The VLDB Journal (2014) 23:517–539
DOI 10.1007/s00778-013-0335-9

REGULAR PAPER

Partitioning functions for stateful data parallelism in stream processing

Buğra Gedik

Received: 1 January 2013 / Revised: 1 June 2013 / Accepted: 13 August 2013 / Published online: 31 August 2013

Abstract In this paper, we study partitioning functions for stream processing systems that employ stateful data parallelism to improve application throughput. In particular, we develop partitioning functions that are effective under workloads where the domain of the partitioning key is large and its value distribution is skewed. We define various desirable properties for partitioning functions, ranging from *balance properties* such as memory, processing, and communication balance, *structural properties* such as compactness and fast lookup, and *adaptation properties* such as fast computation and minimal migration. We introduce a partitioning function structure that is compact and develop several associated heuristic construction techniques that exhibit good balance and low migration cost under skewed workloads. We provide experimental results that compare our partitioning functions to more traditional approaches such as uniform and consistent hashing, under different workload and application characteristics, and show superior performance.

Keywords Stream processing · Load balance · Partitioning functions

1 Introduction

In today's highly instrumented and interconnected world, there is a deluge of data coming from various software and hardware sensors. This data are often in the form of continuous streams. Examples can be found in several domains, such as financial markets, telecommunications, surveillance, manufacturing, and healthcare. Accordingly, there is an increasing need to gather and analyze data streams in near realtime to extract insights and detect emerging patterns and outliers. Stream processing systems [1,3,10,22,27,28] enable carrying out these tasks in an efficient and scalable manner, by taking data streams through a network of operators placed on a set of distributed hosts.

Handling large volumes of live data in short periods of time is a major characteristic of stream processing applications. Thus, supporting high throughput processing is a critical requirement for streaming systems. It necessitates taking advantage of multiple cores and/or host machines to achieve scale. This requirement becomes even more prominent with the ever increasing amount of live data available for processing. The increased affordability of distributed and parallel computing, thanks to advances in cloud computing and multi-core chip design, has made this problem tractable. This requires language and system level techniques that can effectively locate and efficiently exploit data parallelism opportunities in stream processing applications. This latter aspect, which we call *auto-fission*, has been studied recently [9,23,24].

Auto-fission is an operator graph transformation technique that creates replicas, called *parallel channels*, from a sub-topology, called the *parallel region*. It then distributes the incoming tuples over the parallel channels so that the logic encapsulated by the parallel region can be executed by more than one core or host, over different data. The results are then usually merged back into a single stream to reestablish the original order. More advanced transformations, such as shuffles, are also possible. The *automatic* aspect of the fission optimization deals with making this transformation transparent as well as making it safe (at compile-time [24]) and adaptive (at runtime [9]). For instance, the number of parallel channels can be elastically set based on the workload and resource availability at runtime.

B. Gedik (✉)
Computer Science Department, Bilkent University,
Bilkent, 06800 Ankara, Turkey
e-mail: bgedik@cs.bilkent.edu.tr

In this paper, we are interested in the work distribution across the parallel channels, especially when the system has adaptation properties, such as changing the number of parallel channels used at runtime. This adaptation is an important capability, since it is needed both when the workload and resource availability show variability, as well as when it does not. As an example for the former, vehicle traffic and phone call data typically have peak times during the day. Furthermore, various online services need scalability as they become successful, due to increasing user base and usage amount. It is often helpful to scale stream processing applications by adapting the number of channels without downtime. In the latter case (no workload or resource variability), the adaptation is needed to provide transparent fission, as the system needs to find an effective operating point before it settles down on the number of parallel channels to use. This relieves the developer from specifying the number of parallel channels explicitly (typically done through hints [27]).

The work distribution often takes place inside a *split* operator, which determines the parallel channel a tuple is to be routed for processing. For parallel regions that are stateless, this routing can be accomplished in a round-robin fashion. In this paper, we are interested in stateful operators, in particular, the *partitioned stateful* operators that are amenable to data parallelism. Such operators keep state on a sub-stream basis, where each sub-stream is identified by a *partitioning key*. Examples of such operators include streaming aggregation, progressive sort, one-way join, as well as user-defined operators [13]. Note that stateless operators can be combined with the partitioned stateful ones to create larger parallel regions, which behave similar to partitioned stateful operators. Even multiple partitioned stateful operators can be combined if their partitioning keys are compatible (there is a common subset).

Importantly, for partitioned stateful operators, the partitioning cannot be performed by simply routing tuples using a round-robin policy. Instead, a hash function is used, which always routes the tuples with the same partitioning key value to the same parallel channel. This way state can be maintained on a sub-stream basis, thus preserving the application semantics. Typically, a uniform hash function is used for this purpose. This works well unless the system supports adjusting the number of parallel channels at runtime. Uniform hash functions are not suitable for adaptation, because the number-of-channel adaptation in the presence of stateful operators requires *state migration* and uniform hash functions perform poorly under this requirement. For instance, when a new channel is added, the state associated with the sub-streams that will execute on that channel should be moved over from their current channels (possibly on a different host).

With uniform hash functions, the number of items that migrate when a new channel is added/removed is far from the ideal that can be achieved. A common solution to this problem is to use a *consistent hash* [15] in place of a uniform hash. Consistent hashing is a technique that can both balance the load and minimize the migration. In particular, when a new channel is added, the amount of migration that is introduced by consistent hashing is equal to the size of the new channel's fair share of state, and this migration only happens between the new channel and the existing ones, never between the existing channels.

However, in the presence of skew in the distribution of the partitioning key, the balance properties cannot be maintained by the consistent hash. As an example, consider a financial stream that contains trade and quote information. There are many financial computations that can be performed on this stream, including those that require computation of certain metrics such as VWAP (volume-weighted average price) on a per sub-stream basis. In this case, each sub-stream is identified by a stock ticker. However, the distribution of stock tickers is highly skewed—a few high volume tickets constitute a large portion of the total volume. Such skew in the workload creates several problems:

- The memory usage across parallel channels may become imbalanced.
- The computation cost across parallel channels may become imbalanced.
- The communication cost across parallel channels may become imbalanced.

Any one of these can result in a bottleneck, limiting application scalability in terms of throughput. Furthermore, several of these metrics are dependent on the application characteristics. For instance, if the computation cost for a tuple from a given sub-stream is dependent on that sub-stream's volume (i.e., the frequency of the partitioning key value), then the computation balance will be more difficult to accomplish in the presence of skew. This is because, not all sub-streams will be equal in terms of their computation cost.

We assume a general purpose stream processing system, in which a parallel channel can be arbitrarily costly in terms of time and/or space. This is because in such systems there is no limit to the number of streaming operators that can appear in a parallel region, as well as no limit on the complexity of these operators. If a partitioning function associated with a parallel region does not do a good job in balancing the load, the channel that becomes overloaded will slow down the entire flow, limiting the scalability of fission.

Coming up with a partitioning function that preserves balance in the presence of workload skew brings several challenges. First, the system needs to track the frequencies of the partitioning key values. When the partitioning key comes from a large domain (e.g., the domain of IP addresses), this has to be performed without keeping a frequency for each possible partitioning key value. Second, while achieving balance, the system should also maintain low migration cost. Often these two metrics are conflicting, as migrating items provides additional flexibility in terms of achieving good balance, at the cost of a higher migration cost. Third, the partitioning function should be computable in short time, so as not to disturb the adaptation process. The number-of-channel adaptation often requires suspending the stream briefly to perform the migrations, introducing a migration delay. The creation of the partitioning function should not become the bottleneck for the migration delay.

In this paper, we propose a partitioning function and associated construction algorithms that address these challenges. Concretely, we introduce a partitioning function structure that is a hybrid between a consistent hash and an explicit mapping. This results in a compact hash function that is flexible enough to provide good balance in the presence of high skew. We use the lossy counting algorithm [18] in a sliding window setting to keep track of the high frequency items. We determine the frequency threshold automatically. We develop heuristic algorithms that use the last partitioning function and the current frequencies to construct a new partitioning function, with the aim of keeping the migration cost low and the various forms of balance high. The heuristic nature of the algorithms ensures fast computation time. We propose and evaluate alternative metrics that drive the partition function construction algorithms. These metrics help us improve the balance and migration characteristics of the algorithms. Our results show that the new partitioning functions exhibit desirable properties across a wide range of workload and application characteristics and outperform alternatives such as uniform and consistent hashing.

In summary, this paper makes the following contributions:

- Formalizes the characteristics expected from partitioning functions to be used for auto-fission in stream processing systems.
- Introduces a partitioning function structure that is amenable to time and memory efficient mapping of tuples to parallel channels.
- Develops construction algorithms and associated metrics that can be used to build partitioning functions with good balance and cheap migration.
- Presents an evaluation of the proposed techniques, showcasing the superior behavior of the partitioning functions under different workload and application characteristics.

The rest of this paper is organized as follows. Section 2 provides an overview of the problem, followed by a detailed formalization in Sect. 3. The solution approach, which includes the partitioning function and associated construction algorithms with their heuristic metrics, is given in Sect. 4. Experimental results are presented in Sect. 5. The related work is discussed in Sect. 6, and the conclusions are given in Sect. 7.

2 Overview

In this section, we overview the partitioning problem and exemplify it with a toy scenario.

Let S be a stream of tuples and $\tau \in S$ a tuple. For each tuple τ, let $\iota(\tau)$ denote the value of the partitioning key. We represent the domain of the partitioning key by $\mathcal{D}$. Thus, we have $\iota(\tau) \in \mathcal{D}$. For each value of the partitioning key $d \in \mathcal{D}$, we denote its relative frequency as $f(d) \in [0, 1]$. We assume that the frequencies of items can change in the long term.

We define a partitioning function $p : \mathcal{D} \rightarrow [1..N]$, where this function maps the partitioning key value $\iota(\tau)$ of a tuple τ to an index in the range $[1..N]$. The index represents the parallel channel the tuple is assigned to. The number of channels, that is N, can change as well (for instance, as a result of changes in the workload or resource availability).

Let $p^{(t)}$ be the partitioning function used during time period t. Our goal is to update this function for use during time period $t+1$ as $p^{(t+1)}$, such that load balance properties, structural properties, and adaptation properties are satisfied. As the time progresses, two kinds of changes could happen. The number of channels can change from $N^{(t)}$ to $N^{(t+1)}$. This could be an increase in the number of channels or a decrease. Similarly, the frequencies of items, that is the function f, can change.

We summarize the desired properties of the partitioning function as follows:

1. **Load balance properties**: These properties deal with the ability of the partitioning function to balance memory, processing, and bandwidth consumptions of different parallel channels.
2. **Structural properties**: These properties deal with the computational and size complexity of performing lookups on the partitioning function.
3. **Adaptation properties**: These properties deal with the computational complexity and the migration cost associated with updating the partitioning function in the presence of changes in the number of channels or in the frequencies of the data items.

We look at these properties more formally in the next section. For now, consider the toy scenario depicted in Fig. 1. In this scenario, we have at time t, $N^{(t)} = 2$ and at time $t+1$, $N^{(t+1)} = 3$. There are 8 unique partitioning key values in this example, thus $\mathcal{D} = \{\text{X, Z, V, R, U, Y, W, L}\}$, with frequencies $\{5, 3, 2, 1, 4, 3, 3, 1\}$, respectively.

Assume that both the communication and the computation across the channels needs to be balanced (i.e., both are

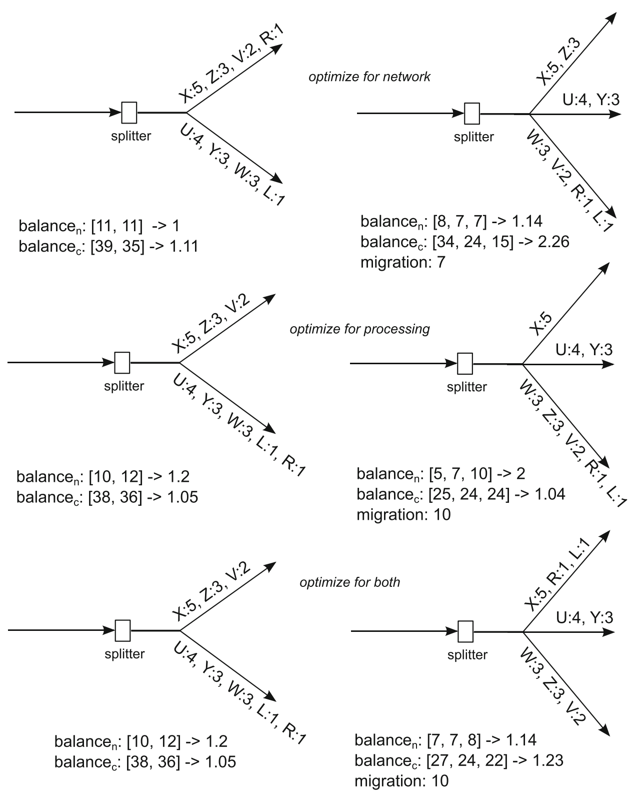


Fig. 1 A toy example showcasing different trade-offs in construction of the partitioning function

bottlenecks). Further assume that the processing cost for an item is quadratic in its frequency. We will look at examples of such applications in the next section. In the figure, we see 3 alternative ways of constructing the partitioning function.

In the first setup, shown at the top of the figure, we see that the initial partitioning is optimized for communication, where for $N = 2$ we have a perfect communication balance (balance$_\mathrm{n}$ in the figure): The ratio of the maximum communication cost for a channel divided by the minimum is simply 1 (the communication costs are given by [11, 11]). Incidentally, the balance of the computation cost (balance$_\mathrm{c}$ in the figure) is also good, since the max-to-min ratio is 1.1 (the communication costs are given by [39, 35]). As we move to $N = 3$, the communication load is kept balanced (1.14), but since we are not optimizing for processing, the computation balance suffers (2.26). Also note that, the move from $N = 2$ to $N = 3$ results in a migration cost of 7 (items U and Y with costs 4 and 3 has moved).

In the middle of the figure, we see a different setup where the partitioning is optimized for computation. We can see that the initial setup with $N = 2$ has great computation balance (1.05) and good communication balance (1.2). But as we move to $N = 3$, the computation is kept balanced (1.04), but the communication suffers (2). Also note that, keeping the computation balanced resulted in a higher migration cost of 10, compared to keeping the communication balanced (a quadratic function versus a linear function).

