performed or not. Collected statistics are based on histograms. Stillger et al. [20] propose re-optimization of future queries based on statistics collected during runtime. This approach assumes workloads of similar queries that will execute often, and hence can benefit from statistics of past query executions.

Query re-optimization in shared-nothing parallel dataflows Bruno et al. [6] and Agarwal et al.[2] have tackled query re-optimization during runtime in the shared-nothing, parallel dataflow system SCOPE (Scalable Computation of Parallel Execution). In parallel dataflow systems it is common to have complex, long running queries for which the overhead of re-optimizing can be negligible with respect to the entire query runtime. In SCOPE, data properties are collected at a task level using data streaming algorithms. These statistics (along with information regarding which intermediate data sets have been materialized) are piggybacked to the job manager which feeds them to the query optimizer. SCOPE uses runtime statistics to improve operator cost estimations (see Appendix B), however the system does not use statistics to make choices regarding degree of parallelism or partitioning functions. To our knowledge, SCOPE is the only parallel dataflow system to collect statistics during query runtime and use them to re-optimize running queries.

SCOPE: incorporating partitioning into dataflow optimization Zhou et al. [25] are some of the first authors to incorporate *partitioning* and *merge* as operators into the reasoning of a parallel dataflow optimizer (SCOPE). The authors analyze the effect that the merge and partition operators have on structural data properties (distribution, sorting and grouping) and use this information to optimize parallel execution plans.

SCOPE: reducing data shuffling More recent studies by Zhou et al. [23] discuss how to leverage information about data distribution in order to reduce data transfer. The authors propose re-partitioning strategies that do not require

sending the entire data set over the network, but rather a subset of it. This partial re-partitioning can be achieved by exploiting dependencies between partitioning functions or between data fields. An example of partial re-partitioning is to carefully selecting boundaries for range partitioning on a key when it follows range partitioning on a pair composed of the same key together with some other key.

SCOPE: adapting partitioning based on data skew The latest studies in SCOPE [7] propose using join strategies that can reduce the need for entire re-partitioning when data skew is detected. The proposal involves detecting high frequency values on either side of the join, and selecting a different partitioning strategy for non frequent values (broadcast) than for frequent ones (partition).

Optimization in Flink Like SCOPE, Flink includes a built in optimizer capable of making cost-based selection of parallel execution strategies[3]. One fundamental difference between Flink and SCOPE is that, while SCOPE always chooses to materialize intermediate data before performing data shuffling, the Flink optimizer will avoid materialization when possible and instead pipeline early results in a similar way to RDBMSs [5]. Optimization in Flink includes re-use of partitions and sort orders across operators, hash join vs. sort-merge join, partition join vs. broadcast join, among others. The Flink optimizer does not consider join re-ordering or index vs. table scan selection.

Optimization in Spark Whereas Flink treats data like a continuous stream, in Spark data is abstracted in the form of distributed collections (RDD) [22]. Transformations on these collections are performed in-memory, thus intermediate collections are not materialized to disk. Spark does not include a built-in operator based optimizer. Optimization has mainly targeted column compression to reduce the size of data transfer over the network and optimization of SQL to Spark programs [4].

3 Conceptualization

3.1 Architecture of a shared-nothing distributed dataflow

The architecture of a shared-nothing distributed dataflow generally consists of a client which submits a data job to a node which will act as job manager (also called job tracker or driver). The job manager is the only node with a complete overview of the job execution and is in charge of scheduling parallel tasks to be executed by task nodes (called task managers, task trackers or workers).

Task managers can exchange intermediate data sets among each other if the job manager indicates to do so. Data exchange between task managers can occur in a "pull" fashion (data stays at the task where it is processed and awaits to be fetched by the next task manager) or "push" fashion (data is directly sent to the next task manager that must process it). This depends on each system. Figure 3.1 shows a high level schema of the shared-nothing distributed dataflow architecture.

Since task managers execute data transformations, it is at the task level where we will perform statistics collection. The statistics collected at each task

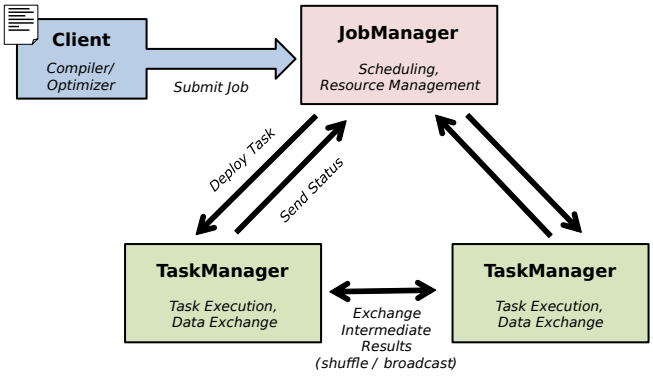

Fig. 1. Architecture of a shared-nothing parallel dataflow environment [1]

are then sent to the job manager who merges them to obtain a global view of the data distribution.

3.2 Statistics collection in a distributed environment

Key-based operations (e.g. group by or join) in parallel shared nothing dataflows require partitioning data based on the operator key. When the distribution of the key values is skewed, key-based partitioning is likely to result in very unbalanced partitions. Our proposal is that by tracking statistics over keys of downstream key-based operators, we can detect skew at an early stage and use this information to improve a query execution plan. In order to solve our problem we require the following conceptual structures:

- A **state** s
  Representing the local state of a worker, or global state for a job. This state contains the statistical information of interest, i.e. statistics of downstream operator keys.
- An **update function** $u : (t, s) \mapsto s'$
  Applies some transformation on an incoming tuple t to derive the key for a downstream operator and updates state s with the information, resulting in an updated state s'.
- A **compose function** $c : (s_1, s_2) \mapsto s'$
  Composes two local states into a global one, representing information from both input states. This function should be commutative and associative.

The importance of state compositionality. Since in a distributed environment data is partitioned across nodes, at each node there is only access to a subset of the entire data set. Because of this we require that states be composable, i.e. it must be possible to define a compose function which derives a merged state equal to the state that we would obtain if all tuples from both partitions had been used to update that state. To illustrate this, consider a state containing only the count distinct of a given key. It is not possible to compose this state

with another state without additional information about the key set upon which count distinct was derived. It could be that the key set in one partition is contained in the key set of the other, in which case the count distinct of the merged key set should be the minimum count distinct among both states. However, it could also be the entire opposite case: the key sets from both partitions may be disjoint, in which case the count distinct of the global key set should be the sum of the count distinct of both partitions. We require that our "statistics state" contains enough information so that it can be composed properly with the state of another partition.

Memory requirements should be sub linear to cardinality. Given that parallel data flows must scale to very large data sets, we are likely to deal with very large cardinalities. Due to this it is necessary that states have memory requirements which are sub linear in the cardinality of the data set, otherwise we are likely to run out of memory. For example, keeping a list of every element in the data set to keep track of the frequency of each element is not possible since this would memory linear in the size of the data set.

Given the sub-linear memory requirement, and the fact that we wish to impact job latency as little as possible, we choose to use **data streaming** algorithms for our statistics collection. These algorithms are designed to find sufficiently accurate statistical estimates while requiring little memory, low processing times and only one pass over a data set.

3.3 Communication of statistics from tasks to job manager

We consider two approaches to communicating statistics between task managers and job manager:

- Send statistics to job manager upon completion of execution of a task along with task status update message (finished). This approach has the advantage of only requiring one merge phase for every task, but will only provide statistical information of data that has been entirely processed by a task.
- Piggy back statistics on the task heartbeat, which is a message that is sent periodically from the task managers to the job manager to indicate that they are still alive. The advantage of this approach is that it provides early statistics of data which can be important in systems that pipeline. However, this approach will require applying the merge function much more often, which can translate into significant overhead.

4 Statistics Collection

In this section we present the statistics that we collect on operator keys and in section 7 we discuss how these statistics might be leveraged to reduce query execution times.

4.1 Min/Max

Collecting the min and max values in a data set with an ordered domain is trivial. This statistic informs the range on which the values in a data set are contained and can hence be used to determine ranges for range-partitioning functions. Min and max are clearly **composable** since merging local values into a global one only requires comparing the values and keeping the overall minimum/maximum.

4.2 Count Distinct

The number of distinct elements in a data set can give us a good clue about the distribution of the data. If the number of distinct elements in a data set is close to the total cardinality of the set, we know that we are dealing with a data set in which most elements appear only once. On the contrary, if count distinct is close to 1, we know that we are dealing with a data set in which some elements are very frequent. Flajolet et al.[13] introduced the use of probabilities to reach approximate values for count distinct while using memory sub linear in the size of the data set. We incorporate two probability based algorithms for count distinct as part of our statistics collection in the Flink runtime.

Linear Counting. The basic idea behind Linear Counting[21] is to map each incoming element to a position in a bitmap. When the algorithm starts, the bitmap has all its elements set to "0". Every time an element maps to a given position, its bit is set to "1". The number of distinct elements in a data set is estimated by counting the number of bits still set to "0", after all elements have been mapped. For a detailed description of the algorithm refer to Appendix C.1. Our choice to use Linear Counting despite the fact that it has memory requirements linear in the size of the data set, was based on studies made by Metwally et al. [17] which showed that Linear Counting could provide high accuracy, while still requiring a reasonably low amount of memory.

HyperLogLog. HyperLogLog [12] uses a creative way to track count distinct in which each incoming element is mapped to a fixed-length array of bits. The first $\log_2 m$ bits of the array are used to assign the element to one of m buckets, where m is a power of 2. The algorithm then looks at the remaining bits of the array and tracks the number of leading "0's". The intuition behind this is that the probability of an array of random bits having k leading "0's" is 2^{-k}, so the algorithm uses the maximum number of leading "0's" among all bit arrays to estimate how many different elements have been seen. For a more detailed explanation of the algorithm refer to Appendix C.2. We choose to incorporate HyperLogLog to the Apache Flink runtime since it is a very well known algorithm for cardinality estimation and, as its name implies, its memory requirements are of order $O(\log_2 \log_2 n)$ (where n is the maximum expected count distinct).

Table 1 provides a summary of the properties of both count distinct algorithms. Linear Counting receives a parameter b, which is the size of the bitmap. Larger bitmaps are less likely to experience collisions, so they provide more accuracy while requiring more memory. HyperLogLog takes a parameter $\log_2 m$

	Linear Counting	**HyperLogLog**
Parameters	$b \rightarrow$ size of bitmap	$\log_2 m \rightarrow m$ is number of buckets
Accuracy	$O(1/\sqrt{n})$	$O(1.04/\sqrt{m})$
Memory	$O(n)$	$O(\log_2 \log_2 n)$

Table 1. Main properties of count distinct streaming algorithms

where m is equal to the number of buckets to which the algorithm sends incoming elements. Accuracy for Linear Counting depends on the ratio between the real cardinality and the size of the bitmap; when this ratio is low, the accuracy is in the order $O(1/\sqrt{n})$. In HyperLogLog accuracy depends on m, the number of buckets in which incoming elements are partitioned.