Finally, at the bottom of the figure, we see a setup where the partitioning is optimized for both communication and computation. We see that for both $N = 2$ and $N = 3$, we have good communication (1.2 and 1.14, respectively) and computation (1.05 and 1.23, respectively) balance. It is interesting to note that this requires migrations between the existing channels, as well as from existing channels to the new channel.

3 Problem definition

In this section, we formalize the desired properties of the partitioning function.

3.1 Load balance properties

Load balance becomes a problem when there is skew in the distribution of the partitioning key. Skew can result in sub-optimal performance, as a data parallel stream processing flow is limited by its slowest parallel channel. The bottleneck could be due to memory imbalance (resulting in thrashing), processing imbalance (resulting in overload), and bandwidth imbalance (resulting in backpressure [10]).

The load balance problem is non-trivial in the presence of partitioned stateful parallelism, since a round-robin distribution of tuples is not possible under this model. A uniform or consistent hash-based distribution of the partitioning keys, while maintaining semantic correctness, can result in imbalance when the value frequencies follow a skewed distribution.

Memory load balance. The partitioning should ensure that the load imposed on each channel in terms of the state they need to maintain is close to each other. For this purpose, we define a *resource function* $\beta_s : [0, 1] \rightarrow \mathbb{R}$ that maps a given frequency to a value proportional to the amount of state that has to be kept on a channel for tuples having a partitioning key value with that frequency. Let us denote the state that needs to be maintained for $d \in \mathcal{D}$ as $\mathcal{S}(d)$, then we have $|\mathcal{S}(d)| \propto \beta_s(f(d))$.

As an example, consider a channel that contains an operator keeping a time-based window of size T. We have $|\mathcal{S}(d)| \propto T \cdot f(d)$ and since T is constant, $\beta_s(x) = x$. If the operator is keeping a count-based window of size C, then we have $|\mathcal{S}(d)| \propto C$ and thus $\beta_s(x) = 1$.

Let $L_s(i)$ denote the memory load of a host $i \in [1..N]$. We have:

$$L_s(i) = \sum_{d \in \mathcal{D} \text{ s.t. } p(d)=i} \beta_s(f(d)) \tag{1}$$

We express the memory load balance requirement as:

$$r_s = \frac{\max_{i \in [1..N]} L_s(i)}{\min_{i \in [1..N]} L_s(i)} \leq \alpha_s \tag{2}$$

Here, $\alpha_s \geq 1$ represents the level of memory imbalance (r_s) tolerated.

Computation load balance. The partitioning should ensure that the load imposed on each channel in terms of the computation they handle is close to each other.

For this purpose, we define a resource function $\beta_c : [0, 1] \rightarrow \mathbb{R}$ that maps a given frequency to a value proportional to the amount of computation that has to be performed on a channel to process tuples having a partitioning key value with that frequency. Let us denote the cost of computation that needs to be performed for $d \in \mathcal{D}$ as $\mathcal{C}(d)$, then we have $\mathcal{C}(d) \propto \beta_c(f(d))$.

As an example, again consider a channel that contains an operator keeping a time-based window of size T. Further assume that each new tuple needs to be compared against all existing tuples in the window (a join-like operator). This means that we have $\mathcal{C}(d) \propto f(d) \cdot \beta_s(f(d)) \propto (f(d))^2$, and thus $\beta_c(x) = x^2$. Various different β_c functions are possible based on the nature of the processing, especially the size of the portion of the kept state that needs to be involved in the computation.

Let $L_c(i)$ denote the computation load of a channel $i \in [1..N]$. We have

$$L_c(i) = \sum_{d \in \mathcal{D} \text{ s.t. } p(d)=i} f(d) \cdot \beta_c(f(d)) \tag{3}$$

We express the computation load balance requirement as follows:

$$r_c = \frac{\max_{i \in [1..N]} L_c(i)}{\min_{i \in [1..N]} L_c(i)} \leq \alpha_c \tag{4}$$

Here, $\alpha_c \geq 1$ represents the level of computation load imbalance (r_c) tolerated.

Communication load balance. The communication load captures the flow of traffic from the splitter to each one of the channels. Let $L_n(i)$ denote the communication load of a node $i \in [1..N]$. We have

$$L_n(i) = \sum_{d \in \mathcal{D} \text{ s.t. } p(d)=i} f(d) \tag{5}$$

This is same as having $\beta_n(x) = x$ as a fixed, linear resource function for the communication load. We express the communication load balance requirement as follows:

$$r_n = \frac{\max_{i \in [1..N]} L(i)}{\min_{i \in [1..N]} L(i)} \leq \alpha_n \tag{6}$$

Here, $\alpha_n \geq 1$ represents the level of communication load imbalance (r_n) tolerated.

Discussion. When one of the channels become the bottleneck for a particular resource k, then the utilization of resources for other channels is lower-bounded by α_{k}^{-1}. For instance, if we do not want any channel to be utilized less than 90 % when one of the channels hits 100 %, then we can set $\alpha_c = 1/0.9 = 1.11$.

Another way to look at this is to consider the capacities of different kind of resources. For instance, if the total memory requirement is $x = 10$GB and if each channel ($N = 4$) has a capacity for $y = 3$GB amount of state ($y > x/N$), then α_s can be set as $\frac{(N-1)\cdot y}{x-y} = \frac{3\cdot 3}{10-3} = 1.28$ to avoid hitting the memory bottleneck.

3.2 Structural properties

Structural properties deal with the size of the partitioning function and its lookup cost. In summary, compactness and fast lookup are desirable properties.

Compactness. Let $|p|$ be the size of the partitioning function in terms of the space required to implement the routing, and let $|\mathcal{D}|$ be the domain size for the partitioning key, that is the number of unique values for it. The partitioning function should be compact so that it can be stored at the splitter and also at the parallel channels (for migration [9]). As an example, uniform hashing requires $\mathcal{O}(1)$ space, whereas consistent hashing requires $\mathcal{O}(N)$ space, both of which are acceptable (since $N << |\mathcal{D}|$). However, such partitioning schemes cannot meet the balance requirements we have outlined, as they do not differentiate between items with varying frequencies and do not consider the relationship between frequencies and the amount of memory, computation, and communication incurred.

To address this, our partitioning function has to keep around mappings for different partitioning key values. However, this is problematic, since $|\mathcal{D}|$ could be very large, such as the list of all IP addresses. As a result, we have the following desideratum:

$$|p| = \mathcal{O}(\log |\mathcal{D}|) \tag{7}$$

The goal is to keep the partitioning function small in terms of its space requirement, so that it can be stored in memory even if the domain of the partitioning key is very large. This way the partitioning can be implemented at streaming speeds and does not consume memory resources that are better utilized by the streaming analytics.

Fast lookup. Since a lookup is going to be performed for each tuple τ to be routed, this operation should be fast. In particular, we are interested in $\mathcal{O}(1)$ lookup time.

3.3 Adaptation properties

Adaptation properties deal with updating the partitioning function. The partitioning function needs to be updated when the number of parallel channels changes or when the item frequencies change.

Fast computation. The reconstruction of the partitioning function should take reasonable amount of time so as not to interrupt the continuous nature of the processing. Given the logarithmic size requirement for the partitioning function, we want the computation time of p, denoted by $C(p)$, to be polynomial in terms of the function size:

$$C(p) = poly(|p|) \tag{8}$$

Minimal migration. One of the most critical aspects of adaptation is the migration cost. Migration happens when the balance constraints are violated due to changes in the frequencies of the items or when the number of nodes in the system (N) is increased/decreased in order to cope with the workload dynamics. Changing the partitioning results in migrating state for those partitioning key values whose mapping has changed.

The amount of state to be migrated is given by

$$M(p^{(t)}, p^{(t+1)}) = \sum_{d \in \mathcal{D}} \beta_s(f(d)) \cdot \mathbf{1}(p^{(t)}(d) \neq p^{(t+1)}(d)) \tag{9}$$

Here, $\mathbf{1}$ is the indicator function.

3.4 Overall goal

The goal of the partitioning function creation can be stated in alternative ways. We first look at a few ways that are not flexible enough for our purposes.

One approach is to minimize the migration cost $M(p^{(t)}, p^{(t+1)})$, while treating the balance conditions as hard constraints. However, when the skew in the distribution of the partitioning key is high and the number of channels is large, we will end up with infeasible solutions. Ideally, we should have a formulation that could provide a best effort solution when the constraints cannot be met exactly.

Another approach is to turn the migration cost into a constraint, such as $M(p^{(t)}, p^{(t+1)}) \leq \gamma \cdot \overline{L}_s$. Here, $\overline{L}_s$ is the ideal migration cost with respect to adding a new channel, given as follows:

$$\overline{L}_s = \sum_{d \in \mathcal{D}} \frac{\beta_s(f(d))}{N} \tag{10}$$

We can then set the goal as minimizing the load imbalance. In this alternative, we treat migration as the hard constraint. The problem with this formulation is that it is hard to guess a good threshold (γ) for the migration constraint. For skewed datasets, one might sacrifice more with respect to migration (higher γ) in order to achieve good balance.

In this paper, we use a more flexible approach where both the balance and the migration are treated as part of the objective function. We first define *relative load imbalance*, denoted as b, as follows:

$$b = \left(\prod_{\mathsf{k} \in \{s,c,n\}} b_{\mathsf{k}} \right)^{\frac{1}{3}}, \quad \text{where } b_{\mathsf{k}} = \frac{r_{\mathsf{k}}}{\alpha_{\mathsf{k}}} \tag{11}$$

Here, b_{k} is the relative imbalance for resource k. A value of 1 for b_{k} means that the imbalance for resource k, that is r_{k}, is equal to the acceptable limit α_{k}. Values greater than 1 imply increased imbalance beyond the acceptable limit. The overall relative load imbalance b is defined as the geometric mean of the per-resource relative imbalances.

We define the *relative migration cost*, denoted as m, as follows:

$$m = \frac{M(p^{(t)}, p^{(t+1)})}{\overline{L}_s} \tag{12}$$

A value of 1 for it means that the migration cost is equal to the ideal value (what consistent hashing guarantees, for non-skewed datasets). Larger values imply increased migration cost beyond the ideal. An objective function can then be defined as a combination of relative load imbalance b and relative migration cost m, such as

$$b \cdot (1 + m) \tag{13}$$

In the next section, as part of our solution, we introduce several metrics that consider different trade-offs regarding migration and balance.

4 Solution

In this section, we look at our solution, which consists of a partitioning function structure and a set of heuristic algorithms to construct partitioning functions that follow this structure.

4.1 Partitioning function structure

We structure the partitioning function as a hash pair, denoted as $p = \langle \mathcal{H}_p, \mathcal{H}_c \rangle$. The first hash function $\mathcal{H}_p$ is an *explicit hash*. It keeps a subset of the partitioning key values, denoted as $D_p \subset \mathcal{D}$. For each value, its mapping to the index of the parallel channel that will host the state associated with the value is kept in the explicit hash. We define $D_p = \{d \in \mathcal{D} \mid f(d) \geq \delta\}$. In other words, the partitioning key values whose frequencies are beyond a threshold δ are stored explicitly. We investigate how δ can be set automatically later in this section. The second hash function, $\mathcal{H}_c$, is a *consistent hash* function for N channels. The size of the partitioning function is proportional to the size of the set D_p, that is $|p| \propto |D_p|$.

Algorithm 1: LOOKUP(p, τ)

```
Param : $p = \langle \mathcal{H}_p, \mathcal{H}_c \rangle$, the partitioning function
Param : $\tau \in S$, a tuple in stream $S$
$d \leftarrow \iota(\tau)$                    ▷ Extract the partition by attribute
if $\mathcal{H}_p(d) \neq nil$ then            ▷ Lookup from the explicit hash
    return $\mathcal{H}_p(d)$                  ▷ Return the mapping if found
return $\mathcal{H}_c(d)$                      ▷ Otherwise, fall back to consistent hash
```

4.1.1 Performing lookups

The lookup operation, that is $p(d)$ for $d \in \mathcal{D}$, is carried out by first performing a lookup $\mathcal{H}_p(d)$. If an index is found from the explicit hash, then it is returned as the mapping. Otherwise, a second lookup is performed using the consistent hash, that is $\mathcal{H}_c(d)$, and the result is returned. This is shown in Algorithm 1. It is easy to see that lookup takes $\mathcal{O}(1)$ time as long as the consistent hash is implemented in $\mathcal{O}(1)$ time. We give a brief overview of consistent hashing next. Details can be found in [15].

Consistent hashing. A consistent hash is constructed by mapping each node (parallel channel in our context) to multiple representative points, called *replicas*, in the unit circle, using a uniform hash function. Using a 128-bit ring for representing the unit circle is a typical implementation technique, which relies on 2^{128} equi-spaced discrete locations to represent the range $[0, 1)$. The resulting ring with multiple replicas for each node forms the consistent hash. To perform a lookup on the consistent hash, a given data item is mapped to a point on the same ring using a uniform hash function. Then, the node that has the closest replica (in clockwise direction) to the data point is returned as the mapping. Consistent hashing has several desirable features. Two are particularly important for us. First, it balances the number of items assigned to each node, that is, each node gets around $1/N$th of all the items. Second, when a node is inserted/removed, it minimizes the number of items that move. For instance, the newly added node, say the Nth one, gets $1/N$th of all the items.[1] These properties hold when the number of replicas is sufficiently large. Consistent hashing can be implemented in $\mathcal{O}(\log(N))$ time using a binary search tree over the replicas. Bucketing the ring is an implementation technique that can reduce the search cost to $\mathcal{O}(1)$ time [16], meeting our lookup requirements.

[1] Consistent hash only migrates items from the existing nodes to the newly added node. No migrations happen between existing nodes.

4.1.2 Keeping track of frequencies

Another important problem to solve is to keep track of items with frequency larger than δ. This is needed for constructing the explicit hash $\mathcal{H}_p$. The trivial solution is to simply count the number of appearances of each value for the partitioning key. However, this would require $\mathcal{O}(|\mathcal{D}|)$ space, violating the compactness requirement of the partitioning function.