Compositionality. In the previous section we have mentioned that compositionality is a very important property for statistics collection in a distributed environment. Both Linear Counting and Hyperloglog are composable. Given two or more count distinct sketches, they can be merged into one which is exactly the same sketch that would have been obtained, had the elements for each merging sketch been tracked by only one sketch. The merge function for Linear Counting is a simple OR between bitmaps. This of course requires that both bitmaps be the same length and be generated by the same hash function. HyperLogLog sketches are merged by selecting the maximum number of leading "0's" for every bucket. This merge function requires that merging sketches have the same number of buckets m, and have been generated using the same hash function.

4.3 Heavy Hitters

The heavy hitters in a data set are elements whose frequency exceeds some defined threshold, or percentage of the total cardinality. Detecting heavy hitters in a data set can help detect data skew. This is important when data is required to be key-partitioned, since skew generally causes key-based partitions to become unbalanced. In our work we incorporate a count based algorithm, Lossy Counting [16], and a sketch based algorithm, Count-Min Sketch [10][9] to detect heavy hitters in the Apache Flink runtime.

Lossy Counting. The idea behind Lossy Counting is to track elements in memory, along with a lower and upper bound for their frequency. For a period of elements (determined by the error of the algorithm), all elements are stored in memory. Each time an element that is already in memory is seen, its lower bound is increased by one. At the end of the period, elements whose upper bound is not above a minimum frequency are removed from memory. For a more detailed explanation refer to Appendix C.3. We chose to implement Lossy Counting following the fact that it was successfully implemented in the SCOPE system (Appendix B).

Count Min Sketch The Count Min Sketch algorithm stores a matrix of counters in memory and assigns a different hash function to every row of the matrix. Incoming elements are hashed by each function to a counter in every row.

The algorithm estimates the frequency of an element as the minimum counter that the element maps to in the matrix. For more details about the Count Min Sketch algorithm refer to appendix C.6. Count Min Sketch allows estimating the frequency for any element that appears in a data stream. This means that it provides statistical information for any element in a data set, rather than only of the heavy hitters. This motivates our choice to incorporate Count Min Sketch to our statistics collection.

Table 2 provides a summary of the properties of Lossy Counting and Count Min Sketch. We use f_i to represent the real frequency and $\hat{f}_i$ to represent the estimated frequency of an item i. Both algorithms require a parameter s, the fraction that indicates the frequency threshold for an element to be considered a heavy hitter. The algorithms also require specifying a parameter ϵ, the error tuning the algorithms' accuracy. The smaller the error, the closer the estimated frequencies are to the real item frequencies. Count Min Sketch takes an additional parameter δ representing the confidence in frequency estimations. While Lossy Counting is guaranteed to underestimate frequencies by at most $\epsilon.n$, Count Min Sketch guarantees that it will always overestimate, and that the probability that the estimate error exceeds $\epsilon.n$ is bounded by $(1-\delta)$. The number of rows of the Count Min Sketch matrix grow as a function of δ.

	Lossy Counting	**Count Min Sketch**
Parameters	– $s \rightarrow$ fraction for heavy hitter – $\epsilon \rightarrow$ error	– $s \rightarrow$ fraction for heavy hitter – $\epsilon \rightarrow$ error – $\delta \rightarrow$ confidence
Accuracy	– All elements with $f_i > s.n$ will be output – No element with $f_i < (s-\epsilon)n$ will be output – algorithm **UNDER**estimates by at most $\epsilon.n$ ($f_i - \widehat{f_i} < \epsilon.n$)	– $P((\hat{f} - f) > \epsilon.n) < (1-\delta)$ – algorithm **OVER**estimates

Table 2. Properties of Lossy Counting and Count Min Sketch

Count Min Sketch extension for heavy hitters The original version of Count Min Sketch is designed to estimate the frequency of any element, given the element. Since we wish to track heavy hitters, it is necessary to keep an additional list containing the heavy hitters. We implement an extension to Count Min Sketch (Appendix C.7) in which every time a new element is seen, it is first mapped to the frequency matrix. Afterwards, if its estimated frequency is higher than the desired threshold, it is either stored in the heavy hitters list, or if it was already in the list, its frequency estimate is updated. Periodically, any element whose counter is under the desired threshold is removed from the list.

Compositionality. Composing heavy hitter statistics is not trivial since it is possible to have globally frequent that are not frequent in all local partitions.

However we can guarantee that if an element is globally frequent, it will be frequent in at least some local partition. We prove this by contradiction.

Proof. We use integers in the range $[1, p]$ to identify partitions. Suppose an element i has global frequency (f_i) greater than or equal to $s.n$. Now suppose that the element is not frequent in any local partition, i.e.

$$\forall j \in [1, p] : f_i^j < s.n_j$$

If we sum the frequencies over the j partitions we will obtain the global frequency. Summing $s.n_j$ over the j partitions we obtain $s.n$:

$$\sum_{j \in [1,p]} f_i^j = f_i < \sum_{j \in [1,p]} s.n_j = s.n$$

This is clearly a contradiction since our initial assumption was that $f_i \geq s.n$.

Compositionality Lossy Counting. We implement a distributed version of Lossy Counting (appendix C.4) in which local results are merged by adding local estimates (lower bound and upper bound respectively) for the local heavy hitters, and keeping only heavy hitters with upper bound higher than $\epsilon.n$ (n is the sum of local cardinalities). We observe that if an element is present in only one of the merging Lossy Counting structures we will ignore any appearances of the element in the other partition, even though in reality the element could have had frequency $\epsilon.n_i$, where n_i is the cardinality of the partition where the element was not tracked. Thus when merging, it is necessary to add this error to the upper bound. However, estimates in Lossy Counting are determined by the lower bound. So while merging estimates in this fashion increases the global range for the frequency of an element, the elements' global estimate will only be derived from local estimates where the element was tracked. Due to this, the larger the number of partitions, the further off the global estimated frequency will be from the real frequency. We prove however (appendix C.5) that even though the error may grow, it will continue to be bounded by $\epsilon.n$. Given this fact, and the fact that a global heavy hitter will be frequent in at least one local partition, we can guarantee that our distributed version of Lossy Counting will detect all heavy hitters.

Compositionality Count Min Sketch. Count Min Sketch for estimating frequencies is composable, i.e. given two Count Min Sketch matrices, summing the matrices will result in a matrix that estimates the frequencies of the merged set of items with the same error as the two original matrices. For our Count Min Sketch Extension for heavy hitters, we implement a distributed version in which two extended Count Min Sketches are merged by first merging the Count Min Sketch matrices and then iterating over the set of heavy hitters from both merging sketches. For each heavy hitter, the new estimated frequency is looked up in the merged matrix. The element continues to be tracked as a heavy hitter only if its estimated frequency exceeds the newly calculated threshold (with the sum of the cardinalities from both sets). Given the proof that globally frequent elements will be frequent in at least one partition, we can guarantee that distributed Count Min Sketch will find all heavy hitters.

5 Implementation

We implement our statistics collection in the Flink [1] shared-nothing parallel dataflow system. The motivation for choosing this system is the fact that it is an open source project, and that it has a built-in optimizer which could potentially benefit from statistics of distribution of data flowing through UDFs.

5.1 The Flink Job Life Cycle

When the Flink system is started, the job manager is initiated along with one or more task managers. The job manager is the coordinator of the system, and the task managers execute parts of parallel programs. Communication between the client and job manager, and the job manager and task managers is performed asynchronously using actor based concurrency (Akka library) [1].

Flink jobs are defined as operators connected in a directed acyclic graph (DAG). Each operator has properties, like the degree of parallelism and the code of the transformation it performs. The job manager transforms the **Job Graph** into a parallel execution graph, i.e. for each job graph vertex, it generates an execution vertex per parallel subtask. An operator with DoP n will have one **Job Vertex** and n **Execution Vertices**. The Execution Vertex tracks the state of execution of a particular subtask.

Each task manager offers one or more task slots (generally as many as the number of CPU cores), each of which can run one pipeline of parallel tasks. A pipeline consists of multiple successive tasks, such as the n-th parallel instance of a map function together with the n-th parallel instance of a reduce function. The way Flink deals with these pipelines, is by chaining operators in a recursive fashion.

5.2 Accumulators

Flink implements a class called **RuntimeContext** (Appendix D.5) which contains information about the context in which UDFs are executed, e.g. current parallelism or broadcast variables. During runtime, some types of UDFs can access and update their runtime context. These UDFs belong to a class of functions defined as **Rich Functions** (Appendix D.3).

The runtime context of a function also contains a Map of **Accumulators**. As their name implies, accumulators are classes that can accumulate information across functions. Each parallel instance in a job execution creates and updates its own accumulator objects, and the different parallel instances of the accumulator are merged by the system at the end of the job.

Accumulators are identified by a String, which is the key that is used to store them in the runtime context Map of accumulators. This key allows associating accumulators from different execution instances so that they can be merged into a global value. Typically, accumulators are initialized in the initialization method

[1] http://akka.io/

of a UDF, and are updated using the method *add* inside of the body of the UDF. The example in appendix D.2 shows an example code of a map function incorporating accumulators. UDFs are coded by extending abstract functions supported by Flink.

At the of the execution of a UDF, accumulators from that function, and functions chained to it through a pipeline are gathered and reported to the job manager. The job manager then merges incoming accumulators that are identified with the same id. Appendix D.1 shows how Flink classes work together to send accumulators from Tasks to the Job Manager.

In our implementation, we take advantage of the communication infrastructure which accumulators provide. We implement an accumulator called OperatorStatisticsAccumulator designed to store the statistics that we wish to track. To comply with the accumulators design pattern, we create a single class OperatorStatistics which holds all of the statistics that we wish to track, i.e. min, max, count distinct and heavy hitters. This class is the local value in our custom accumulator. Due to the fact that accumulators are sent upon task completion, our statistic collection will follow approach number two from the communication schemas that were explained in section 3.3.

5.3 Implementation Specifics

Apache Flink is an open source project under the Apache License and can be found in Github. Flink is entirely implemented in Java and Scala so it runs on the JVM. Our implementation can be found as an open pull request [2] to the Flink source code on Github.

As part of our development we have made use of a data streaming library called stream-lib [3] which is available under the Apache 2 License. We have used the following implementations from stream-lib:

- The **abstract class ICardinality** a a field to track count distinct in our OperatorStatistics (D.4) class.
- The implementation of **Linear Counting**.
- The implementation of **HyperLogLog**.
- The implementation of **Count Min Sketch**.

Our implementation efforts are summarized as follows:

- Extension of the structures offered in the stream-lib library, with custom serialization functions since not all structures in stream-lib are serializable.
- Implementation of Lossy Counting.
- Implementation of Count Min Sketch extension for heavy hitters.
- Extension of the class Accumulator for collecting our statistics.
- Accumulator configuration class to allow specifying the algorithms to use for statistics collection and their parameters.

Details of the classes used in our implementation can be found in Appendix D

[2] https://github.com/apache/flink/pull/605
[3] https://github.com/addthis/stream-lib

6 Experimental Results

In this section we evaluate the performance of our statistics collection algorithms in terms of processing overhead and accuracy. We evaluate processing overhead while fixing algorithms parameters to default values and scaling out in number of nodes (section 6.1). We then evaluate the effect of varying algorithm parameters on processing overhead (section 6.2). Finally we analyze the effect of varying algorithm parameters on accuracy (section 6.3).

We ran our experiments on a cluster of 20 machines (8 cores, 16GB RAM) running Flink v0.9.0. The data for experiments is generated in parallel and stored in the HDFS file system. For experiments measuring processing overhead we report median runtime over 7 executions. For accuracy experiments we only perform one execution, since outcome is deterministic.

6.1 Processing Overhead (Weak Scale Out)

We wish to answer the question of how much processing overhead our statistics collection algorithms cause. For this purpose we will measure the processing overhead in a job which collects statistics compared to runtime of the same job but with no statistics collection. We perform weak scale out, i.e. increase number of nodes while keeping a fixed amount of data per node. We perform a job which does not require data shuffling to avoid accounting for overhead due to data transfer. We expect that keeping data per node constant, will keep processing times constant while scaling out. If this were not the case for the jobs including statistics collection we can attribute extra overhead while scaling out to the statistics merge phase.

Experiment Setup We run a job consisting of a FlatMap followed by a Filter function (no shuffling). We measure runtime when using no statistics collection (labeled none), and compare it to runtime when collecting each of the statistics that we have implemented: min/max, Linear Counting (linearc), Hyperloglog (hyperloglog), Count Min Sketch (countmin), and Lossy Counting (lossy). In addition, we run an experiment using a histogram accumulator to compare the overhead of collecting real heavy hitters versus the overhead of estimating heavy hitters. We run experiments on **5, 10 and 20** nodes, with 2 Billion integers per node. For our data sets we use uniform (range [1,10.000]), gaussian (range [-20,20]), and pareto (range [1,10.000]) data distributions. We use the following parameters for our data streaming collection algorithms:

- **Linear Counting**: bitmap size $\rightarrow 10^6$
- **HyperLogLog**: buckets $log_2 m \rightarrow 10$
- **Heavy Hitters:**
  - fraction to determine threshold $s \rightarrow 0.05$
  - error $\epsilon \rightarrow 0.0005$
  - **Count Min Sketch**: confidence $\delta \rightarrow 0.99$

Results Discussion

In order to quantify overhead, we plotted the overhead percentage that our statistics collection algorithms represent with respect to the runtime of the job with no statistics collection. Figure 2 shows the percentage overhead for statistics collection of min/max and count distinct.

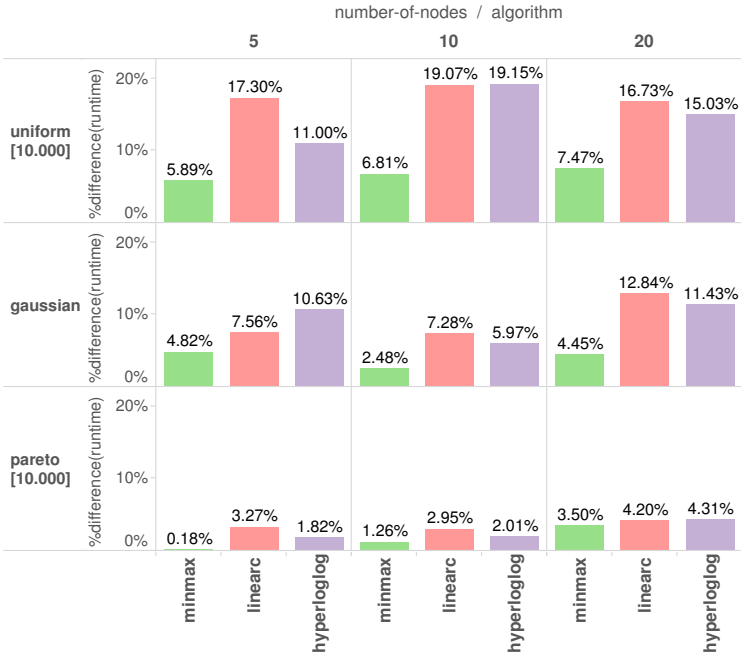

Fig. 2. Percentage difference of min/max and count distinct collection overhead with respect to no statistics collection

The following observations are considered of interest:

- The overhead of for count distinct represents at most 19%.
- The overhead of collecting min/max represents at most 7%.
- Linear Counting shows to have higher overhead than hyperloglog when dealing with uniform distribution. We believe that this is due to the size of the bitmap, and the fact that uniformly distributed data will requiring randomly accessing all bits spread through the whole bitmap. This could be responsible for causing cache misses.
- For a very skewed data distribution (pareto) we observe that the overhead of collecting count distinct reaches at most 4%. This is a positive outcome since in reality it is common to be dealing with skewed data.

In figure 3 we plot the percentage overhead for statistics collection of estimated heavy hitters and true item frequencies (histogram).
The following observations are considered of interest:

- The overhead of heavy hitter collection is very high for uniform data distribution (around 50%).

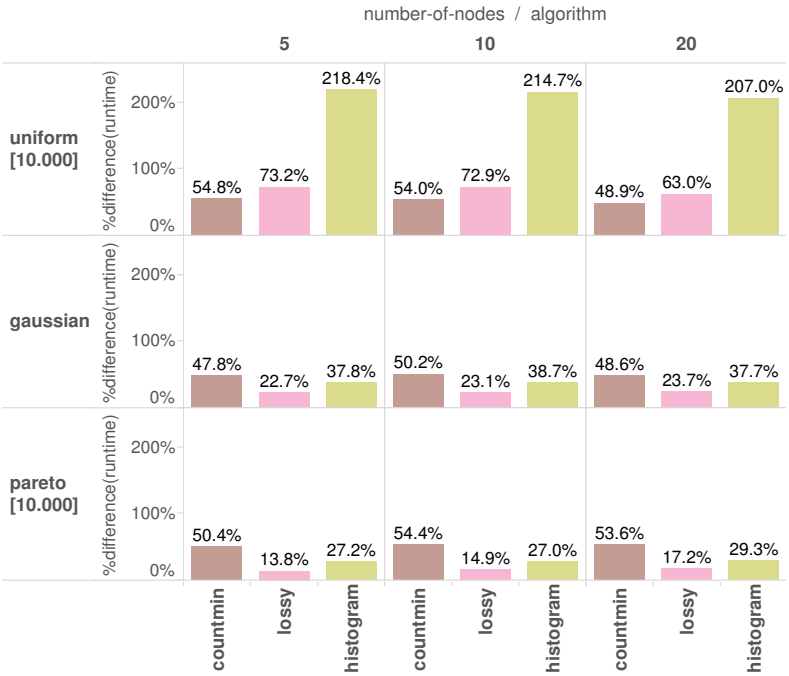

Fig. 3. Percentage difference of heavy hitter and histogram collection overhead with respect to no statistics collection

- The overhead of collecting exact statistics (histogram) is enormous for uniform data distribution (200%) but for data with skewed distribution (pareto) or small range (gaussian), collecting a histogram causes less overhead than collecting heavy hitters using countmin. We might be tempted to think that it would be preferable to use histogram to collect statistics, however it is important to take into account that histogram would require an amount of memory linear to the domain of the key that we are tracking.
- When data is distributed uniformly, lossy has a much greater impact on performance than countmin (70% vs. 45%). This is due to the fact that lossy stores all elements per period in memory and wipes non-frequent elements out periodically. Uniform data distribution will cause new elements to constantly be stored and then wiped out again.
- The overhead of collecting heavy hitters with countmin is not strongly affected by data distribution. This is due to the fact that overhead in countmin is dominated by the mapping of an element to the matrix of counters. The overhead for countmin sketch represents around 50%, independently of the data distribution.

Figure 4 shows a joint graph representing the overhead generated by our count distinct and heavy hitter algorithms when compared to no statistics collection. The bars represent median runtime in seconds. We can summarize the results of this experiment as follows:

- No significant difference can be observed in overhead when scaling out in number of nodes. This means that the overhead generated by the merging is not significant, so it would be possible to implement early statistics collection, rather than sending statistics when tasks finish executing.
- Heavy hitter statistics collection causes considerably higher overhead than count distinct.
- Overhead for all algorithms except countmin is strongly bound to data distribution.

6.2 Processing Overhead Varying Algorithm Parameters

In order to quantify how the overhead of our algorithms are affected by the algorithm parameters, we setup experiments to run a job on 20 nodes with a total data set size of 40 Billion integers (2B per node) and varied the algorithm parameters.

Count Distinct: Scale Parameters

Results Discussion Figure 13 (appendix E) shows the results for this experiment. The bars represent the median runtime and the labels indicate the percentage difference with respect to the previous bar. We make the following observations:

- When using a bitmap size of 10^8 experiments failed.

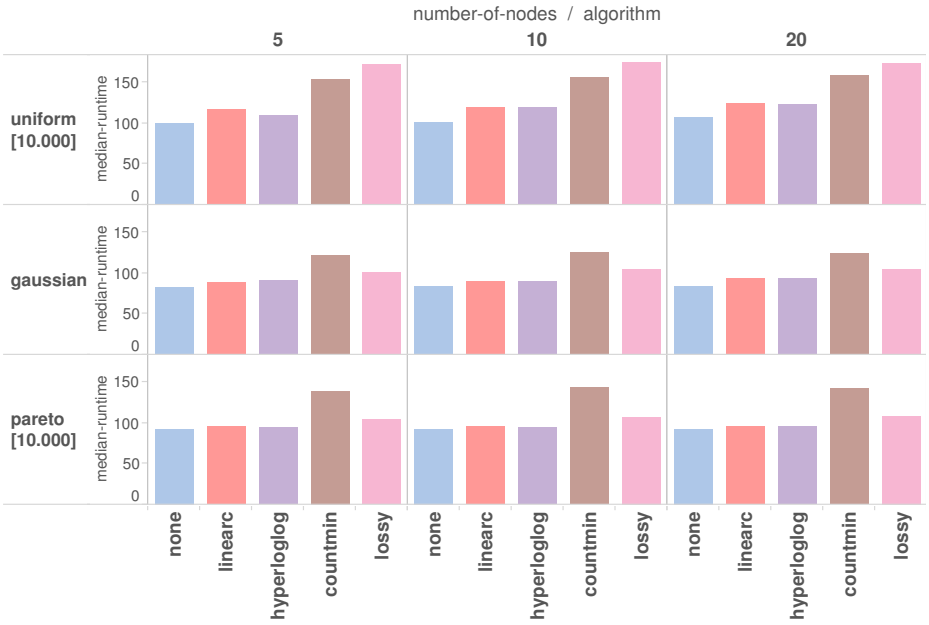

Fig. 4. Processing overhead data streaming statistics collection compared to no statistics collection

- Increasing the size of the bitmap increases overhead significantly (6-12%) for uniform data distributions. For pareto data distributions the behavior difference is not so clear.
- Increasing the $log_2 m$ parameter for hyperloglog has a small effect on the processing overhead, meaning that in terms of overhead, it is safe to use a large value for this parameter which will improve count distinct estimates for very large count distinct.