For this purpose, we use the *lossy counting* technique, which can track items with frequency greater than $\delta - \epsilon$ by using logarithmic space in the order of $\mathcal{O}\left(\frac{1}{\epsilon} \cdot \log(\epsilon \cdot M)\right)$, where M is the size of the history over which the lossy counting is applied. A typical value for ϵ is 0.1 [18]. We can take M as a constant factor of the domain size $|\mathcal{D}|$, which would give us a space complexity of $\mathcal{O}\left(\frac{1}{\delta} \cdot \log(\delta \cdot |\mathcal{D}|)\right)$. We briefly outline how lossy counting works next. The details can be found in [18].

Lossy counting. This is a sketch-based [5] technique that only keeps around logarithmic state in the stream size to locate frequent items. The approach is lossy in the sense that it returns items whose frequencies may be less than the desired level δ, where ϵ is used as a bound on the error. That is, the items with frequencies greater than δ are guaranteed to be returned, where additional items with frequencies in the range $(\delta - \epsilon, \delta]$ may be returned as well. The algorithm operates by adding newly seen items into memory, and evicting some items when a window boundary is reached. The window size is set as $w = 1/\epsilon$. Two values are kept in memory for each item: an appearance count, c_a, and an error count c_e. When an item that is not currently in memory is encountered, it is inserted into memory with $c_a = 1$ and $c_e = i - 1$, where i is the current window index (starts from 1). When the ith window closes, items whose count sums $c_f + c_e$ are less than or equal to i are evicted (these are items whose frequencies are less than ϵ). When frequent items are requested, all items in memory whose appearance counts c_a are greater or equal to $\delta - \epsilon$ times the number of items so far are returned. This simple method guarantees the error bounds and space requirements outlined earlier.

Handling changes. The lossy counting algorithm works on the entire history of the stream. However, typically we are interested in the more recent history. This helps us capture changes in the frequency distribution. There are extensions of the lossy counting algorithm that can handle this via sliding windows [2]. However, these algorithms have more complex processing logic and more involved error bounds and space complexities. We employ a pragmatic approach to support tracking the more recent data items. We achieve this by emulating a sliding window using 3 lossy counters built over tumbling windows as shown in Fig. 2. In the figure, we show the time frame during which a lossy counter is used in dark

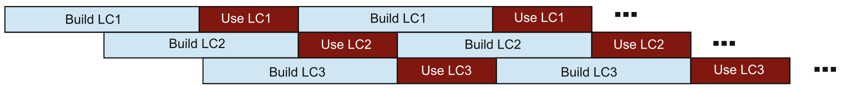


Fig. 2 Using three lossy counters over tumbling windows to emulate a sliding window

color and the time frame during which it is built in light color. Let W be the tumbling window size. This approach makes sure that the lossy counter we use at any given time always has between W and $\frac{3}{2} \cdot W$ items in it.[2] In general, if we use x lossy counters, this technique can achieve an upper range value of $\left(1 + \frac{1}{x-1}\right) \cdot W$, getting closer to a true sliding window of size W as x increases.

4.1.3 Setting δ

To set δ, we first look at how much the load on a channel can deviate from the ideal load, given the imbalance threshold. For a resource $\mathsf{k} \in \{s, c, n\}$, the balance constraint implies the following:

$$\forall_{i \in [1..N]}, |L_\mathsf{k}(i) - \overline{L}_\mathsf{k}| \leq \theta_\mathsf{k} \cdot \overline{L}_\mathsf{k}, \tag{14}$$

where

$$\theta_\mathsf{k} = (\alpha_\mathsf{k} - 1) \cdot \left(1 + \frac{\alpha_\mathsf{k}}{N-1}\right)^{-1} \tag{15}$$

Here, $\overline{L}_\mathsf{k} = \sum_{i=1}^{N} L_\mathsf{k}(i)/N$ is the average load per channel. The gap between the min and max loads is maximized when one channel has the max load $\alpha_\mathsf{k} \cdot x$, and all other channels has the min load x. Thus, we have $x \cdot (\alpha_\mathsf{k} + N - 1) = N \cdot \overline{L}_\mathsf{k}$. Solving for x gives $x = (N \cdot \overline{L}_\mathsf{k})/(\alpha_\mathsf{k} + N - 1)$. Setting $\theta_\mathsf{k} = (\alpha_\mathsf{k} \cdot x - \overline{L}_\mathsf{k})/\overline{L}_\mathsf{k}$ leads to $\theta_\mathsf{k} = (\alpha_\mathsf{k} \cdot N)/(\alpha_\mathsf{k} + N - 1) - 1$, which simplifies to Eq. 15.

Since we do not want to be tracking items with frequencies less than δ and rely on the consistent hash to distribute those items, in the worst case, we can have a single item with frequency δ, resulting in $\beta_\mathsf{k}(\delta)$ amount of load to be assigned to one channel. We set delta such that the imbalanced load $\beta_\mathsf{k}(\delta)$ that can be created due to not tracking some items is $\sigma \in (0, 1]$ fraction of the maximum allowed imbalanced load $\theta_\mathsf{k} \cdot \overline{L}_\mathsf{k}$. This leads to the following definition:

$$\forall_\mathsf{k}, \beta_\mathsf{k}(\delta_\mathsf{k}) \leq \sigma \cdot \theta_\mathsf{k} \cdot \overline{L}_\mathsf{k} \tag{16}$$

Then, the δ can be computed as the minimum of δ_k values for different resources, that is $\delta = \min_{\mathsf{k} \in \{s,c,n\}} \delta_\mathsf{k}$. Considering different β functions, we have

$$\delta_\mathsf{k} = \begin{cases} 1 & \text{if } \beta_\mathsf{k}(x) = 1 \\ \frac{\sigma \cdot \theta_\mathsf{k}}{N} & \text{if } \beta_\mathsf{k}(x) = x \\ \sqrt{\frac{\sigma \cdot \theta_\mathsf{k}}{|\mathcal{D}| \cdot N}} & \text{if } \beta_\mathsf{k}(x) = x^2 \end{cases} \tag{17}$$

[2] The lower bound does not hold during system initialization, as there is not enough history to use.

For $\beta_\mathsf{k}(x) = 1$, the result from Eq. 17 follows, since we have $\beta_\mathsf{k}(\delta_\mathsf{k}) = 1$ and $\overline{L}_\mathsf{k} = |\mathcal{D}|$, thus $\delta_\mathsf{k} = 1$. This is the ideal case, as we do not need to track any items, in which case our partitioning function reduces to the consistent hash.

For $\beta_\mathsf{k}(x) = x$, we have $\overline{L}_\mathsf{k} = 1/N$ (since the frequencies sum up to 1) and thus $\delta_\mathsf{k} = \sigma \cdot \theta_\mathsf{k}/N$.

For $\beta_\mathsf{k}(x) = x^2$, $\overline{L}_\mathsf{k}$ is upper bounded by $1/|\mathcal{D}|$ and thus $\delta_\mathsf{k} = \sqrt{(\sigma \cdot \theta_\mathsf{k})/(|\mathcal{D}| \cdot N)}$. However, the upper bound on $\overline{L}_\mathsf{k}$ is reached when all the items have the same frequency of $1/|\mathcal{D}|$, in which case there is no need to track the items, as consistent hashing would do a perfect job at balancing items with minimal migration cost when all items have the same frequency. Using Eq. 17 for the case of quadratic beta functions results in a low δ value and thus large number of items to be tracked. This creates a problem in terms of the time it takes to construct the partitioning function, especially for polynomial construction algorithms that are superlinear in the number of items used (discussed in Sect. 4.2).

To address this issue, we use a post-processing step for the case of quadratic beta functions. After we collect the list of items with frequency at least δ, say I, we predict $\overline{L}_\mathsf{k}$ as $\sum_{d \in I} \beta_\mathsf{k}(f(d)) + (|\mathcal{D}| - |I|) \cdot \beta_\mathsf{k}(\frac{1 - \sum_{d \in I} f(d)}{|\mathcal{D}| - |I|})$. The second part of the summation is a worst case assumption about the untracked items, which maximizes the load. Using the new approximation for $\overline{L}_\mathsf{k}$, we compute an updated δ', which is higher than the original δ, and use it to filter the data items to be used for constructing the partitioning function.

4.1.4 Setting σ

σ is the only configuration parameter of our solution for creating partitioning functions, which is not part of the problem formulation. We study its sensitivity as part of the experimental study in Sect. 5. A value of $\sigma = 0.1$, which is a sensible setting, would allocate one-tenth of the allowable load imbalance to the untracked items, leaving the explicit hash construction algorithm enough room for imbalance in the mapping. The extreme setting of $\sigma = 1$ would leave the explicit hash no flexibility and should be avoided, since in a skewed setting, the explicit hash cannot achieve perfect balance.

4.2 Construction algorithms

We now look at algorithms for constructing the partitioning function. In summary, the goal is to use the partitioning function created for time period t, that is $p^{(t)} = \langle \mathcal{H}_p^{(t)}, \mathcal{H}_c^{(t)} \rangle$, and recent item frequencies f, to create a new partitioning function to use during time period $t+1$, that is $p^{(t+1)} = \langle \mathcal{H}_p^{(t+1)}, \mathcal{H}_c^{(t+1)} \rangle$, given the number of parallel channels has changed from $N^{(t)}$ to $N^{(t+1)}$.

We first define some helper notation that will be used in all algorithms. Recall that $D_p^{(t)}$ and $D_p^{(t+1)}$ denote the items with explicit mappings in $p^{(t)}$ and $p^{(t+1)}$, respectively. We define the following additional notation:

- The set of items not tracked for time period $t+1$ but tracked for time period t is denoted as $D_o^{(t+1)} = D_p^{(t)} \setminus D_p^{(t+1)}$.
- The set of items tracked for time period $t+1$ but not tracked for time period t is denoted as $D_n^{(t+1)} = D_p^{(t+1)} \setminus D_p^{(t)}$.
- The set of items tracked for both time period t and $t+1$ are denoted as $D_e^{(t+1)} = D_p^{(t+1)} \cap D_p^{(t)}$.
- The set of items tracked for time period t or $t+1$ are denoted as $D_a^{(t_1)} = D_p^{(t)} \cup D_p^{(t+1)}$.

We develop three heuristic algorithms, namely the *scan*, the *redist*, and the *readj* algorithms. They all operate on the basic principle of assigning tracked items to parallel channels considering a *utility function* that combines two metrics: the relative imbalance and the relative migration cost. The algorithms are heuristic in the sense that at each step, they compute the utility function on the partially constructed partitioning function with different candidate mappings applied and, at the end of the step add, the candidate mapping that maximizes the utility function. The three algorithms differ in how they define and explore the candidate mappings. Before looking at each algorithm in detail, we first detail the metrics used as the basis for the utility function.

4.2.1 Metrics

We use a slightly modified version of the relative migration cost m given by Eq. 12 in our utility function, called the *migration penalty* and denoted as γ. In particular, the migration cost is computed for the items that are currently in the partially constructed partitioning function, and this value is normalized using the ideal migration cost considering all items tracked for time periods t and $t+1$. Formally, for a partially constructed explicit hash $\mathcal{H}_p^{(t+1)}$, we define

$$\gamma(\mathcal{H}_p^{(t+1)}) = \frac{\begin{array}{c}\sum_{d \in D_o^{(t+1)}} \beta_s(f(d)) \cdot \mathbf{1}(p^{(t)}(d) \neq \mathcal{H}_c^{(t+1)}(d)) \\ + \sum_{d \in \mathcal{H}_p^{(t+1)}} \beta_s(f(d)) \cdot \mathbf{1}(p^{(t)}(d) \neq \mathcal{H}_p^{(t+1)}(d))\end{array}}{\sum_{d \in D_a^{(t+1)}} \beta_s(f(d))/N^{(t+1)}} \tag{18}$$

Here, the first part in the numerator is the migration cost due to items not being tracked anymore ($D_o^{(t+1)}$). Such items cause migration if the old partitioning function ($p^{(t)}$) and the new consistent has ($\mathcal{H}_c^{(t+1)}$) map the items to different parallel channels. The second part in the numerator is due to the items that are currently in the partially constructed explicit hash ($\mathcal{H}_p^{(t+1)}$), but map to a different parallel channel than before (based on $p^{(t)}$). The denominator is the ideal migration cost, considering items tracked for time periods t and $t+1$ ($D_a^{(t+1)}$).

Similarly, we use a modified version of the relative imbalance b given in Eq. 11 in our utility function, called the *balance penalty* and denoted as ρ. This is because a partially constructed partitioning function yields a b value of ∞ when one of the parallel channels does not yet have any assignments. Instead, we use a very similar definition, which captures the imbalance as the ratio of the difference between the max and min loads to the maximum load difference allowed. Formally, for a partially constructed explicit hash $\mathcal{H}_p^{(t+1)}$, we have

$$\rho_{\mathsf{k}}(\mathcal{H}_p^{(t+1)}) = \frac{\begin{array}{c}\max_{i \in [1..N^{(t+1)}]} L_{\mathsf{k}}(i, \mathcal{H}_p^{(t+1)}) \\ - \min_{i \in [1..N^{(t+1)}]} L_{\mathsf{k}}(i, \mathcal{H}_p^{(t+1)})\end{array}}{\theta_{\mathsf{k}} \cdot \overline{L}_{\mathsf{k}}(\mathcal{H}_p^{(t+1)})} \tag{19}$$

$$\rho(\mathcal{H}_p^{(t+1)}) = \left(\prod_{\mathsf{k} \in \{s,c,n\}} \rho_{\mathsf{k}}(\mathcal{H}_p^{(t+1)}) \right)^{\frac{1}{3}} \tag{20}$$

In Eq. 19, $L_{\mathsf{k}}(i, \mathcal{H}_p^{(t+1)})$ values represent the total load on channel i for resource k, considering only the items that are in $\mathcal{H}_p^{(t+1)}$. Similarly, $\overline{L}_{\mathsf{k}}(\mathcal{H}_p^{(t+1)})$ is the average load for resource k, considering only the items that are in $\mathcal{H}_p^{(t+1)}$.

Given the ρ and γ values for a partially constructed partitioning function, our heuristic algorithms pick a mapping to add into the partitioning function, considering a set of candidate mappings. A utility function $U(\rho, \gamma)$ is used to rank the potential mappings. We investigate such utility functions at the end of this section.

Construction algorithms start from an empty explicit hash, and thus with a low γ value. As they progress, γ typically increases and thus mappings that require migrations become less and less likely. This provides flexibility in achieving balance early on, by allowing more migrations early. On the other hand, ρ is kept low throughput the progress of the algorithms; as otherwise, in the presence of skew, fixing imbalance introduced early on may be difficult to fix later.

We now look at the construction algorithms.

4.2.2 The scan algorithm

The scan algorithm, shown in Algorithm 2, first performs a few steps that are common to all three algorithms: Creates a new consistent hash for $N^{(t+1)}$ parallel channels as $\mathcal{H}_c^{(t+1)}$, computes the migration cost (variable m in the algorithm) due to items not tracked anymore, as well as the ideal

Algorithm 2: SCAN($p^{(t)}$, $D_p^{(t)}$, $D_p^{(t+1)}$, $N^{(t+1)}$, f)

```
Param : $p^{(t)} = \langle \mathcal{H}_p^{(t)}, \mathcal{H}_c^{(t)} \rangle$, Current partitioning function
Param : $D_p^{(t)}$, $D_p^{(t+1)}$, Items tracked during period $t$, $t+1$
Param : $N^{(t+1)}$, New number of parallel channels
Param : $f$, Item frequencies
Let $p^{(t+1)} = \langle \mathcal{H}_p^{(t+1)}, \mathcal{H}_c^{(t+1)} \rangle$        ▷ Next partitioning function
$\mathcal{H}_c^{(t+1)} \leftarrow$ CREATECONSISTENTHASH($N^{(t+1)}$)
▷ Migration cost due to items not being tracked anymore
$m \leftarrow \sum_{d \in D_o^{(t+1)}} \beta_s(f(d)) \cdot \mathbf{1}(p^{(t)} \neq \mathcal{H}_c^{(t+1)}(d))$
$\overline{m} \leftarrow \sum_{d \in D_a^{(t+1)}} \beta_s(f(d))/N^{(t+1)}$        ▷ Ideal migration cost
$\mathcal{H}_p^{(t+1)} \leftarrow \{\}$        ▷ The mapping is initially empty
$D_c \leftarrow$ SORT($D_p^{(t+1)}$, $f$)        ▷ Items to place, in decr. freq. order
for each $d \in D_c$ do        ▷ For each item to place
    $j \leftarrow -1$        ▷ Best placement, initially invalid
    $u \leftarrow \infty$        ▷ Best utility value, lower is better
    $h \leftarrow p^{(t)}(d)$        ▷ Old location
    for each $l \in [1..N^{(t+1)}]$ do        ▷ For each placement
        $a \leftarrow \rho(\mathcal{H}_p^{(t+1)} \cup \{d \Rightarrow l\})$        ▷ Balance penalty
        $\gamma \leftarrow \frac{m+\beta_s(f(d))\cdot\mathbf{1}(l\neq h)}{\overline{m}}$        ▷ Migration penalty
        if $U(a,\gamma) < u$ then        ▷ A better placement
            $j, u \leftarrow l, U(a,\gamma)$        ▷ Update best
    $m \leftarrow m + \beta_s(f(d)) \cdot \mathbf{1}(j \neq h)$        ▷ New migration cost
    $\mathcal{H}_p^{(t+1)} \leftarrow \mathcal{H}_p^{(t+1)} \cup \{d \Rightarrow j\}$        ▷ Add the mapping
```

Algorithm 3: REDIST($p^{(t)}$, $D_p^{(t)}$, $D_p^{(t+1)}$, $N^{(t+1)}$, f)

```
Param : $p^{(t)} = \langle \mathcal{H}_p^{(t)}, \mathcal{H}_c^{(t)} \rangle$, Current partitioning function
Param : $D_p^{(t)}$, $D_p^{(t+1)}$, Items tracked during period $t$, $t+1$
Param : $N^{(t+1)}$, New number of parallel channels
Param : $f$, Item frequencies
Let $p^{(t+1)} = \langle \mathcal{H}_p^{(t+1)}, \mathcal{H}_c^{(t+1)} \rangle$        ▷ Next partitioning function
$\mathcal{H}_c^{(t+1)} \leftarrow$ CREATECONSISTENTHASH($N^{(t+1)}$)
▷ Migration cost due to items not being tracked anymore
$m \leftarrow \sum_{d \in D_o^{(t+1)}} \beta_s(f(d)) \cdot \mathbf{1}(p^{(t)} \neq \mathcal{H}_c^{(t+1)}(d))$
$\overline{m} \leftarrow \sum_{d \in D_a^{(t+1)}} \beta_s(f(d))/N^{(t+1)}$        ▷ Ideal migration cost
$\mathcal{H}_p^{(t+1)} \leftarrow \{\}$        ▷ The mapping is initially empty
while $|D_c| > 0$ do        ▷ While not all placed
    $j \leftarrow -1$        ▷ Best placement
    $d \leftarrow \emptyset$        ▷ Best item to place
    $u \leftarrow \infty$        ▷ Best utility value
    for each $c \in D_c$ do        ▷ For each candidate
        $h \leftarrow p^{(t)}(c)$        ▷ Old location
        for each $l \in [1..N^{(t+1)}]$ do        ▷ For each placement
            $a \leftarrow \rho(\mathcal{H}_p^{(t+1)} \cup \{c \Rightarrow l\})$        ▷ Balance penalty
            $\gamma \leftarrow \frac{m+\mathbf{1}(l\neq h)\cdot\beta_s(f(c))}{\overline{m}}$        ▷ Migration penalty
            $u' \leftarrow U(a,\gamma)/f(c)$        ▷ Placement utility
            if $u' < u$ then        ▷ Better placement
                $j, d, u \leftarrow l, c, u'$        ▷ Update best
    $m \leftarrow m + \mathbf{1}(j \neq h) \cdot \beta_s(f(d))$        ▷ New migration cost
    $\mathcal{H}_p^{(t+1)} \leftarrow \mathcal{H}_p^{(t+1)} \cup \{d \Rightarrow j\}$        ▷ Add the mapping
```

migration cost (variable $\overline{m}$ in the algorithm) considering all items tracked for time periods t and $t+1$. Then, the algorithm moves on to perform the scan-specific operations. The first of these is to sort the items in decreasing order of frequency. Then, it scans the sorted items and inserts a mapping into the explicit hash for each item, based on the placement that provides the best utility function value (lower is better). As a result, for each item, starting with the one that has the highest frequency, it considers all possible $N^{(t+1)}$ placements. For each placement, it computes the balance and migration penalties to feed the utility function.

Note that, the migration penalty can be updated incrementally in constant time (shown in the algorithm). The balance penalty can be updated in $\mathcal{O}(\log(N))$ time using balanced trees, as it requires maintaining the min and max loads. However, for small N, explicit computation as shown in the algorithm is faster. The complexity of the algorithm is $\mathcal{O}(R \cdot N \cdot \log N)$, where $R = |D_p^{(t+1)}|$ is the number of items tracked.

The scan algorithm considers the items in decreasing order of frequency, since items with higher frequencies are harder to compensate for unless they are placed early on during the construction process.

4.2.3 The redist algorithm

The redist algorithm, shown in Algorithm 3, works in a similar manner to the scan algorithm, that is, it distributes the items over the parallel channels. However, unlike the scan algorithm, it does not pick the items to place in a pre-defined order. Instead, at each step, it considers all unplaced items and for each item all possible placements. For each placement, it computes the utility function and picks the placement with the best utility (u' in the algorithm). The redist algorithm uses the inverse frequency of the item to scale the utility function, so that we pick the item that brings the best utility per volume moved. This results in placing items with higher frequencies early. While this is similar to the scan algorithm, in the redist algorithm, we have additional flexibility, as an item with a lower frequency can be placed earlier than one with a higher frequency, if the former's utility value ($U(a,\gamma)$ in the algorithm) is sufficiently lower.

The additional flexibility provided by the redist algorithm comes at the cost of increased computational complexity, which is given by $\mathcal{O}(R^2 \cdot N \cdot \log N)$ (again, R is the number of items tracked). This follows as there are R steps (the outer while loop), where at the ith step placement of $R - i$ items (first for loop) over N possible parallel channels (second for loop) are considered, with $\log N$ being the cost of computing the utility for each placement (not shown in the algorithm, due to ρ maintenance as discussed earlier).

4.2.4 The readj algorithm

The readj algorithm is based on the idea of readjusting the item placements rather than making brand new placements.

It removes the items that are not tracked anymore ($D_o^{(t+1)}$) from the explicit hash and adds the ones that are now tracked ($D_n^{(t+1)}$) based on their old mappings (using $\mathcal{H}_c^{(t)}$). This results in a partial explicit hash that only uses $N^{(t)}$ parallel channels. Here, it is assumed that $N^{(t)} \leq N^{(t+1)}$. Otherwise, the items from channels that are not existing anymore can be assigned to exiting parallel channels using $\mathcal{H}_c^{t+1}$. The readj algorithm then starts making readjustments to improve the partitioning. The readjustment continues until there are no readjustments that improve the utility.

The readjustments that are attempted by the readj algorithm are divided into two kinds: *moves* and *swaps*. We represent a readjustment as $\langle i, d_1, j, d_2\rangle$. If $d_2 = \emptyset$, then this represents a move, where item d_1 is moved from the ith parallel channel to the jth parallel channel. Otherwise, ($d_2 \neq \emptyset$) represents a swap, where item d_1 from the ith parallel channel is swapped with item d_2 from the jth parallel channel. Given a readjustment $\langle i, d_1, j, d_2\rangle$ and the explicit hash $\mathcal{H}_p^{(t+1)}$, the readjustment is applied as follows:

$$A(\mathcal{H}_p^{(t+1)}, \langle i, d_1, j, d_2\rangle) = \begin{cases} \mathcal{H}_p^{(t+1)} \setminus \{d_1 \Rightarrow i\} \cup \{d_1 \Rightarrow j\} & \text{if } d_2 = \emptyset \\ \mathcal{H}_p^{(t+1)} \setminus \{d_1 \Rightarrow i, d_2 \Rightarrow j\} \cup \{d_1 \Rightarrow j, d_2 \Rightarrow i\} & \text{otherwise} \end{cases} \tag{21}$$

Given a readjustment and the old partitioning function $p^{(t)}$, the migration cost incurred by the readjustment is given as follows:

$$\begin{aligned} M(p^{(t)}, \langle i, d_1, j, d_2\rangle) = \\ \beta_s(f(d_1)) \cdot \mathbf{1}(p^{(t)}(d_1) = i) - \beta_s(f(d_1)) \cdot \mathbf{1}(p^{(t)}(d_1) = j) \\ \beta_s(f(d_2)) \cdot \mathbf{1}(p^{(t)}(d_2) = j) - \beta_s(f(d_2)) \cdot \mathbf{1}(p^{(t)}(d_2) = i) \end{aligned} \tag{22}$$

Note that, Eq. 22 could yield a negative value when an item is placed to its old channel as part of a move or swap.

The details of the readj algorithm are given in Algorithm 4. The algorithm considers all pairs of parallel channels, and for each pair, it considers all moves and all swaps that reduce the imbalance penalty. The readjustment that results in the best *gain* in the utility value is applied, unless none can be found. In the latter case, the search terminates. The gain is the reduction in the utility function value per frequency moved. Since the total number of items in the explicit hash is constant for the readj algorithm, the utility values from different steps can be compared, and thus, the difference can be used to compute the gain. Unlike the other algorithms, the readj algorithm has a strong bias toward reducing the load imbalance, as it only considers readjustments that reduce the imbalance, and only uses the utility function for picking the best among those.

There are $\mathcal{O}(N^2)$ pairs of parallel channels and for each pair $\mathcal{O}((R/N)^2)$ possible readjustments. Again assuming

Algorithm 4: READJ($p^{(t)}, D_p^{(t)}, D_p^{(t+1)}, N^{(t+1)}, f$)

```
Param : $p^{(t)} = \langle \mathcal{H}_p^{(t)}, \mathcal{H}_c^{(t)} \rangle$, Current partitioning function
Param : $D_p^{(t)}, D_p^{(t+1)}$, Items tracked during period $t, t+1$
Param : $N^{(t+1)}$, New number of parallel channels
Param : $f$, Item frequencies
Let $p^{(t+1)} = \langle \mathcal{H}_p^{(t+1)}, \mathcal{H}_c^{(t+1)} \rangle$          ▷ Next partitioning function
$\mathcal{H}_c^{(t+1)} \leftarrow$ CREATECONSISTENTHASH($N^{(t+1)}$)
▷ Migration cost due to items not being tracked anymore
$m \leftarrow \sum_{d \in D_o^{(t+1)}} \beta_s(f(d)) \cdot \mathbf{1}(p^{(t)} \neq \mathcal{H}_c^{(t+1)}(d))$
$\overline{m} \leftarrow \sum_{d \in D_a^{(t+1)}} \beta_s(f(d))/N^{(t+1)}$          ▷ Ideal migration cost
▷ Tracked items stay put initially (assume $N$ went up)
$\mathcal{H}_p^{(t+1)} \leftarrow \{d \Rightarrow p^{(t)}(d) : d \in D_p^{(t+1)}\}$
$u \leftarrow 0$          ▷ Last utility value
while true do          ▷ Improvement possible
    $v \leftarrow \emptyset$          ▷ Best readjustment
    $g \leftarrow -\infty$          ▷ Best gain value
    for each $i, j \in [1..N^{(t+1)}]$ s.t. $i \neq j$ do
        for each $d_1, d_2$ s.t. $\mathcal{H}_p^{(t+1)}(d_1) = i \wedge$
                ($\mathcal{H}_p^{(t+1)}(d_2) = j \vee d_2 = \emptyset$) do
            $w \leftarrow \langle i, d_1, j, d_2 \rangle$          ▷ Candidate readjustment
            $a \leftarrow \rho(A(\mathcal{H}_p^{(t+1)}, w))$          ▷ Balance penalty
            if $a \geq \rho(\mathcal{H}_p^{(t+1)})$ then          ▷ Worse balance
                break          ▷ Move on to next option
            $\gamma \leftarrow \frac{m + M(p^{(t)}, w)}{\overline{m}}$          ▷ Migration penalty
            $u' \leftarrow U(a, \gamma)$          ▷ Placement utility
            $g' \leftarrow (u - u')/|f(d_1) - f(d_2)|$          ▷ Placm. gain
            if $g' > g$ then          ▷ Better placement
                $v, g, u \leftarrow w, g', u'$          ▷ Update best
    if $v = \emptyset$ then          ▷ No readjustments with gain
        break          ▷ Terminate the search
    $m \leftarrow m + M(p^{(t)}, v)$          ▷ New migration cost
    $\mathcal{H}_p^{(t+1)} \leftarrow A(\mathcal{H}_p^{(t+1)}, v)$          ▷ Update the mappings
```

that for each readjustment the utility can be computed in $\log N$ time, the complexity of the code within the main loop of the algorithm is given by $\mathcal{O}(R^2 \cdot \log N)$. The number of times the main loop runs can be bounded by limiting the number of times an item can move, say by c, resulting in an overall complexity of $\mathcal{O}(R^3 \cdot \log N)$. This limiting of moves is not shown in Algorithm 4. In our experiments, with a c value of 5, the limited and unlimited versions did not result in any difference, suggesting that the termination condition is reached before the explicit limits put on the number of readjustments allowed per item are hit.