Heavy Hitters: Scale Parameters

Experiment Setup We analyze the effect that reducing the frequency threshold may have on processing overhead. We fix the fraction to error ratio to 10:1 (confidence for countmin is fixed at 0.99). We also analyze the effect that increasing confidence for count min sketch algorithm may have on processing overhead (fraction is fixed to 0.01, error is fixed to 0.001).

Results Discussion Figure 14 (appendix E plots the results obtained from this experiment. The bars represent median runtime and the labels represent the percentage difference with the previous column. We make the following observations:

- countmin failed when attempting to set the fraction to 10^{-4} or lower.
- lossy proved to be more robust for small fraction and error.
- Overhead differences when scaling the fraction are very small for pareto distribution (up to 6%).

- Overhead turned out to become very significant (121%) when scaling from fraction 10^{-4} to $10^{-}-5$ on uniform distribution.
- Increasing confidence by 0.5 causes an extra overhead of around (3-4%).

6.3 Accuracy Varying Algorithm Parameters

Linear Counting vs. HyperLogLog
We wish to quantify the order of magnitud of the error of the count distinct estimate provided by our algorithms.
Experiment Setup. We execute a simple FlatMap data job on 20 machines with a uniformly distributed set of **40 billion integers**. We scale the range of the data distribution from 10^5 to 10^9, increasing in powers of 10. We set the algorithm parameters to 10^6 **for the size of the bitmap** in linearc and $10 = \log_2 m$ for the hyperloglog bucket parameter.
Results Discussion The experiment results can be found in table 4 (appendix E). We observe that for smaller count distinct, linearc has a smaller error. However, as the actual count distinct grows orders of magnitude above the size of the bitmap, the error for linearc becomes very large (E+18). From the previous results we observed that the overhead of hyperLogLog and of linearc was similar, so given the increased accuracy of hyperLogLog for large cardinalities, we are inclined to pick hyperloglog for future statistics collection.
Lossy Counting vs. Count Min Sketch (Scale Out)
We analyze the effect that increasing the DoP may have on the accuracy of the heavy hitter algorithms.
Experiment Setup We fix the size of the data set to 10 Billion integers. We scale out in number of nodes (5, 10, 20). We use skewed data with pareto distribution.
Results Discussion Table 5 (appendix E) shows the results obtained from experiment 2. The following observations are of interest:

- The estimated frequencies for Count Min Sketch did not vary when varying the DoP. This is expected since the Count Min Sketch matrix is composable.
- The mean error of Lossy Counting is greater than that of Count Min Sketch, and it increases as the DoP does.
- While Count Min Sketch only detects true heavy hitters, Lossy Counting detects high frequency elements that are under the frequency threshold. For these elements the estimation error showed to be much larger than for true heavy hitters. If we observe the average of the error estimate for Lossy Counting when only considering the true heavy hitters (excluding non HH), it is very small.
- Since we wish to use the estimated frequencies to partition data, we calculate the sum of the errors, divided by the total set cardinality to get an idea of how these errors may impact balance. We see that for both algorithms this value is very small (E-006).

Count Min Sketch (Scale Confidence) Given the performance differences in overhead when reducing confidence for countmin, we decided to analyze how the frequency estimations would be affected by decreasing confidence.

Experiment Setup We ran a job on 40 Billion integers and 20 nodes and fixed fraction to 0.001 and error to 0.0001.
Results Discussion Table 6 (appendix E) shows the average of the frequency estimation errors when varying confidence. We observe that though the average error decreases by around 20 with every increase in confidence, if we compare the error to the total cardinality (40Billion) it is negligible. This means that we are willing to compromise accuracy by setting the confidence for countmin to 0.85, thus expecting countmin to cause around 10% less overhead.

7 Outlook: re-optimization opportunities

In this section we discuss re-optimization opportunities that arise from the knowledge that our statistics collection implementation provides us about the data distribution of operator keys. Previous works have focused on reducing query execution time by minimizing the cost of data shuffling (see section 2). We instead discuss opportunities which adapt the degree of parallelism and partitioning functions to achieve balanced partitions.

Adapting DoP If data is evenly distributed, the higher the DoP, the faster an operator will execute. However, if data is required to be key-partitioned and a small number of keys are very frequent, it might make sense to reduce the parallelism assigned to an operator since the execution of the operator will be bounded by the instance to which the very frequent key is assigned. Reducing the parallelism of an operator may free resources so that a data can be pipelined to a downstream operator. The following scenarios would benefit from change of DoP:

- If among the keys on an upcoming key-based operator there is a heavy hitter whose frequency represents a fraction of the total cardinality which is higher than the 1/DoP, the DoP should be reduced to the total cardinality divided by the heavy hitter frequency. This is because no matter what the distribution is of the remaining elements, the partition containing the heavy hitter will dominate the runtime (see diagram in appendix F.2).
- If the count distinct of the keys on an upcoming key-based operator is smaller than the DoP for that operator, the DoP should be reduced to the count distinct, since otherwise some instances will be idle (appendix F.2).

Custom partitioning based on frequencies If we have detected heavy hitters using a frequency threshold fraction s, we know that the number of heavy hitters is at most $1/s$. We propose the use of a custom partitioning function ($P : i \rightarrow \{0, DOP\}$) which attempts to balance heavy hitters across nodes by assigning elements to partitions in a greedy fashion. Since the number of heavy hitters is limited, this custom function can be stored in the memory of the Job Manager. By setting s to be a function of the degree of parallelism ($s = 1/c.DoP$ where c is some constant) we guarantee that we will detect those elements that

can have a strong impact on partition balance. Non heavy hitters can be distributed using hash partitioning or any other key based function.

We can assume a greedy strategy which builds the custom partition function by assigning heavy hitters to partitions in order of most frequent to least frequent. Alternatively, the partition function can be built on the fly by assigning every incoming heavy hitter to the partition that has least tuples (sum of heavy hitter frequencies) assigned to it.

Given certain conditions on the distribution of the data and the amount of memory available at the job manager, it is possible to build a custom partitioning function that maps all elements to a partition and not only heavy hitters (see appendix F.1).

8 Conclusions

We have successfully incorporated data streaming algorithms for statistics collection into the Apache Flink runtime. We have proposed and implemented distributed versions of Lossy Counting and Count Min Sketch, and proved the these distributed versions hold guarantees similar to the non distributed versions.

From our results we have shown that count distinct algorithms have an acceptably low impact on processing overhead. Among both algotihms testes we consider HyperLogLog to be preferable due to smaller overhead and more accurate results on high cardinality data. Heavy Hitter algorithms showed to have a heavy impact on runtime overhead, although this impact decreases as data skew increases. The behavior of Count Min Sketch proved to be less susceptible to changes in data distribution and DoP than Lossy Counting. However, Lossy Counting proved to be more robust to small fractions and errors.

8.1 Future Work

This project sets the ground for re-optimization of execution plans running in Flink by the use of statistics collected in runtime. We briefly mention study topics that may follow this contribution.

Leverage count distinct for re-optimization. Given that count distinct algorithms showed to have acceptably low overhead, we propose to begin studies of re-optimization using this statistic, and quantify how much processing times can be reduced.

Collect early statistics. Currently accumulators are sent to the job manager when a task closes. However, a new JIRA issue of the Apache Flink project [4] will develop periodic shipping of accumulators before during task execution. This will allow tracking early statistics.

Leverage early statistics. Study the possibility to leverage early statistics collected on heavy hitters to decide if data appears to be skewed. If not, discard tracking of this statistic to avoid overhead.

[4] https://issues.apache.org/jira/browse/FLINK-2292

References

1. Apache Flink. `https://flink.apache.org/`, accessed: 2015-05-30
2. Agarwal, S., Kandula, S., Bruno, N., Wu, M.C., Stoica, I., Zhou, J.: Re-optimizing data-parallel computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. NSDI'12, USENIX Association, Berkeley, CA, USA (2012)
3. Alexandrov, A., Bergmann, R., Ewen, S., Freytag, J.C., Hueske, F., Heise, A., Kao, O., Leich, M., Leser, U., Markl, V., Naumann, F., Peters, M., Rheinländer, A., Sax, M.J., Schelter, S., Höger, M., Tzoumas, K., Warneke, D.: The Stratosphere platform for big data analytics. The VLDB Journal (May 2014)
4. Armbrust, M., Ghodsi, A., Zaharia, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J.: Spark SQL: Relational Data Processing in Spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15. pp. 1383–1394. ACM Press, New York, New York, USA (May 2015)
5. Battré, D., Ewen, S., Hueske, F., Kao, O., Markl, V., Warneke, D.: Nephele/-PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In: Proceedings of the 1st ACM symposium on Cloud computing - SoCC '10. p. 119. ACM Press, New York, New York, USA (Jun 2010)
6. Bruno, N., Jain, S., Zhou, J.: Continuous cloud-scale query optimization and processing. Proceedings of the VLDB Endowment 6(11), 961–972 (Aug 2013)
7. Bruno, N., Kwon, Y., Wu, M.C.: Advanced join strategies for large-scale distributed computation. Proceedings of the VLDB Endowment 7(13), 1484–1495 (Aug 2014)
8. Chaudhuri, S.: An overview of query optimization in relational systems. In: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems - PODS '98. pp. 34–43. ACM Press, New York, New York, USA (May 1998)
9. Cormode, G., Hadjieleftheriou, M.: Methods for finding frequent items in data streams. The VLDB Journal 19(1), 3–20 (Feb 2010)
10. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. Journal of Algorithms 55(1), 58–75 (Apr 2005)
11. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. Communications of the ACM 51(1), 107 (Jan 2008)
12. Flajolet, P., ric Fusy, Gandouet, O., et al.: Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. AOFA 07: PROCEEDINGS OF THE 2007 INTERNATIONAL CONFERENCE ON ANALYSIS OF ALGORITHMS (2007)
13. Flajolet, P., Nigel Martin, G.: Probabilistic counting algorithms for data base applications. Journal of Computer and System Sciences 31(2), 182–209 (Oct 1985)
14. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edn. (2008)
15. Kabra, N., DeWitt, D.J.: Efficient mid-query re-optimization of sub-optimal query execution plans. ACM SIGMOD Record 27(2), 106–117 (Jun 1998)
16. Manku, G.S., Motwani, R.: Approximate frequency counts over data streams. In: Proceedings of the 28th International Conference on Very Large Data Bases. pp. 346–357. VLDB '02, VLDB Endowment (2002)
17. Metwally, A., Agrawal, D., Abbadi, A.E.: Why go logarithmic if we can go linear? In: Proceedings of the 11th international conference on Extending database technology Advances in database technology - EDBT '08. p. 618. ACM Press, New York, New York, USA (Mar 2008)

18. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. In: Proceedings of the 35th SIGMOD international conference on Management of data - SIGMOD '09. p. 165. ACM Press, New York, New York, USA (Jun 2009)
19. Sakr, S., Liu, A., Fayoumi, A.G.: The family of mapreduce and large-scale data processing systems. ACM Computing Surveys 46(1), 1–44 (Oct 2013)
20. Stillger, M., Lohman, G.M., Markl, V., Kandil, M.: Leo - db2's learning optimizer. In: Proceedings of the 27th International Conference on Very Large Data Bases. pp. 19–28. VLDB '01, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001)
21. Whang, K.Y., Vander-Zanden, B.T., Taylor, H.M.: A linear-time probabilistic counting algorithm for database applications. ACM Transactions on Database Systems 15(2), 208–229 (Jun 1990)
22. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing (Apr 2012)
23. Zhou, J., Bruno, N., Lin, W.: Advanced partitioning techniques for massively distributed computation. In: Proceedings of the 2012 international conference on Management of Data - SIGMOD '12. p. 13. ACM Press, New York, New York, USA (May 2012)
24. Zhou, J., Bruno, N., Wu, M.C., Larson, P.A., Chaiken, R., Shakib, D.: Scope: parallel databases meet mapreduce. The VLDB Journal 21(5), 611–636 (Jun 2012)
25. Zhou, J., Larson, P.A., Chaiken, R.: Incorporating partitioning and parallel plans into the SCOPE optimizer. In: 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010). pp. 1060–1071. IEEE (2010)

A Database technologies background

A.1 Cost Based Query Optimization in RDBMSs

In RDBMSs, incoming queries are parsed and optimized by a query compiler. The result is a physical query plan, or sequence of actions that the execution engine of the DBMS will perform to answer the query. Most query optimizers are cost based, i.e. they consider the best execution plan to be the one with the lowest cost. The cost of an execution plan is defined as the sum of the costs of the operators in the plan. Meanwhile, an operators' cost is determined mainly by the size of the data input it handles. Since input data of a query is transformed by every operator, the size of the data at an intermediate step of a query must be estimated at query compile time.

Query optimization is a difficult search problem, since it requires exploring a large search space of different physical query execution plans in order to choose the least costly plan (or at least a cost efficient one) [8]. In order to solve the problem, the optimizer requires: (1) a space of plans, (2) a cost estimation technique to assign a cost to each plan in the search space, and (3) an enumeration algorithm that can search through the execution space.

Pipelining The naive way to execute a query plan is to materialize the result of each operation on disk until it is needed for the next operation. However, this can be avoided by directly pipelining the results of one operation to the next operation, without storing the intermediate results on disk. The obvious advantage to pipelining is that it can reduce disk I/O's. Unary operations (e.g. selection and projection) are excellent candidates for pipelining since they are tuple-at-a-time operations. The results of binary operations can also be pipelined. An example illustrating this would be the following:

Example 1. Say we wish to perform $(R \bowtie S) \bowtie T$. If $(R \bowtie S)$ if is executed as a sort-merge join, the result will be a sorted group and can therefore be pipelined into a new sort-merge join with T. This would avoid the need to materialize the result of $R \bowtie S$.

A.2 MapReduce

MapReduce is a framework which allows developers to perform data processing jobs in parallel, without having to deal with the complexity of parallelization, fault-tolerance, data distribution and load balancing inherent to distributed execution environments. The idea behind MapReduce is very simple:

- Data is assumed to be stored in files.
- A master controller splits files into large blocks, and then distributes them among the nodes in the computer cluster.
- To process the data on a node, user defined code (in the form of *map* and *reduce* functions) is transferred to the node where it is executed.
- The *map* function is designed to take one key-value pair as input and to produce a list of key-value pairs as output.

- The output of the map processes is a collection of key-value pairs called the intermediate result.
- The input to *reduce* is a single key value from the intermediate result, together with the list of all values that appear with this key.

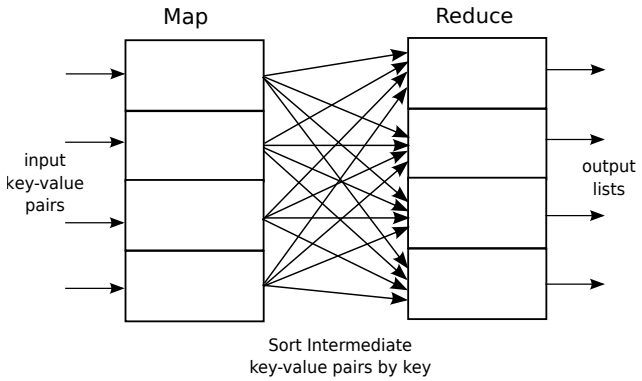

Fig. 5. Execution of *map* and *reduce* in MapReduce [14]

MapReduce was designed under the fundamental assumption that hardware failures (of individual machines, or racks) happen frequently in a large cluster of nodes, and thus should be automatically handled. To avoid data loss if a disk crashes, file chunks are replicated over different nodes, and in between map and reduce tasks data is materialized onto local files.

The strength of MapReduce lies in its flexibility and ability to scale horizontally almost effortlessly. However, there are limitations to this programming framework [18] which have lead to the implementation of hybrid systems between MapReduce and RDBMSs (such as Apache Flink [3] and Microsoft SCOPE [24]).

MapReduce Limitations

Intermediate results are always materialized Each map instance produces an output file for each reducer. These files are written to the local disk of the node where the map is executed. Then, each reduce instance needs to read its input files using a file-transfer protocol to pull each of them from the mappers. With many reduce instances running simultaneously, it is inevitable that two or more reduce instances will attempt to read their input files from the same map node simultaneously, inducing large numbers of disk seeks and slowing the effective disk transfer rate. This is why, in general, parallel database systems do not materialize their split files and instead use a "push" approach to transfer data. Though forced materialization of intermediate data sets can be seen as a limitation, it is due to this that tasks can be automatically restarted when the system detects that a node has failed.

Low level programming model MapReduce requires users to program data transformations in map and reduce function. This simple model is very flexible in terms of what can be done. However, it can be a barrier for applying traditional optimization techniques (such as operator re-ordering) that require knowing the data transformations that will be performed.

No indexes DBMSs generally provide indexes to accelerate access to data. MapReduce frameworks do not provide built-in indexes. The programmer must implement indexes inside of their application. This is not easily accomplished, as the frameworks data fetching mechanisms must also be instrumented to use these indexes when pushing data to running Map instances.

B Statistics Collection for RoPE

RoPE [2] (Reoptimizer for Parallel Executions) is a re-optimization component built on top of SCOPE [24]. RoPE collects statistics at worker nodes and piggybacks them to the job manager during query runtime, in order to re-optimize query plans that may have been chosen due to incorrect estimations. The statistics are sent once the worker has finished its current task. Table 3 summarizes the statistics that are collected in RoPE.

Type	**Description**	**Granularity**	**Algorithm**
Data Properties	Cardinality	Query subgraph	
	Avg. row length	Query subgraph	
	Count distint	Column@subgraph	Linear Counting
	Heavy hitters and their frequency	Column@subgraph	Lossy Counting
Operation costs	CPU and memory usage per data read and written	Task	Estimation
Leading statistics	Hash histogram	Column@subgraph	Hash based
	Exact sample	Column@subgraph	Reservoir sampling

Table 3. Statistics collected by RoPE

Data properties These are properties specific to the data. The number of rows and the average row length informs whether the data grows or shrinks as it flows through the query tree. Count distinct and heavy hitters are tracked on columns that are relevant for upcoming operators.

Operation costs Processing time and memory used by each task is tracked. A task is a multi-threaded process that reads one or more inputs from disk (locally or over the network) and writes its output to disk. Tasks can combine more than one operation running within the same thread.

Leading statistics RoPE collects simple leading statistics to help with typical pain points. Histograms and samples are collected on interesting columns. Hash-based histograms are used since equi-depth, equi-width and serial histograms proved not to satisfy the single pass, and bounded memory criterion. Reservoir sampling is used to pick constant sized random samples.

They authors propose a distributed form of Lossy Counting, in which heavy hitter structures collected from disjoint data sets, are merged by adding up the frequency estimates. The fraction is set to $s = 2\epsilon$ and the error to ϵ at local estimates. Global estimates are derived by summing the local estimated frequencies, and keeping only those elements with estimated frequency greater or equal to $\epsilon.n$.

Building on top of the research done by Agarwal et al. for RoPE, Bruno et al. [6] calculate *skew* over sets of columns C by collecting histograms over the sets of columns. The skew is then provided to the optimizer in runtime to improve the chosen execution plan. Skew is defined as the ratio between the size of the largest bucket and the average bucket sizes in the histogram:

$$skew(C) = \left(\frac{max\{f_i\}}{\sum f_i/N} - 1\right)\left(\frac{1}{N-1}\right) \quad (1)$$

where f_i is the size of bucket i in the histogram and N is the total number of buckets. The optimizer charges each operator with the cost of the largest partition that will be processed by that operator. When no information is known about data skew, the size of every partition es assumed to be the same, i.e. T/P where T is the total cardinality and P is the number of partitions. However, if skew is detected and data is partitioned on C, the size of the largest partition is estimated as:

$$\frac{T}{P}(1 + skew(C)(P-1)) \quad (2)$$

making cost estimations much more accurate.

C Data Streaming Algorithms for Statistics Collection

C.1 Linear Counting Algorithm (Count Distinct)

Whang et.al. [21] proposed this linear probabilistic counting algorithm in 1990 to count the number of unique values in columns in relational databases. Linear counting relies on a hash function to map incoming elements to a bitmap. The count distinct estimate is obtained by calculating how "full" the bitmap becomes.

At the beginning, a bit map of size b is allocated in main memory, with all entries initialized to "0's". A hash function is then applied to each incoming value, mapping it to a bit in the bitmap. When an element maps to a bit, it's value is turned to "1". Assuming uniform hashing, the expected probability that a bit is zero, after seeing n distinct elements is

$$V = (1 - 1/b)^n \approx e^{-n/b} \tag{3}$$

This probability can be estimated by the "empty" fraction of the bitmap, $\hat{V}$, i.e. the total number of "0's" left in the bitmap, divided by the size of the bitmap, b. Finally, the number of distinct elements n can be estimated by the maximum likelihood estimator:

$$\hat{n} = -b \ln \hat{V} \tag{4}$$

Since the size of the original bitmap can be much smaller than the expected cardinality, there will be collisions of different elements in the bitmap. These collisions will affect the accuracy of the estimate, and can be reduced by increasing the size of the bitmap. In particular, if we expect a maximum count distinct of n_{max}, an accurate estimate can be obtained by setting the size of the bitmap to n_{max}/ρ with $0 < \rho \leq 10$. Under these settings, the standard error is $O(1/\sqrt{n})$.

Linear counting is *composable*, since bitmaps can be easily merged. Provided multiple bitmaps have been generated using the same hash function, they can be merged by applying the OR operator. This will result in a bitmap in which every bit that is set to "1" in some of the merging bitmaps, will be set to "1" in the final merged one.

C.2 HyperLogLog Algorithm (Count Distinct)

HyperLogLog was proposed in 2007 by Flajolet et.al. [12]. The algorithm builds on the idea that the probability of observing k "0" bits at the beginning of a string of bits is 2^{-k}. In other words, in random data, a sequence of k "0" bits will occur once in every 2^k values. So, given a hash function that can map any incoming element to a sequence of bits, tracking the maximum number of leading "0's" that are observe can provide and estimate of the number of distinct elements that have been seen. Equal elements are not counted twice since they hash to the same value (hence to the same number of leading "0's").