4.3 Utility functions

For the utility function, we consider a number of different ways of combining the imbalance penalty with the migration penalty. The alternatives we consider either give good balance preference over low cost migration or treat them equal. We do not consider alternatives that give migration more importance relative to load balance; as with skewed workloads, it is a bigger challenge to achieve good balance. The various utility functions we consider are listed below:

$$\begin{aligned} U^{\mathrm{A}}(\rho,\gamma) &= \rho \\ U^{\mathrm{APM}}(\rho,\gamma) &= \rho+\gamma \\ U^{\mathrm{APLM}}(\rho,\gamma) &= \rho+\log(1+\gamma) \\ U^{\mathrm{ATM}}(\rho,\gamma) &= \rho\cdot(1+\gamma) \\ U^{\mathrm{ATLM}}(\rho,\gamma) &= \rho\cdot(1+\log(1+\gamma)) \end{aligned}$$

We consider only using the imbalance penalty (U^{A}), summation and multiplication of imbalance and migration penalties (U^{APM} and U^{ATM}, respectively), and variations of the latter two where the migration penalty's impact is logarithmic (U^{APLM} and U^{ATLM}, respectively).

4.4 A note on resource functions

In this paper, we considered three resource functions, that is, constant, linear, and quadratic. These three functions are quite common in windowed operators, as we outlined earlier. For other functions, additional cases need to be added to the Eq. 17. Constant resource functions are special in the sense that they can be balanced without using the explicit hash. Given that a majority of the items are not tracked, load balance comes free for a resource with a constant resource function. As such we do not consider a resource with a constant function in our overall imbalance penalty, so as to give additional flexibility to the construction algorithms.

4.5 Use of partitioning functions

We briefly describe the way partitioning functions are used and updated as part of auto-fission. A stream processing system that supports dynamic adaptation typically employs an *adaptivity loop* [6], which involves the steps of *measure*, *analyze*, *plan*, and *activate*. As part of the measure step, various performance metrics are computed, such as throughput and congestion [9]. The updating of the lossy counter is piggybacked on the measurement step. Concretely, when a new tuple reaches the splitter, its partitioning key value is extracted and the value is run through the sliding lossy counter. This operation takes $\mathcal{O}(1)$ time. The value of the partitioning key is then provided to the partitioning function to locate the parallel channel to use for processing the tuple. This lookup takes $\mathcal{O}(1)$ time as well.

As part of the analysis step, the auto-fission controller decides whether a change in the number of channels is required, typically based on examining the throughput and congestion metrics. If such a change is required, then the planning phase starts, which includes determining the new number of parallel channels to use as well as constructing the new partitioning function, with the aim of maintaining balance and minimizing the migration cost. The final step, activation, involves the mechanics of adding/removing parallel channels and performing the migration of state maintained in partitioned stateful operators that are part of the parallel region whose number of channels is being updated.

4.6 Parameter discussion

Finally, we provide a brief summary of the parameters used in our system, and how they are configured.

N is a system parameter that specifies the number of channels in the parallel region. It is not an exposed parameter and is set automatically by the stream processing runtime, as part of the adaptivity loop.

β_{k} parameters are application parameters that capture the memory/network/processing characteristics of the parallel region. They are not exposed parameters and are set based on the nature of operators that form the parallel region served by the partitioning function.

α_{k} parameters are user parameters that capture the tolerance to memory/network/processing load imbalance. These are exposed to system developers. Optionally, a sensible default (e.g., in [1.1, 1.2]) can be provided as described at the end of Sect. 3.1.

σ is an algorithmic parameter that adjusts the trade-off between space used by the partitioning function and its effectiveness in terms of load balance. While it is exposed to the system developers, a default value of 0.1 is considered a robust setting as described in Sect. 4.1.4 and later studied in Sect. 5.

5 Experimental results

In this section, we present our experimental evaluation. We use four main metrics as part of our evaluation. The first is the relative load imbalance, b, as given in Eq. 11. We also use the per-resource load imbalances, b_{k}, for $\mathsf{k} \in \{s, c, n\}$. The second is the relative migration cost, m, as given in Eq. 12. The third is the space requirement of the partitioning function. We divide this into two: the number of items kept in the lossy counter and the number of mappings used by the explicit hash. The fourth and the last metric is the time it takes to build the partitioning function.

As part of the experiments, we investigate the impact of various workload and algorithmic parameters on the aforementioned metrics. The workload parameters we investigate include resource functions (β_{k}), data skew (z), domain size ($|D|$), number of nodes (N), and the imbalance thresholds (α_{k}).

The algorithmic parameters we investigate include the frequency threshold scaler (σ) and the utility function used (U). These parameters apply to all three algorithms we introduced:

Table 1 Experimental params.: default values, ranges

Description	Default	Range
# of channels, N	10	[1, 100]
Imbalance tol., α	1.2	[1, 4]
Resource functions, β_s, β_c, β_n	Linear, Constant, Linear (LCL)[a]	{CCL,LCL, LLL,LQL}
Domain size, $\lvert\mathcal{D}\rvert$	10^6	$[10^4, 10^8]$
Zipf skew, $z = 1$	1.0	[0.1, 1]
Freq. thres. scaler, σ	0.1	[0.01, 1]
Utility function, U	U^{APM}	$\{U^A, U^{APM}, U^{APLM} U^{ATM}, U^{ATLM}\}$

[a]Letters Q, L, and C represent Quadratic, Linear, and Constant functions, respectively. XYZ is used to mean β_s =X, β_c = Y, β_n = Z, where X, Y, Z are one of Q, L, or C

scan, redist, and readj. We also compare these three algorithms to the uniform and consistent hash approaches.

5.1 Experimental setup

The default values of the parameters we use and their ranges are given in Table 1. To experiment with the skew in the partitioning key values, we use a Zipf distribution. The default skew used is $z = 1$, where the kth most frequent item d_k has frequency $\propto 1/k^z$. The default number of parallel channels is set to 10. This value is set based on our previous study [24], where we used several real-world streaming applications to show scalability of parallel regions. The average number of parallel channels that gave the best throughput over different applications was around 10. As such, we do not change the load. We start with a single channel and keep increasing the number of channels until all the load can be handled.

To test a particular approach for $N^{(t)}$ parallel channels, we start from $N^{(0)} = 1$ and successively apply the partitioning function construction algorithm until we reach $N^{(t)}$, increasing the number of channels by one at each adaptation period, that is, $N^{(t+1)} - N^{(t)} = 1$. We do this because the result of partitioning function at time period $t + 1$ depends on the partitioning function from time period t. As such, the performance of a particular algorithm for a particular number of channels also depends on its performance for lower number of channels.

We set the default imbalance threshold to 1.2. The default resource functions are set as linear, constant, and linear for the state (β_s), computation (β_c), and communication (β_n) resources, respectively. β_n is always fixed as linear (see Sect. 3.1). For the state, the default setting assumes a time-based sliding window (thus $\beta_s(x) = x$). For computation, we assume an aggregation computation that is incremental (thus $\beta_c(x) = 1$). We investigate various other configurations, listed in Table 1. The default utility function is set as U^{APM}, as it gives the best results, as we will report later in this section. Finally, the default domain size is a million items, but we try larger and smaller domain sizes as well.

All the results reported are averages of 5 runs.

5.2 Implementation notes

The partitioning function is implemented as a module that performs three main tasks: *frequency maintenance, lookup*, and *construction*. Both the frequency maintenance and the lookup are implemented in a streaming fashion. When a new tuple is received, the lossy counters are updated, and if needed the active lossy counter is changed. Then, lookup is performed to decide which parallel channel should be used for routing the tuple. The construction functionality is triggered independently, when adaptation is to be performed. The construction step runs one of the algorithms we have introduced, namely one of scan, redist, or readj.

Our particular implementation is in C++ and is designed as a drop-in replacement for the consistent hash used by a fission-based auto-parallelizer [24] built on top of System S [14]. The consistent hashing implementation we use provides $\mathcal{O}(1)$ lookup performance by using the bucketing technique [16]. More concretely, we divide the 128-bit ring into buckets and use a sorted tree within each bucket to locate the appropriate mapping. We rely on MurMurHash3 [19] for hashing. Our experiments were performed on machines with 2× 3GHz Intel Xeon processors containing 4 cores (total of 8 cores) and 64GB of memory. However, partitioning function construction does not take advantage of multiple cores.

5.3 Load balance and migration

We evaluate the impact of algorithm and workload parameters on the load balance and migration.

Impact of resource functions. Figure 3 plots relative migration cost (in log), relative load imbalance, and the individual relative load imbalances for different resources, using radar charts. We have 4 charts, each one for a different resource function combination. The black line marks the ideal area for the imbalance and migration cost (relative values $\leq$ 1). We make a number of observations from the figure.

First, we comment on the relative performance of different algorithms. As expected, the uniform hash results in very

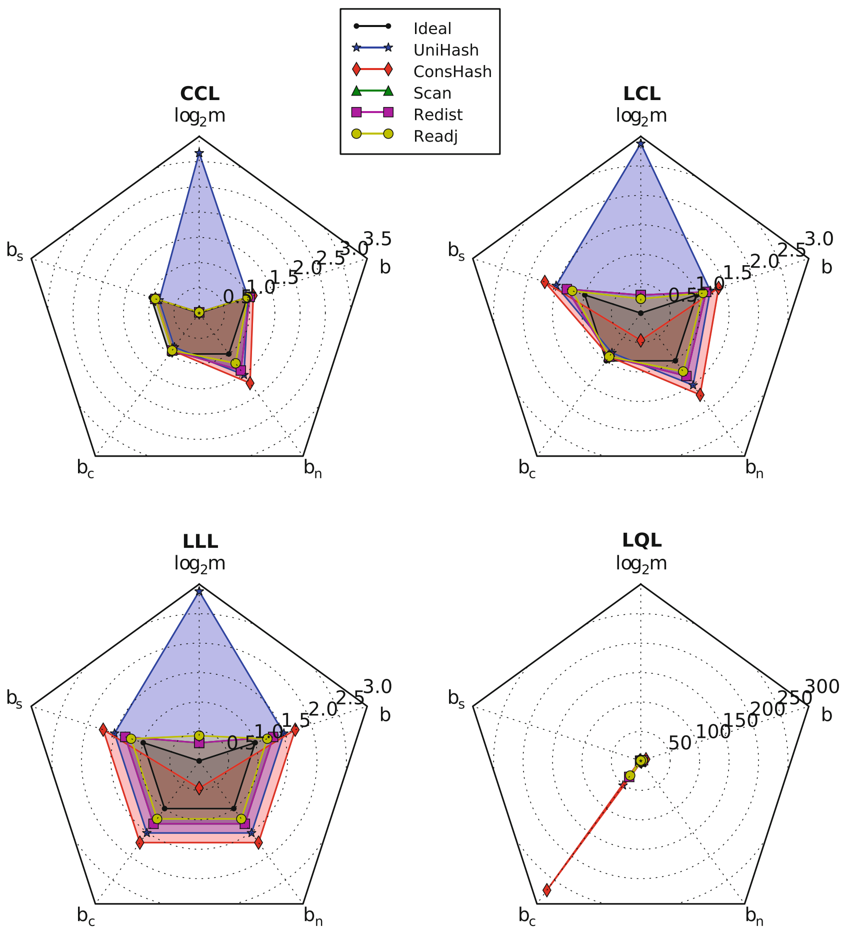


Fig. 3 Impact of resource functions on migration and imbalance, for different algorithms

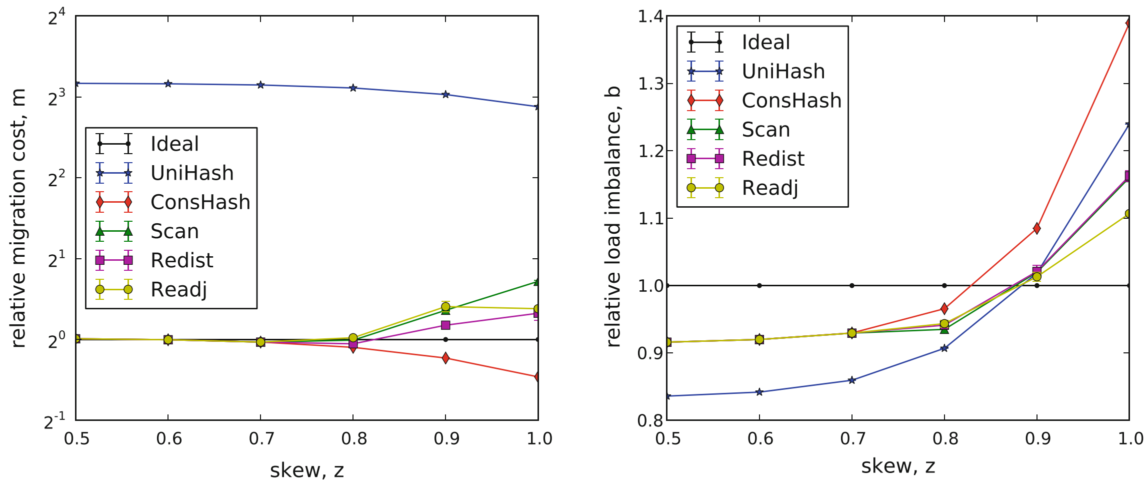
high migration cost, reaching up to more than 8 times the ideal. Consistent hash, on the other hand, has the best migration cost. The relative migration cost for consistent hash is below 1 in some cases. This happens due to skew. When the top few most frequent items do not migrate, the overall migration cost ends up being lower than the ideal. However, consistent hash has the worse balance among all other alternatives. For instance, its balance reaches 1.75 for the case of LLL, compared to 1.55 of uniform hash.

We observe that the readj algorithm provides the lowest relative imbalance, consistently across all resource function settings. The LLL case illustrates this, where relative imbalance is around 1.2 for readj and 1.32 for redist and scan (around 10 % higher). However, readj has a slightly higher relative migration cost, reaching around 1.34 times the ideal for LLL, compared to 1.23 for redist and scan (around 8 % lower). Redist and scan are indistinguishable form each other (in the figure, redist marker shadows the scan marker).

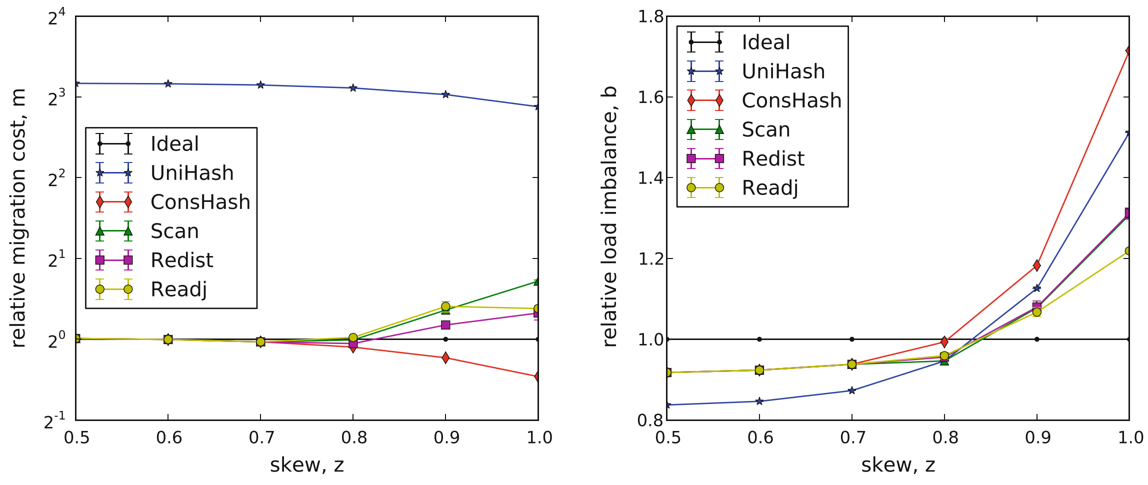
We attribute the good balance properties of the readj algorithm to the large set of combinations it tries out compared with the other algorithms, including swaps of items between channels. The readj algorithm continues as long as an adjustment that improves the placement gain is found. As such it generally achieves better balance. Since balance and migration are at odds, the slight increase in the migration cost with the readj algorithm is expected.

Looking at different combinations of resource functions, it is easy to see that linear and quadratic resource functions are more difficult to balance. In the case of LQL, clearly the computation imbalance cannot be kept under control for the case of consistent hash. Even for the rest of the approaches, the relative computation imbalance is too high (in 30 s). Recall that the Zipf skew is 1 by default. Later in this section, we will look at less skewed scenarios, where good balance can be achieved.

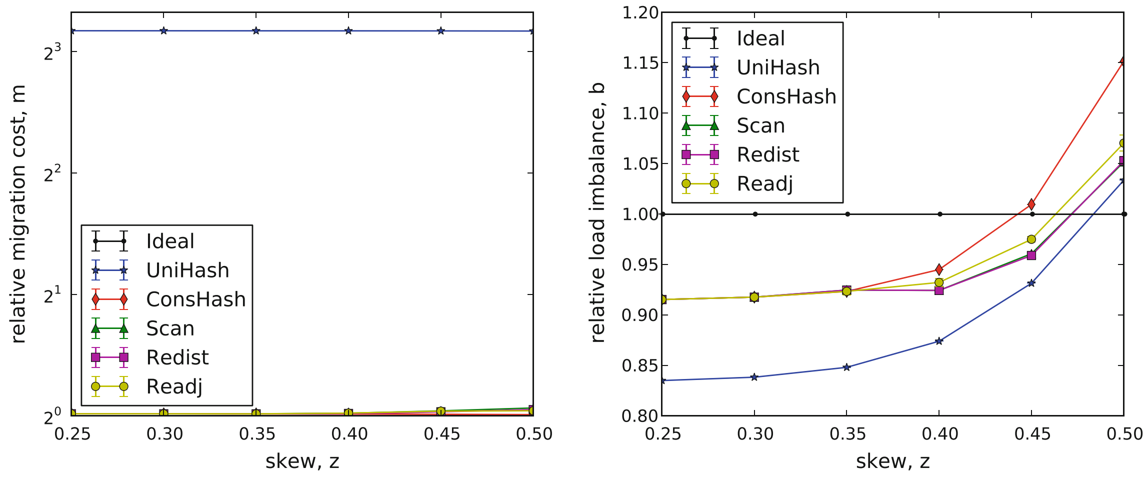
Impact of data skew. The charts in Fig. 4 plot relative migration cost and relative load imbalance as a function of data skew for different algorithms and for different resource function combinations. Each resource function combination is plotted in a separate sub-figure. For the LQL resource combination, the skew range is restricted to [0.25, 0.5], as the imbalances jump up to high numbers as we try higher skews.



(a) For resource functions LCL



(b) For resource functions LLL



(c) For resource functions LQL

Fig. 4 Impact of skew on migration and balance **a** For resource functions LCL **b** For resource functions LLL **c** For resource functions LQL

The most striking observation from the figures is that the uniform hash has a very high migration cost, more than 8 times the ideal. Other approaches have close to ideal migration cost. The migration cost for our algorithms start increasing after the skew reaches $z = 0.8$. Scan has the worst migration cost, readj, and redist following it.

Another observation is that the consistent hash is the first one to start violating the balance requirements (going over the line $y = 1$), as the skew increases. Its relative imbalance is up to 50 % higher compared to the best alternative, for instance, for the LLL resource combination compared to the readj algorithm at skew $z = 1$.

The violations of the balance requirement start earliest for the LQL resource combination and latest for the LCL combination, as the skew is increased. This is expected, as quadratic functions are more difficult to balance compared to linear ones, and linear ones are more difficult compared to constant ones.

For very low skews, all approaches perform acceptably, that is below the ideal line. Relative to others, uniform hash performs the best in terms of the imbalance, when the skew is low. Interestingly, uniform hash starts performing worse compared to our algorithms, either before (in Fig. 4a) for LCL resource combination) or at the point (in Fig. 4b) where the relative imbalance goes above the ideal line.

Among the different algorithms we provided, the readj algorithm performs best for LCL and LLL resource combinations (up to 8 % lower, for instance compared to redist and scan for the LLL case with skew $z = 1$). For the LQL resource combination, all approaches are close, readj having slightly higher imbalance (around 1–2 %). The imbalance values for scan and redist are almost identical.

Impact of frequency threshold scaler. Recall that we employ a frequency threshold scaler, $\sigma \in [0, 1]$, which is used to set δ as shown in Eq. 17. We use a default value of 0.1 for this parameter. Figure 5 plots relative migration cost (on the left) and the relative load imbalance (on the right), as a function of σ. The results are shown for the resource combinations LCL and LQL (LLL results were similar to LCL results).

We observe that lower σ values bring lower imbalance, but higher migration cost. This is expected, as a lower σ value results in more mappings to be kept in the explicit hash, providing additional flexibility for achieving good balance. As discussed before, improved balance comes at the cost of increased migration cost.

In terms of migration cost, the redist algorithm provides the best results and the scan algorithm the worse results, considering only our algorithms. As with other results, consistent hash has the best migration cost and uniform hash the worst.

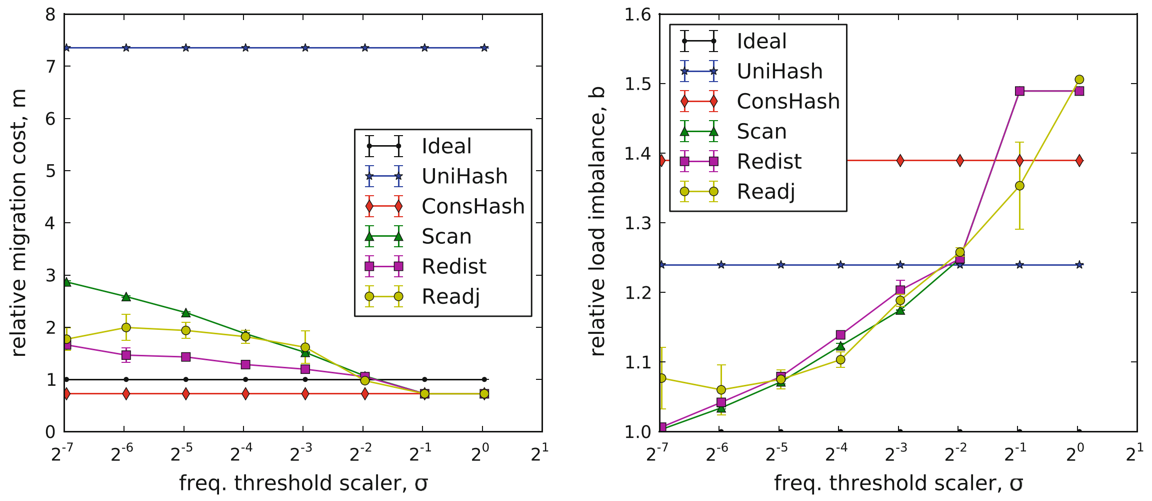
In terms of the load balance, our three algorithms provide similar performance. In the midrange of the frequency threshold for the LCL resource combination, readj algorithm shows slightly lower imbalance. However, for very low values of σ, the readj algorithm is unable to continue keeping the load imbalance lower. For the LQL resource combination, the different heuristic approaches perform closely. Interestingly, the improvement provided by lower σ values in terms of load balance is not as pronounced compared to the LCL case. Also, there is a significant jump in the imbalance around the range [0.25, 0.5].

The default setting of $\sigma = 0.1$ strikes a good balance between keeping the migration cost low and the load relatively well balanced. Even though smaller values seem to provide really good balance, this not only comes at high migration cost, but also—as we will see later see in this section—at a very high cost with respect to partitioning function construction time as well.

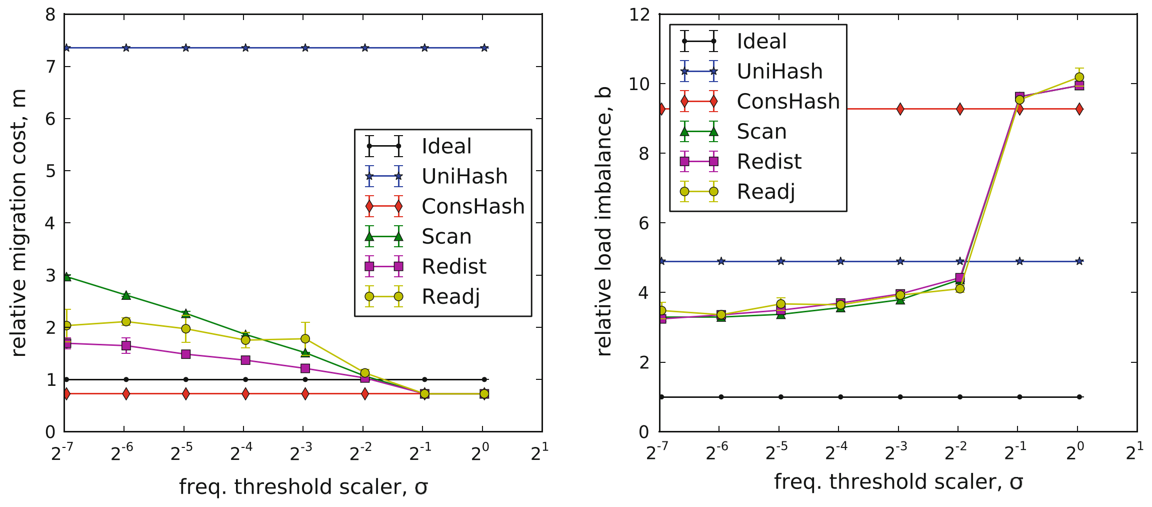
Impact of the number of parallel channels. Figure 6 plots the relative migration cost and the relative load imbalance as a function of the number of parallel channels (N) for different algorithms. As usual, uniform hashing has very high migration cost. It reaches around 22 times the ideal migration cost for 32 channels. Consistent hashing has the lowest migration cost, and our algorithms are in between. As the number of channels increase, the redist algorithm shows almost flat relative migration cost around 1.15 times the ideal. Both the scan and readj algorithms have increasing migration cost with increasing number of channels, the former having slightly higher cost. For 32 channels, the relative migration cost reaches above 3 for the scan algorithm.

Looking at load balance, again we see that consistent hash has the highest imbalance, which increases with increasing number of channels, reaching above 2 for 32 channels. All other approaches have lower imbalance. When the number of parallel channels is in the range $[2 - 20]$, uniform hashing has a clearly higher relative imbalance—up to 36 % higher compared to readj. In this range, readj algorithm performs the best. However, after 20 channels, the imbalance of readj goes above those of redist and scan. Considering both migration and the imbalance, the redist algorithm is the most robust one.

Impact of utility function. Figure 7a plots the relative migration cost (left chart) and relative imbalance (right chart) for the readj algorithm, using different utility functions. Looking at the migration cost, it is easy to see that U^A performs poorly with respect to relative migration cost, as it ignores the migration penalty. In general, U^{APM} provides the lowest relative migration cost, with the exception of LCL case, where U^{APLM} performs better. Looking at the imbalance values, we see almost no difference between different utility functions, except for the case of LQL. In the latter case, U^{APM} provides the lowest imbalance.