To reduce variability in the estimate, a technique called stochastic averaging is used. The incoming elements are partitioned into m disjoint buckets, and for

each bucket the estimate is calculated using the maximum number of leading "0's" seen in that bucket. The final estimate is derived as the harmonic mean of the m bucket estimates. Partitioning incoming data is straightforward: once an incoming object is hashed, the first $log_2 m$ bits are used to select the bucket it corresponds to.

Let $h : D \rightarrow \{0,1\}^\infty$ hash data from domain D to binary domain
Let $\rho(i), \forall i \in \{0,1\}^\infty$ be the number of leading "0's" plus 1 ($\rho(0001) = 4$)
input : $\log_2 m > 0 \rightarrow m$ number of buckets
Multiset of items S from domain D
output: count distinct estimate E

initialize: collection of m registers, $M[1], ..., M[m]$, to $-\infty$

begin
  for $v \in S$ **do**
    set $x := h(v)$;
    `// binary address of first` b `bits of` x
    set $j := 1 + \langle x_1, x_2, ..., x_b \rangle_2$;
    set $w := x_{b+1} x_{b+2} ...$;
    set $M[j] := max(M[j], \rho(w))$;
  end
  compute: $Z := (\sum_{j=1}^{m} 2^{-M[j]})$;
  `//` α_m `is given by equation ( 5)`
  return: $E := \alpha_m . m^2 . Z$;
end

Algorithm 1: HyperLogLog Algorithm [12]

A very nice property of this algorithm, as its name implies, is the fact that it only requires memory that is $O(m \log_2 \log_2 n)$ where n is the cardinality of the data set, and m the number of buckets. In particular, it is only necessary to store the maximum number of leading "0's" seen in each m bucket, and the function used to hash incoming objects to a sequence of bits. The number of leading "0's" are encoded in bits, so that if we expect to estimate cardinalities up to 2^{32} we would need to hash the input to $m + 32$ bits, and then would only need $log_2 32$ bits to store the maximum of leading zero's for each bucket.

$$\alpha_m = (\int_0^\infty (\log_2(\frac{2+u}{1+u}))^m du)^{-1} \tag{5}$$

HyperLogLog is *composable*. Provided the same hash function is used to create two HyperLogLog estimator structures, they can be merged by iterating over the m buckets and selecting the maximum value between the two corresponding buckets.

Algorithm 1 shows the pseudo code for HyperLogLog. α_m is a bias corrector that is introduced to make estimations more accurate. The accuracy of HyperLogLog is $1.04/\sqrt{m}$.

C.3 Lossy Counting Algorithm (Heavy Hitters)

Lossy counting is a count-based algorithm to perform heavy hitter detection, proposed in 2002 by Manku et.al. [16]. Given a fraction s $(0 < s < 1)$, and a total item count n, all elements with a frequency higher than $s.n$ are considered heavy hitters. Given an error ϵ $(0 < \epsilon << s)$, Lossy Counting guarantees that:

- all values with frequency over $s.n$ will be output
- no item with a frequency lower than $(s - \epsilon)n$ will be output
- the estimated frequencies underestimate the true frequencies by less than $\epsilon.n$

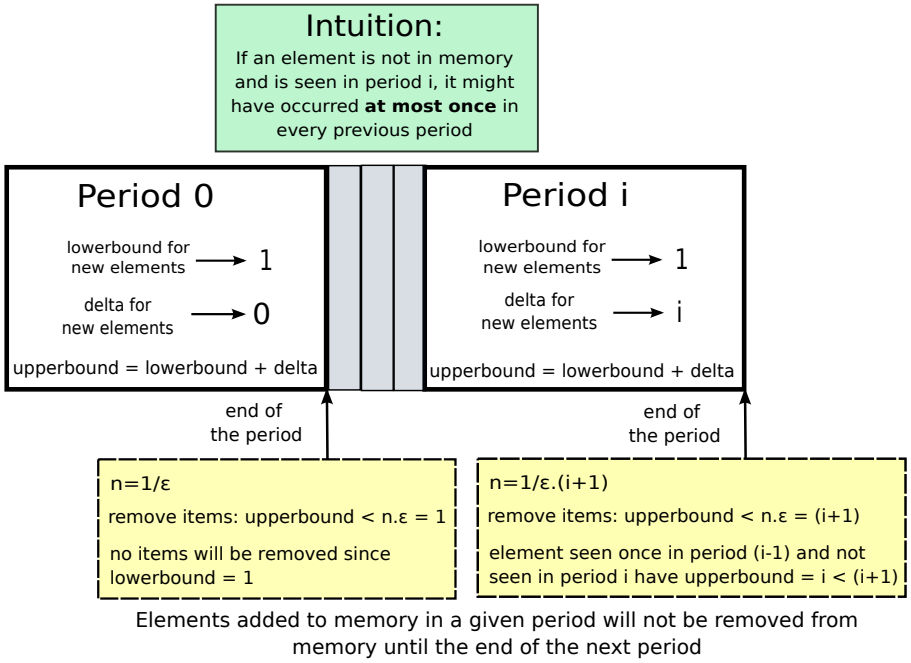

Fig. 6. Intuition behind Lossy Counting Algorithm

Lossy Counting stores heavy hitters in memory, along with a lower bound estimate for their frequency. In addition, the algorithm associates a delta value (Δ) for each item in memory indicating the difference between the lower bound and an upper bound for the frequency of the element. The algorithms works as follows, when processing a new item:

- if it is currently stored as a heavy hitter, its lower bound is increased by one.
- if not, a new tuple is created in the heavy hitter structure with the lower bound set to 1, and its Δ value set to $\lfloor \epsilon.n \rfloor$, where n is the total number of elements seen so far.

Periodically, all tuples whose upper bound is less than $\lfloor \epsilon.n \rfloor$ (n representing current cardinality) are removed from memory. The size of the period is determined by $\lceil 1/\epsilon \rceil$.

Figure 6 shows the intuition behind Lossy Counting. In every period all items are stored in memory. New tracked items in period i will have lower bound 1 and upper bound $i+1$ (lower bound + Δ). At the end of period i, elements with upper bound lower than $(i+1)$ are removed from memory. This means that all elements that were added in memory in period i will remain in memory at least until the end of period $i+1$. However, if by the end of period $(i+1)$ an element that was tracked for the first time in period i and has not been seen again, will be removed. Due to this periodical memory update procedure, if an element is not in memory and is seen in period i, it may have occurred **at most once** in every previous period (otherwise it would not have been removed from memory). This is the reason why its upper bound is set to $i = \rfloor \epsilon.n_i \lfloor$.

input : $s \rightarrow$ fraction that determines frequency for heavy hitters
$\epsilon \rightarrow$ error acceptable for frequency estimations
Multiset of items S in domain D
output : $T = \{(i, l, \Delta)\}; i \in D; l, \Delta \in \mathbb{N}$ set of tuples: (heavy hitter item, lower bound, diff. between lower and upper bound)
initialize: set of heavy hitters $H := \emptyset$
total cardinality $n := 0$

```
begin
    for i ∈ S do
        set n := n + 1;
        if i ∈ H then
            set l_i := l_i+ = 1;
        else
            set H := H ∪ (i, 1, ⌊ε.n⌋);
        end
        /* Every 1/ε elements, remove non frequent from H     */
        if n mod 1/ε == 0 then
            set H := H \ {h ∈ H : h_l + h_Δ < ε.n};
        end
    end
    /* Output elements with lower bound at least (s − ε).n    */
    return: T := {h ∈ H : h_l ≥ (s − ε).n};
end
```

Algorithm 2: Lossy Counting Algorithm [16]

When Lossy Counting ends, there can be elements in memory with a lower bound smaller than the minimum guaranteed frequency $(s-\epsilon)n$. The algorithm only returns elements with lower bound greater than $(s-\epsilon).n$ which are those considered heavy hitters.

Algorithm 2 shows the pseudo code for Lossy Counting. The worst case space required for the algorithm is $O(\frac{1}{\epsilon} \log \epsilon.n)$. Each period, $1/\epsilon$ elements are stored

in memory. By the end of each period, only those elements with upper bound greater than or equal to $\epsilon.n$ are kept. For distributions with few heavy hitters, less items are stored per period reducing memory requirements to be in the order $O(\frac{1}{\epsilon})$.

C.4 Distributed Lossy Counting

Algorithm 3 shows the pseudo code for the merge function for our distributed version of Lossy Counting.

input : $L_1 \rightarrow$ Lossy Counting structure to merge
$L_2 \rightarrow$ Lossy Counting structure to merge
output : $L' \rightarrow$ merged Lossy Counting structure

initialize: Lossy Counting structure L' with $s' = s_1$ and $\epsilon' = \epsilon_1$

begin
  if $s_2 \neq s' \vee \epsilon_2 \neq \epsilon'$ **then**
    throw new MergeException();
  end
  `/* Sum merged cardinality as sum of both cardinalities */`
  set $n' := n_1 + n_2$;
  let $h_1 = \{k : (k, l_k^1, \Delta_k^1) \in L_1\}$;
  let $h_2 = \{k : (k, l_k^2, \Delta_k^2) \in L_2\}$;
  for $i \in h_1 \cup h_2$ **do**
    if $i \in h_1 \wedge i \in h_2$ **then**
      set $L' := L' \cup \{(i, l_1^i + l_2^i, \Delta_1^i + \Delta_2^i)\}$;
    else if $i \in h_1 \wedge i \notin h_2$ **then**
      set $L' := L' \cup \{(i, l_1^i, \Delta_1^i + \epsilon.n_2)\}$;
    else
      set $L' := L' \cup \{(i, l_2^i, \Delta_2^i + \epsilon.n_1)\}$;
    end
  end
end

Algorithm 3: Lossy Counting Merge

C.5 Proof for Distributed Lossy Counting estimate bound

Proof. We use integers in the range $[1, p]$ to identify partitions. Suppose an element i is globally frequent ($f_i \geq s.n$) and is only tracked by one local partition j. This means the global estimate $\hat{f}_i$ will be equal to local estimate $\hat{f_i^j}$. We know that for any other partition k, the frequency of i is bounded by $s.n_k$ where n_k is the cardinality of the partition, otherwise $\hat{f_i^k}$ would have been derived.

$$\forall k \in [1, p], k \neq j : f_i^k < \epsilon.n_k$$

By summing over these errors we obtain:

$$\sum_{k\in[1,p],k\neq j} f_i^k < \sum_{k\in[1,p],k\neq j} \epsilon.n_k = \epsilon.(n - n_j) \tag{6}$$

Due to the Lossy Counting guarantees, we know that $f_i^j < \hat{f_i^j} + \epsilon.n_j$. We want to prove that the global frequency f_i is off from the local estimate $\hat{f_i^j}$ by at most $\epsilon.n$. We do this using the inequality from equation (6) and the one guaranteed by Lossy Counting.

$$f_i = \sum_{k\in[1,p]} f_i^j = (\sum_{k\in[1,p],k\neq j} f_i^k) + f_i^j < \epsilon.(n - n_j) + \hat{f_i^j} + \epsilon.n_j = \hat{f_i^j} + \epsilon.n$$

This shows that even if only one partition contributes to $\hat{f}_i$, the global error will remain under $\epsilon.n$.

C.6 Count Min Sketch Algorithm (Frequency Estimation)

Count Min Sketch is a sketch based frequency estimation algorithm that was proposed in 2005 by Cormode et.al. [10]. The idea behind Count Min Sketch is to map each incoming element to every row of a $d \times w$ matrix of counters, using d independent hash functions (one for each row). Each hash function h_i, $0 \leq i < d$ maps the input to an index j, $0 \leq j < w$ of the i-th row of the matrix. When a data item maps to a matrix position M_{ij}, the counter of that index is increased by one. This way, every incoming item maps to d counters.

Count Min Sketch estimates the frequency of an item by using the value of the minimum counter that the item maps to in the matrix. The intuition behind this is that the minimum counter is the matrix cell with the fewest collisions, and so the one closest to real frequency count. Said counter **overestimates** the real count by less than n/w. Algorithm 4 shows the pseudo code for Count Min Sketch.

The accuracy of Count Min Sketch is determined by the size of the matrix. For a given confidence δ, it is possible to guarantee that the probability that difference between the estimated frequency and the real frequency is higher than $\epsilon.n$, is lower than $(1 - \delta)$:

$$P(|\hat{f} - f| > \epsilon.n) < (1 - \delta) \tag{7}$$

In order to guarantee this, the size of the matrix should be determined by ϵ and δ according to the following equations:

$$w = \lceil \frac{e}{\epsilon} \rceil \tag{8}$$

$$d = \lceil \ln \frac{1}{\delta} \rceil \tag{9}$$

input : $\epsilon \rightarrow$ error acceptable for frequency estimations
$\delta \rightarrow$ confidence in the accuracy
Multiset of items S
output : Set of hash functions $D = \{h_1, ..., h_d\}$
Matrix $M = d \times w$ where the frequency of $v \in S$ can be estimated by $\hat{f}(v) = min(\{m_{ij} \in M : h_i(v) = j\})$
initialize: $d = \lceil \ln(1/\delta) \rceil$
$w = \lceil e/\epsilon \rceil$
matrix $M \rightarrow M[1][1]...M[d][w]$, to 0
Set $D = \{h_1, ..., h_d\}$ of independent hash functions. We use linear hash functions: $h(x) = (a * x) \bmod p$ where $a \in \mathbb{N}$ varies for every h_i and $p \in \mathbb{P}$ is fixed.

```
begin
    for v ∈ S do
        for h_i ∈ D do
            M[i][h_i(v)] += 1;
        end
    end
    return M;
end
```

Algorithm 4: CountMin Sketch Algorithm [10]

Count Min Sketch is *composable*. The Count Min Sketch matrices, generated by two separate data sets, can be merged into one, simply by summing them (provide both matrices have been generated using the same set of hash functions). The merged matrix will guarantee the same accuracy on the frequency estimations than the individual matrices.

C.7 Count Min Sketch Extension (Heavy Hitters)

Since Count Min Sketch tracks element frequencies, if we are interested in tracking heavy hitters, it is necessary to extend it. In the extended algorithm (algorithm 5) the same structures from Count Min Sketch are used to track frequencies of elements but in addition we keep a list of heavy hitters. This list stores the elements whose frequency exceeds the minimum expected frequency for a heavy hitter. Periodically, outdated heavy hitters are removed from the list. Since Count Min Sketch overestimates frequencies, we can assume that any element whose estimated frequency is not larger than the desired frequency, will certainly not be a heavy hitter, and hence should not be tracked. We periodically clean the heavy hitter list from outdated heavy hitters, to reduce memory requirements.

input : $\epsilon \rightarrow$ error acceptable for frequency estimations
$\delta \rightarrow$ confidence in the accuracy
$s \rightarrow$ fraction that determines frequency for heavy hitters
Multiset of items S in domain D
output : $H = \{(i, \hat{f}_i) : i \in D, \hat{f}_i \in \mathbb{N})\}$, $\hat{f}_i$ is estimated frequency
initialize: count min sketch $C :=$ CountMinSketch(ϵ, δ)
total cardinality $n := 0$
heavy hitter Map $H := \emptyset$
begin
  for $i \in S$ **do**
    n+=1;
    $C.add(i)$;
    if $C.frequency(i) \geq s.n$ **then**
      set $H := H \cup \{(i, C.frequency(i))\}$
    end
    `/* Every` $1/\epsilon$ `elements, remove non frequent from` H `*/`
    if $n \bmod 1/\epsilon == 0$ **then**
      set $H := H \setminus \{(h, \hat{f}_h) \in H : \hat{f}_h < s.n\}$;
    end
  end
  `/* Remove non frequent from` H `before returning */`
  return $H \setminus \{(h, \hat{f}_h) \in H : \hat{f}_h < s.n\}$;
end

Algorithm 5: Count Min Sketch for Heavy Hitters

C.8 Distributed Count Min Sketch for Heavy Hitters

Algorithm 6 shows our merge function for combining two Count Min Sketch Extension structures for tracking heavy hitters. First the Count Min Sketch matrices for estimating frequencies are summed in order to estimate the joint frequencies. Then, for each element in either of the heavy hitters maps, its joint frequency is looked up in the summed matrix, and if it larger than the threshold, it it inserted into the merged heavy hitter map.

input : $L_1 \rightarrow$ Count Min Sketch Extension structure to merge
$L_2 \rightarrow$ Count Min Sketch Extension structure to merge
output : $L' \rightarrow$ merged Count Min Sketch Extension structure
initialize: $s' = s_1$
$\epsilon' = \epsilon_1$
$\delta' = \delta_1$
merged Count Min Sketch $C' :=$ CountMinSketch(ϵ', δ')
total cardinality $n' := 0$
merged heavy hitters $H' := \emptyset$
begin
`/* Parameters must be the same for both sketches */`
if $s_2 \neq s' \vee \epsilon_2 \neq \epsilon' \vee \delta_2 \neq \delta'$ **then**
throw new MergeException();
end
`/* Sum cardinalities */`
set $n' := n_1 + n_2$;
`/* Sum Count Min Sketch matrices */`
set $C' := C_1 + C_2$;
let $h_1 = \{k : (k, \hat{f}_k^1) \in H_1\}$;
let $h_2 = \{k : (k, \hat{f}_k^2) \in H_2\}$;
for $i \in h_1 \cup h_2$ **do**
`/* Estimate frequency with merged CMS matrix */`
set $\hat{f}_i' := C'.frequency(i)$;
if $\hat{f}_i' \geq s' * n'$ **then**
set $H' := H' \cup \{(i, \hat{f}_i')\}$
end
end
return $L' =$ CountMinSketchExtension(C', H');
end

Algorithm 6: Merge of Count Min Sketch Extension for Heavy Hitters

D Implementation Details

D.1 Flow of Accumulators from Task to Job Manager

The following figure shows how accumulators flow from the tasks where they are created and updated to the Job Manager.

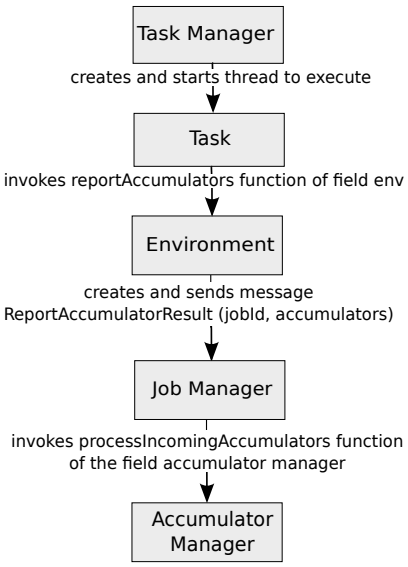

Fig. 7. Accumulator flow from Task to Job Manager

D.2 Example of the use of accumulators in a Rich Map Function

The following code snippet is an example of a UDF written in the Flink JAVA API which takes a String as input and produces exactly one Tuple of size two composed of two Integers. Given that the UDF extends a RichFunction, it can access its runtime context, as well as define open and close functions to be executed before and after the actual UDF. It is in the open function that accumulators are usually initialized. The accumulators are then updated inside of the actual UDF, where the data that is processed by the operator can be accessed.

```
public static class ParseInt extends
 RichMapFunction<String, Tuple2<Integer, Integer>> {

  Accumulator accumulator;

  @Override
  public void open(Configuration parameters) {
        accumulator = getRuntimeContext()
                      .getAccumulator("acc-id");
```

```
  }

  @Override
  public Tuple2<Integer,Integer> map(String value) {
        int intValue = Integer.parseInt(value);
        accumulator.add(intValue);
        return new Tuple2(intValue, 1);
  }
}
```

D.3 Rich Functions in Flink

The following figure shows the class hierarchy for Flink rich functions. The bottom of the hierarchy is the RichFlatMapFunction which corresponds to the function used in the example in appendix D.2. The abstract class AbstractRichFunction must be Serializable since these functions must be communicated through the network from the Job Manager to the Task Managers who will execute them.

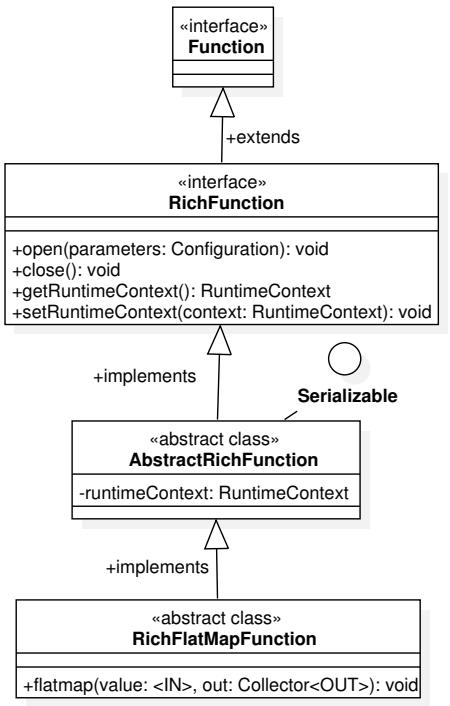

Fig. 8. The function class hierarchy in Flink

D.4 Accumulator Classes

The following figures show the UML diagrams of the Flink Accumulator class and the OperatorStatisticsAccumulator class.