(a) For resource functions LCL



(b) For resource functions LQL

Fig. 5 Impact of frequency threshold scaler on migration and balance **a** For resource functions LCL **b** For resource functions LQL

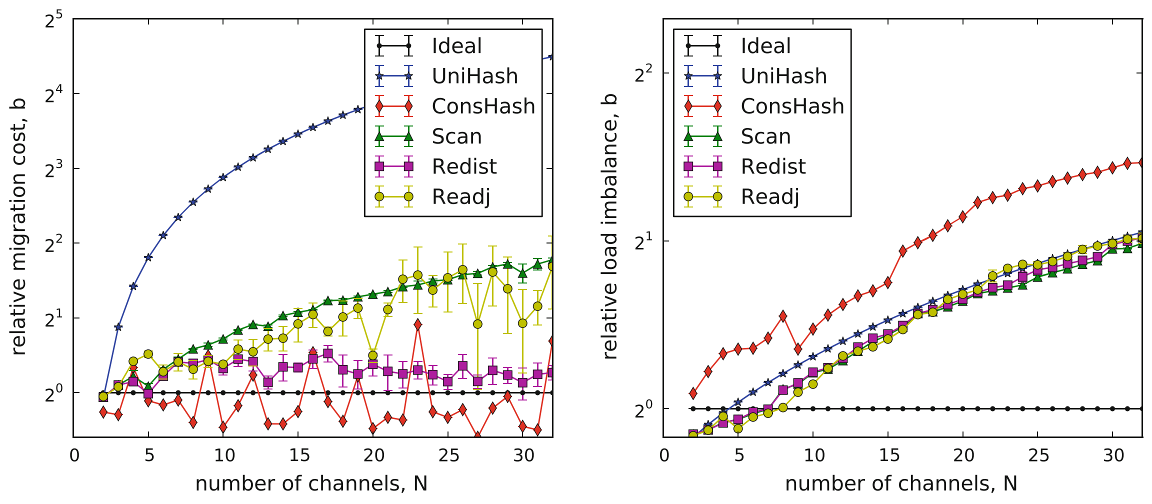
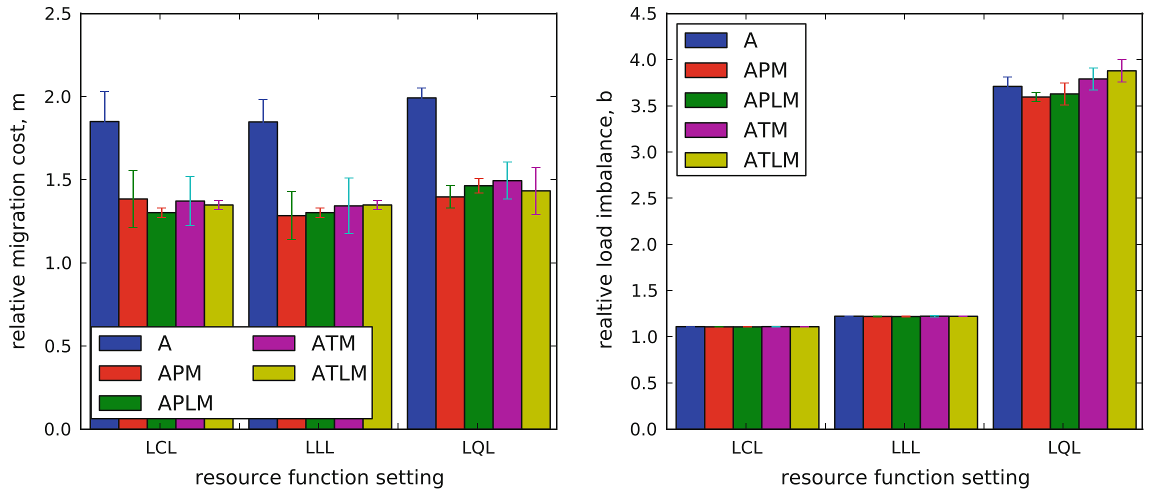
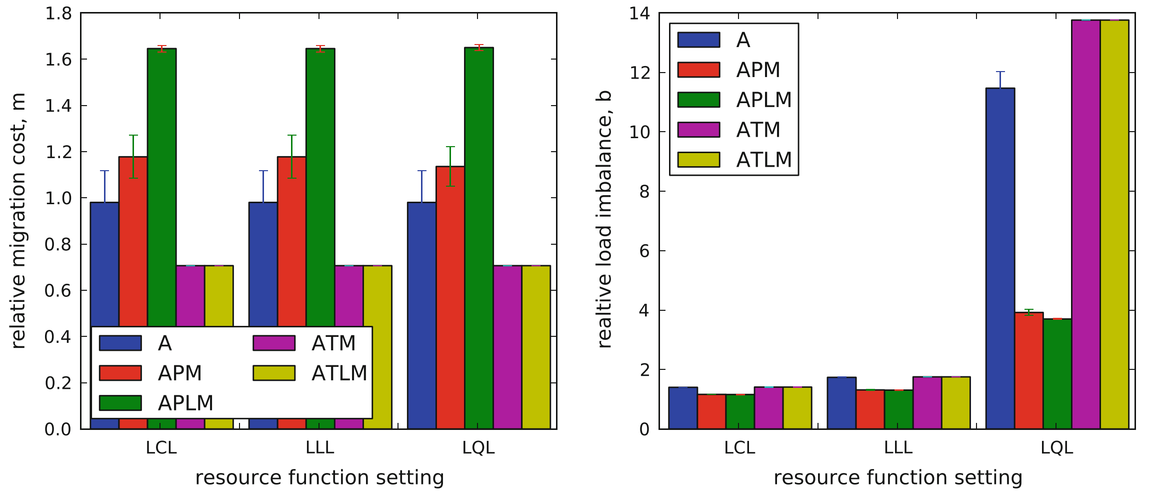


Fig. 6 Impact of number of channels on migration and load imbalance



(a) For readj algorithm



(b) For redist algorithm

Fig. 7 Impact of utility functions on readj algorithm **a** For readj algorithm **b** For redist algorithm

The results for the redist algorithm are shown in Fig. 7b. In terms of load balance, U^{APM} and U^{APLM} are performing the best. For the LQL resource combination, the improvement in relative imbalance is significant: up to 3 times lower. In terms of migration cost, the default utility function (U^{APM}) provides mediocre performance: The worst performing alternative (U^{APLM}) has 30–35 % higher migration cost, and the best performing ones (U^{ATM} and U^{ATM}) have 35–40 % lower migration cost. Considering both migration cost and relative imbalance, U^{APM} is the best choice. This is why we pick it as the default utility function.

Impact of domain size. Figure 8 plots the relative load imbalance as well as migration cost as a function of the domain size, for different algorithms. With respect to load imbalance, the relative performance of different algorithms is in line with our observations so far. Our algorithms perform better than both consistent hash and uniform hash, the former having the highest imbalance. Our three approaches have similar performance, with redist providing up to 3 % higher imbalance compared to readj, scan being almost same as the former. As the domain size increases, given the fixed Zipf skew, it becomes easier to balance the load. However, the relative imbalance shows a flattening trend as the domain size further increases. None of the approaches are able to reach the ideal balance for the default skew of $z = 1$.

When we look at the relative migration cost, we see that uniform hash has unacceptably high migration cost ($6.5\times-7.5\times$ of the ideal), which gets worse with higher domain sizes. Consistent hash, on the other hand, performs the best. Its relative migration cost is below the ideal. This is due

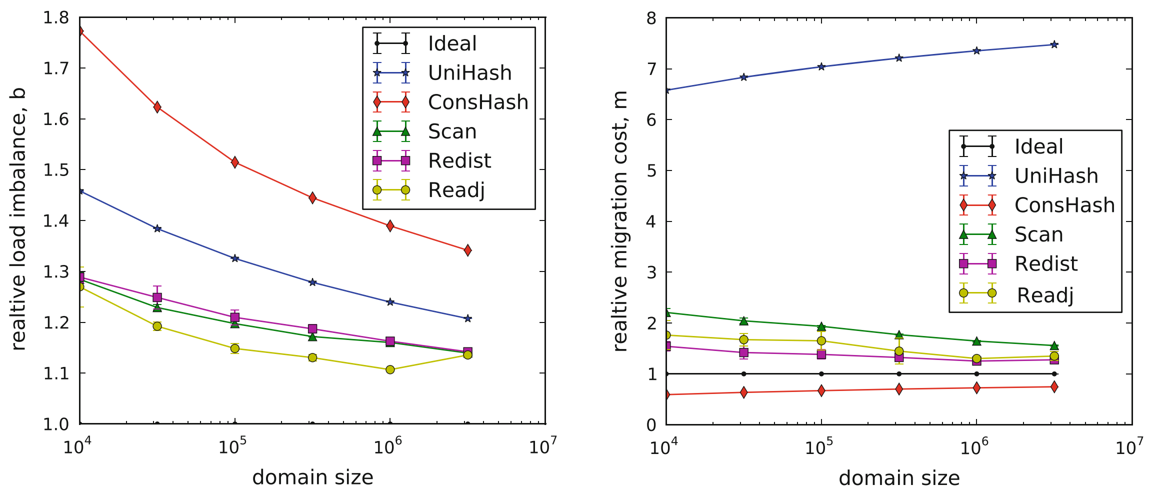


Fig. 8 Impact of domain size on load imbalance and migration

to the skew in the dataset. As the consistent hash tends to not migrate items, not moving items of high frequency can result in relative migration costs below a single channel's worth of migration. Our algorithms achieve a migration cost between $2.2\times-1.25\times$ of the ideal, with reducing costs as the domain size increases. Among our algorithms, redist is the most effective and scan is the least effective.

5.4 Partitioning function size

We evaluate the impact of algorithm and workload parameters on the size of the partitioning function. In particular, we look at the number of items kept in the lossy counter and the number of mappings kept in the explicit hash. Recall that, the lossy counter keeps two counters per item, and the explicit hash keeps a single channel index per item.

Impact of frequency threshold scaler. Figure 9 plots the number of items in the lossy counter (using left y-axis and the solid lines) and the number of items in the explicit hash (using right y-axis and dashed lines) as a function of the frequency threshold scaler, σ, for different resource combinations.

The lossy counter size increases as the frequency threshold becomes smaller. For the LCL and the LQL resource combinations, the lines overlap as the highest order function determines the δ and thus the number of items kept in the lossy counter. For the default setting of $\sigma = 0.1$, the number of items kept is around 2500—quite low compared to the 10^6, which is the total. For the case of LQL, this number reaches 50K, still acceptable as an absolute value, but only 1/20th of the total. This is not too surprising, as the domain size shows up as an inverse term in Eq. 17. As a result, the worst case assumption used to compute δ results in a very low value. This could be improved if an estimate of the data

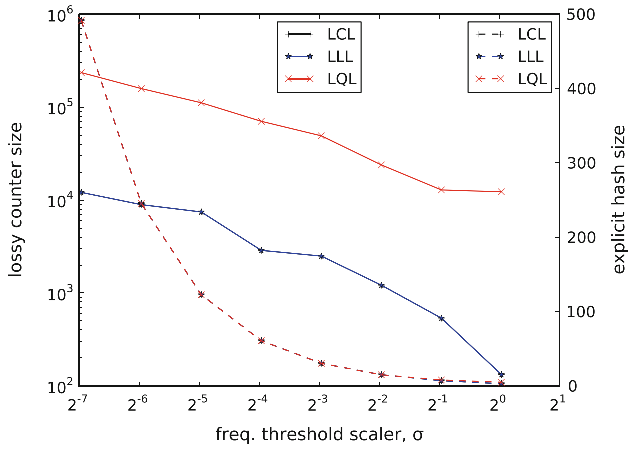


Fig. 9 Impact of freq. threshold scaler on the part. function size

item distribution is known or sampling could be used to get an estimate of it.

The size of the explicit hash is much lower, ranging between 1 and 500. For the default setting of $\sigma = 0.1$, it is around 50. Surprisingly, for all resource functions, the number of items kept in the explicit hash is the same (all three lines overlap). This is because for the quadratic resources, we use the items collected in the lossy counter to readjust our estimate of δ and perform an additional filter step. This was described at the end of Sect. 4.1.3.

Figure 10 plots number of items in the lossy counter (using left y-axis and the solid lines) and the number of items in the explicit hash (using right y-axis and dashed lines) as a function of the data skew, z, for different resource combinations. Since the imbalances reach unacceptable levels with the LQL setting under skew higher than $z = 0.5$, we plot the results for the LQL resource combination for a lower

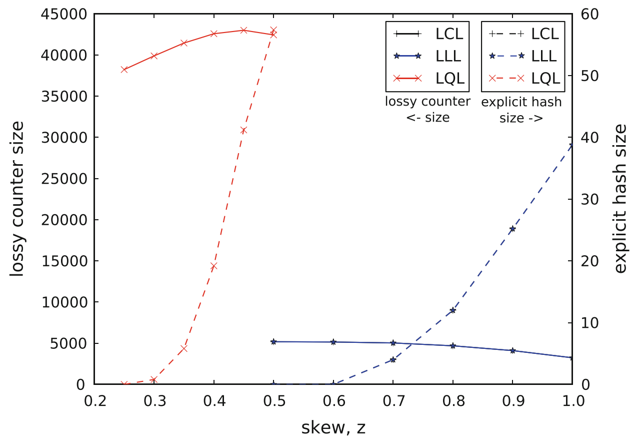


Fig. 10 Impact of skew on partitioning function size

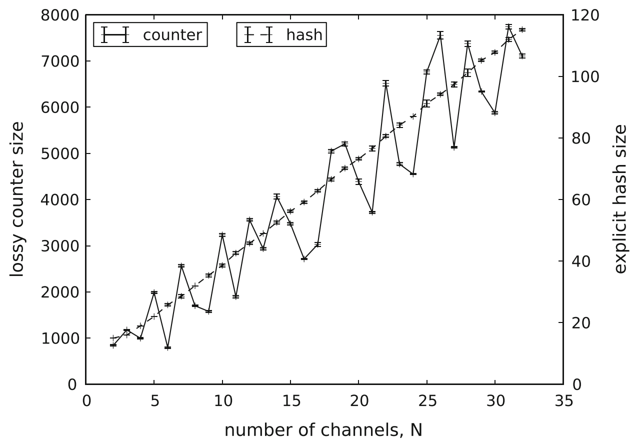


Fig. 11 Impact of # of channels on partitioning function size

range of the skew. The lines for LCL and LLL completely overlap.

The number of items kept in the lossy counter is not significantly impacted by the skew. For the case of non-quadratic resource functions, it stays mostly flat and slightly reduces for very high skew. For the quadratic case, it shows an initial increase, followed by a slight decrease. Interestingly, the number of items kept in the explicit map grows with an increasing rate as the skew increases. This is shown by the dashed lines having a super-linear trend.