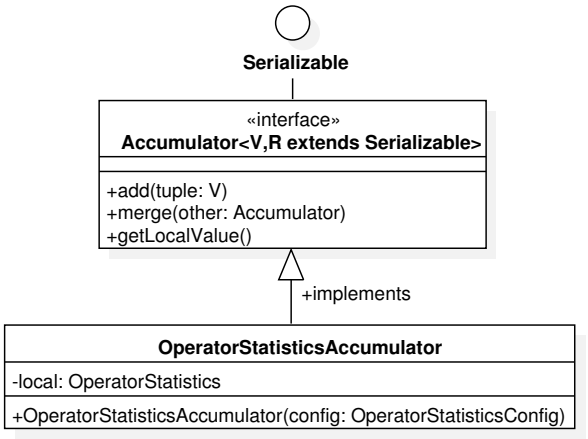

Fig. 9. The class Accumulator in Flink

To make statistics tracking more flexible, we implement a class called OperatorStatisticsConfig, which dictates which statistics will be tracked, as well as the parameters that our count distinct and heavy hitters algorithms require. The following figures show the class OperatorStatistics and OperatorStatisticsConfig.

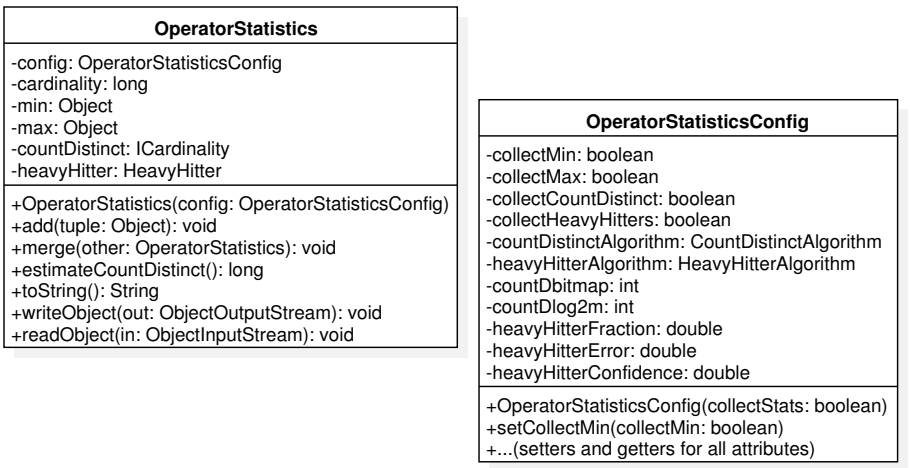

Fig. 10. The classes OperatorStatistics and OperatorStatisticsConfig implemented in Flink in the scope of our project

D.5 Classes in Flink Runtime

The following figure shows the UML diagram of the Flink Task class, i.e. the class which executes UDF's in Flink. This class contains a field Runtime Context and implements a series of functions to support the execution of the UDFs. Of particular interest to us is the function reportAndClearAccumulators which is invoked when tasks finish executing the user defined code and close.

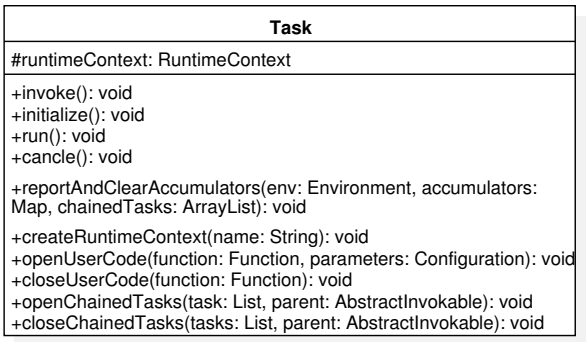

Fig. 11. The Task class in Flink

The upcoming figure shows the UML for the class Runtime Context which contains information about the context in which a UDF is executed. This class allows access to the accumulators associated to a task, among other contextual properties.

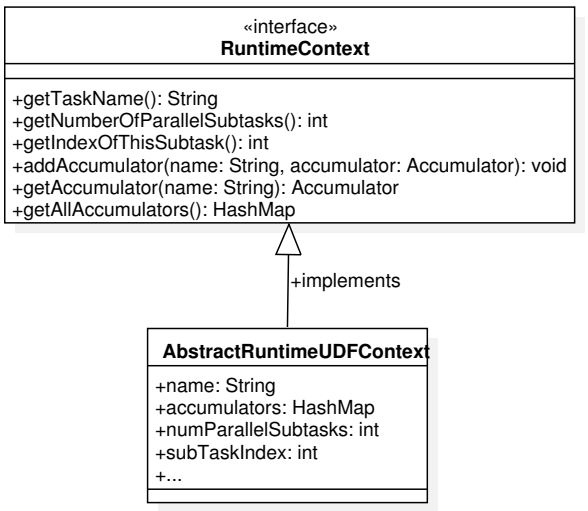

Fig. 12. The Runtime Context in Flink

E Experimental Results

Fig. 13. Processing overhead of count distinct when varying algorithm parameters

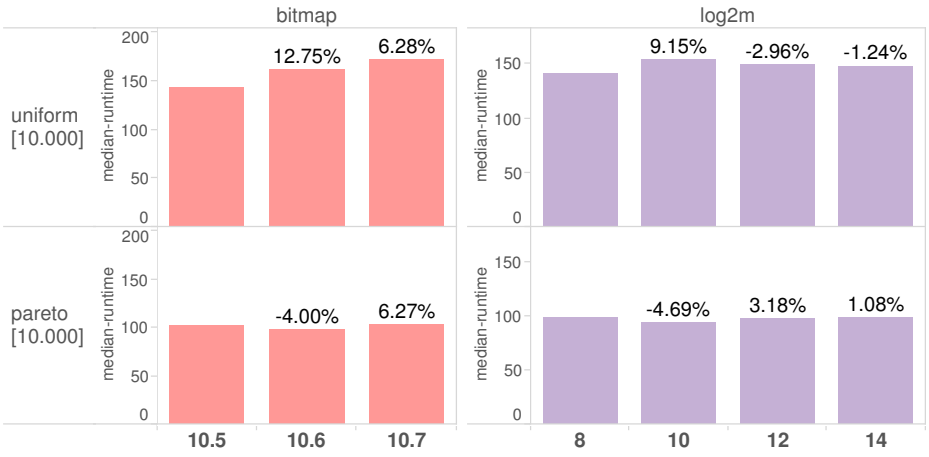

Fig. 14. Processing overhead of heavy hitter when varying fraction

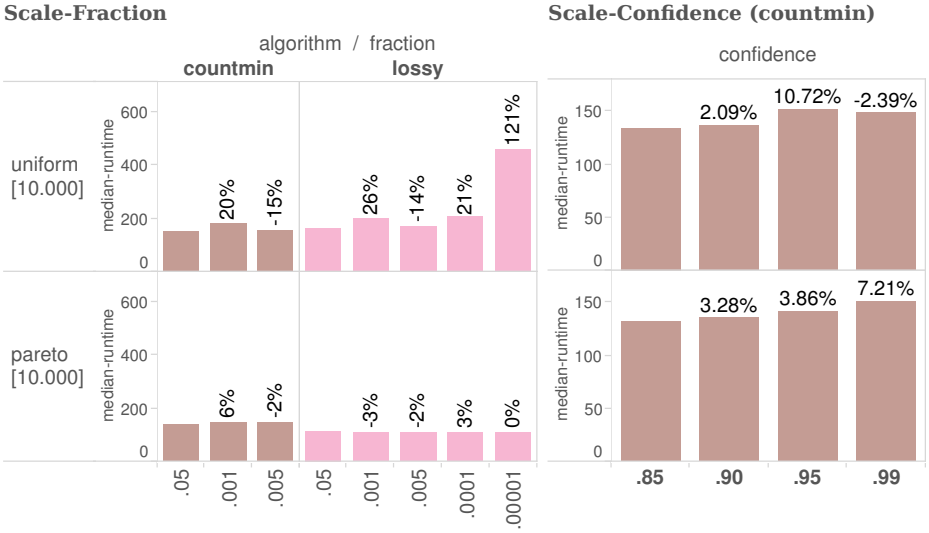

Table 4. Experiment Results for Count Distinct Accuracy

Data Distribution	Algorithm	Error
Uniform(10^5)	linearc	0,23E+2
	hyperloglog	5,19E+3
Uniform(10^6)	linearc	2,27E+2
	hyperloglog	1,75E+4
Uniform(10^7)	linearc	1,56E+4
	hyperloglog	2,96E+5
Uniform(10^8)	linearc	2,80E+6
	hyperloglog	2,69E+5
Uniform(10^9)	linearc	**9,22E+18**
	hyperloglog	1,49E+8

Table 5. Experiment results for effect of scaling out on the accuracy of heavy hitter algorithms

algorithm	**lossy**			**countmin**		
number of workers	**5N**	**10N**	**20N**	**5N**	**10N**	**20N**
mean error	594	1,385	2,154	348	348	348
(excluding non HH)	0.4	0.7	1.6	-	-	-
sum errors/cardinality	2.61E-006	6.10E-006	9.48E-006	1.08E-006	1.08E-006	1.08E-006
(excluding non HH)	1.50E-009	2.30E-009	4.90E-009	-	-	-

Table 6. Experiment results for accuracy when scaling confidence for Count Min Sketch algorithm

confidence	**0.85**	**0.90**	**0.95**	**0.99**
avg. error	96.45	78.97	73.77	54.23
sum error/cardinality	7.48E-008	6.12E-008	5.72E-008	4.20E-008

F Optimization Opportunities using collected statistics

F.1 Custom partitioning based on frequencies

In section 7 we discuss how to build a custom partitioning function which maps heavy hitters to partitions. We discuss that since memory is limited, the mapping function we describe is only defined on heavy hitters. Non heavy hitters need to be partitioned using some other key based function (e.g. hash). However, if in addition to collecting heavy hitters, we collect count distinct and notice that the number of distinct elements is small enough so that it is possible to keep a map of all elements in memory, we can extend our custom partition function to map all elements to a partition.

If heavy hitters are collected using Count Min Sketch, estimated frequencies for non heavy hitters can be found in the Count Min Sketch frequency matrix. Otherwise, if we only have frequency estimations of heavy hitters (like for Lossy Counting), we might assign the same frequency estimate for all non heavy hitters, making the simplifying assumption that they are all equally frequent. We can write this frequency estimation function as follows:

$$\widetilde{f}_i = \begin{cases} \hat{f}_i & \text{if } i \text{ is heavy hitter} \\ \frac{(n - \sum_{j \in H} \hat{f}_j)}{C - |H|} & \text{otherwise.} \end{cases} \tag{10}$$

We use f_i and $\hat{f}_i$ to represent the real and estimated frequency of element i. H represents the set of heavy hitters and C is the count distinct estimate. We use n to represent the total number of elements (cardinality).

F.2 Adapting DoP

Fig. 15. Reducing DoP when detecting a very small count distinct

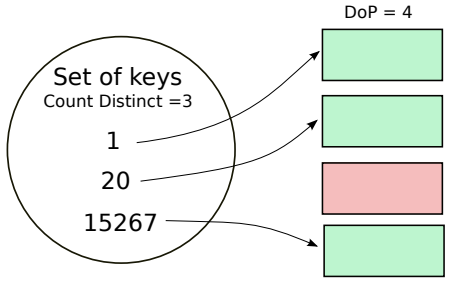

Fig. 16. Reducing DoP when detecting a very frequent heavy hitters