Figure 11 plots the number of items in the lossy counter (using left y-axis and the solid lines) and the number of items in the explicit hash (using right y-axis and dashed lines) as a function of the number of channels. Recall that according to Eq. 17, the larger the number of channels N, the smaller the δ, and thus the higher the number of items tracked. We observe that the size increases linearly with the number of channels.

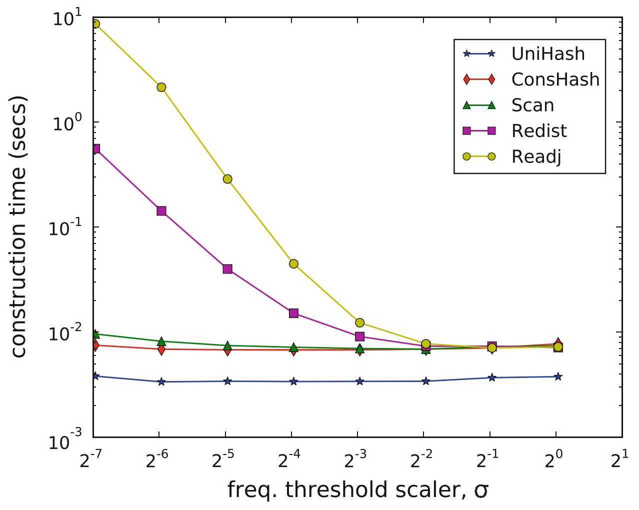


Fig. 12 Impact of frequency threshold scaler on partitioning function construction time

5.5 Partitioning function construction time

We evaluate the impact of algorithm and workload parameters on the time it takes to construct the partitioning function.

Figure 12 plots the partitioning function construction time (in seconds), as a function of the frequency threshold scaler. Recall from Fig. 5 that reduced relative imbalance is possible with values smaller than the default setting of $\sigma = 0.1$, albeit at the cost of increased migration cost. Figure 12 shows that the partitioning function construction time also increases with lower values of σ. For instance, the reconstruction cost for readj algorithm reaches around 10 s for $\sigma = 0.008$, whereas it is below 0.05 s for the default setting. Recall that, readj algorithm's computational complexity is cubic in the number of items, whereas for redist, it is quadratic. For $\sigma = 0.008$, the reconstruction time for the redist algorithm is slightly above 0.5 s, still acceptable considering adaptation pauses in the order of seconds [9]. The scan algorithm has good construction time performance as expected. In general, if higher migration costs are acceptable, the scan algorithm can be a good choice with low σ settings.

Figure 13 plots the partitioning function construction time (in seconds), as a function of the number of parallel channels. Recall that, we provided time complexities in Sect. 4.2, using R and N, where the former is the number of items in the explicit hash. As we have seen in Fig. 11, R scales linearly with the number of channels N. Thus, all our algorithms are superlinear in the number of channels. Scan is the cheapest algorithm with complexity $\mathcal{O}(R \cdot N \cdot \log N)$. For the consistent hash, we use 1000 replicas and 100 buckets. For these

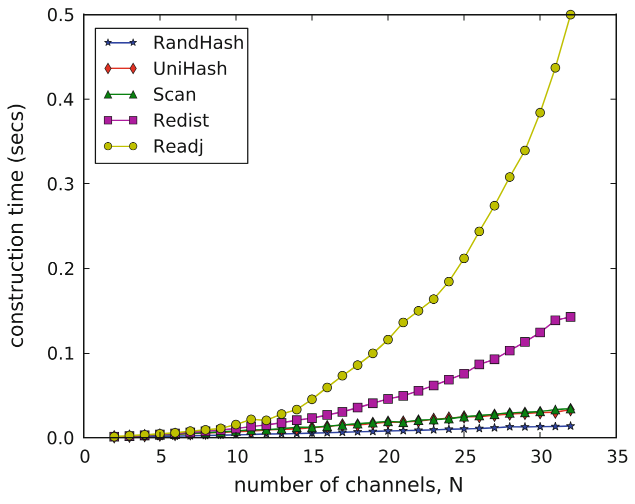


Fig. 13 Impact of number of channels on partitioning function construction time

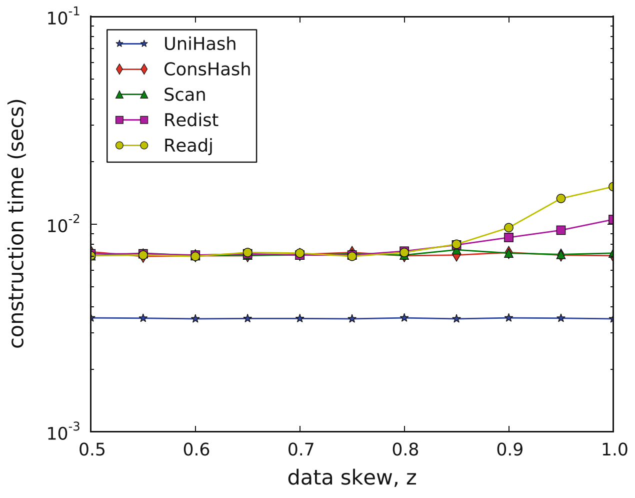


Fig. 14 Impact of data skew on partitioning function construction time

settings, the cost of constructing the consistent hash and the explicit hash is about the same for the scan algorithm. For other algorithms, the construction time for the explicit hash is significantly higher and the rate of increase for the overall construction time is higher.

Figure 14 plots the partitioning function construction time (in seconds), as a function of the data skew z, for different algorithms. In summary, the construction time is mostly insensitive to the data skew for the scan algorithm. For the redist and readj algorithms, the construction time stays flat until the data skew goes beyond 0.8, after which the construction time increases. The rate of increase is faster for readj compared to the redist algorithm.

6 Related work

Impact of data skew on query processing performance has been studied extensively in the context of parallel data base systems [7,8,21,26,29,30]. Most of this work has focused on parallel join processing.

A taxonomy of skew effects in join processing is given in [29]. The skew found in the attribute values of the source data is named as *intrinsic skew*. This is the same kind of skew we are addressing in this paper. The skew that results from the work not being balanced among the nodes that are participating in the parallel computation is named *partition skew*. This is what we call the imbalance problem. In this work, we consider computation, communication, as well as memory imbalance, with different resource functions (constant, linear, and quadratic). Since our work is on stream processing, there is no I/O involved.

In [7], multiple algorithms, each specialized for a different degree of skew, are used to handle skew in join processing. To decide on the specific algorithm to apply, data sampling is used. Since in our context the data is streaming, we rely on sketches to detect the data characteristics (which may change over time as well). Other examples of work addressing join processing under skewed workloads include handling skew found in join query results [21] and handling skew in outer joins [30].

Parallel aggregate processing over skewed data is another relevant area, perhaps more closely related to our work, since an aggregation operator can be considered as a partitioned parallel region with a single operator in it. However, the traditional parallel aggregation computation problem does not consider streaming data. There are two fundamental approaches to parallelizing aggregate operators [26]. The first is to compute the aggregation on different parts of the data and then to merge the results. The second is to perform the aggregation independently on different partitions, where each partition is assigned to one of the nodes. Our work resembles this latter approach. The first approach requires commutative and associative functions, and also is difficult to apply in a streaming setting as the operators are not allowed to block. In [26], a hybrid scheme that relies on variations of the two fundamental approaches to parallel aggregation computation is described, which can also adapt to the data characteristics, such as skew, by changing the algorithm being used at runtime.

Streaming aggregation computation using data parallelism has been studied in the literature as well [4]. For streaming aggregations, data partitioning is performed by taking into account the window boundaries. The basic idea is to distribute windows of data over nodes, but when the successive windows are highly overlapping (e.g., for sliding windows), this approach does not scale. Additional techniques are developed, which divide the windows into panes and distribute the pains across nodes, in order to minimize the amount of repeated work and improve scalability. Our work is orthogonal to this, as we focus on partitioned stateful data parallelism. Our partitioning functions do not work for operators that are not partitioned on a key. Yet, when one or more operators are partitioned on a key, our approach can be applied irrespective of the kinds of the operators being used.

Map/Reduce systems is another area where the existence of data skew, and its impact on query performance has been noted [8]. A solution to this problem addressing skew that arises due to uneven assignment of data to processing nodes as well as due to varying processing costs of different data items is given in [17]. The idea is to detect skew, stop the straggling tasks, and to apply repartitioning. A related technique that can be used to handle skew in Map/Reduce systems is scalable cardinality estimation [11,12].

Another relevant area is adaptive query processing (AQP) [6], in particular the Flux operator [25]. This operator applies partitioned parallel processing in the context of stateful continuous queries. The focus is on dynamic load balancing, but the level of parallelism is not dynamically adjusted. Comparison of several different approaches for query parallelization under this assumption can be found in the literature [20].

None of the previous approaches consider skew in the context of stateful stream processing operators. Furthermore, adaptation that involves adjusting the number of parallel channels at runtime is not considered in these works. As a direct consequence of the latter, none of the previous works consider migration cost in their load balancing approach. Our recent work on auto-parallelizing stateful operators [24] gives a detailed overview of partitioned parallel processing in stream processing systems.

7 Conclusion

In this paper, we studied partitioning functions that can be used to distribute load among parallel channels in a data parallel region within a stream processing application. The functions provide good computation, communication, and memory load balance, while at the same time keeping the overhead of migration low, all in the presence of data skew. The migration is a critical aspect for stateful parallel regions that support elastic scalability—changing the number of parallel channels at runtime based on the resource and workload availability. The partitioning function structure we proposed is compact and provides constant time lookup. We introduced several algorithms that rely on a greedy procedure based on a utility function to quickly construct partitioning functions. Our evaluation shows that the proposed functions provide better load balance compared with uniform and consistent hashing and migration cost close to that of consistent hashing.

Acknowledgments We thank IBM Thomas J. Watson Research Center for providing access to compute and software resources that made this research possible.

References

1. Abadi, D., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The design of the Borealis stream processing engine. In: Proceedings of the Innovative Data Systems Research Conference (CIDR), pp. 277–289 (2005)
2. Arasu, A., Manku, G.S.: Approximate counts and quantiles over sliding windows. In: Proceedings of the Symposium on Principles of Database Systems (ACM PODS) (2004)
3. Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Motwani, R., Nishizawa, I., Srivastava, U., Thomas, D., Varma, R., Widom, J.: STREAM: the stanford stream data manager. IEEE Data Eng. Bull. **26**(1), 665 (2003)
4. Balkesen, C., Tatbul, N.: Scalable data partitioning techniques for parallel sliding window processing over data streams. In: International Workshop on Data Management for Sensor Networks (DMSN) (2011)
5. Cormode, G., Garofalakis, M., Haas, P., Jermaine, C.: Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. Now Publishing, Foundations and Trends in Databases Series (2011)
6. Deshpande, A., Ives, Z.G., Raman, V.: Adaptive query processing. Found. Trends Databases **1**(1) (2007)
7. DeWitt, D., Naughton, J., Schneider, D., Seshadri, S.S.: Practical skew handling in parallel joins. In: Proceedings of the Very Large Data Bases Conference (VLDB) (1992)
8. Gates, A.F., Natkovich, O., Chopra, S., Kamath, P., Narayanamurthy, S.M., Olston, C., Reed, B., Srinivasan, S., Srivastava, U.: Building a high-level data flow system on top of map-reduce: The PIG experience. In: Proceedings of the Very Large Data Bases Conference (VLDB) (2009)
9. Gedik, B., Schneider, S., Hirzel, M., Wu, K.L.: Elastic scaling for data stream processing. IBM Research Technical Report, RC25401 (2013)
10. Gedik, B., Andrade, H.: A model-based framework for building extensible, high performance stream processing middleware and programming language for IBM InfoSphere streams. Softw. Pract. Exp. **42**(11), 1363–1391 (2012)
11. Gufler, B., Augsten, N., Reiser, A., Kemper, A.: Handling data skew in mapreduce. In: Proceedings of the International Conference of Cloud Computing and Services Science (2011)
12. Gufler, B., Augsten, N., Reiser, A., Kemper, A.: Load balancing in mapreduce based on scalable cardinality estimates. In: Proceedings of the International Conference on Data Engineering (IEEE ICDE) (2012)
13. Hirzel, M., Andrade, H., Gedik, B., Kumar, V., Losa, G., Mendell, M., Nasgaard, H., Soulé, R., Wu, K.L.: SPL language spec. Tech. Rep. RC24897, IBM (2009)
14. Jain, N., Amini, L., Andrade, H., King, R., Park, Y., Selo, P., Venkatramani, C.: Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In: Proceedings of the International Conference on Management of Data (ACM SIGMOD) (2006)
15. Karger, D.R., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In: Proceedings of the International Symposium on Theory of Computing (ACM STOC), pp. 654–663 (1997)
16. Karger, D.R., Sherman, A., Berkheimer, A., Bogstad, B., Dhanidina, R., Iwamoto, K., Kim, B., Matkins, L., Yerushalmi, Y.: Web

caching with consistent hashing. Comput. Netw. **31**(11–16), 1203–1213 (1999)
17. Kwon, Y., Balazinska, M., Howe, B., Rolia, J.A.: SkewTune: mitigating skew in mapreduce applications. In: Proceedings of the International Conference on Management of Data (ACM SIGMOD) (2012)
18. Manku, G.S., Motwani, R.: Approximate frequency counts over data streams. In: Proceedings of the International Conference on Very Large Databases (VLDB) (2002)
19. MurMurHash3. http://code.google.com/p/smhasher/wiki/MurmurHash3 (2013). Retrieved May 2013
20. Paton, N.W., Chavez, J.B., Chen, M., Raman, V., Swart, G., Narang, I., Yellin, D.M., Fernandes, A.A.A.: Autonomic query parallelization using non-dedicated computers: An evaluation of adaptivity options. In: Proceedings of the Very Large Data Bases Conference (VLDB) (2009)
21. Poosala, V., Ioannidis, Y.E.: Estimation of query-result distribution and its application in parallel-join load balancing. In: Proceedings of the Very Large Data Bases Conference (VLDB) (1996)
22. S4 distributed stream computing platform. http://www.s4.io/ (2012). Retrieved May 2012
23. Schneider, S., Andrade, H., Gedik, B., Biem, A., Wu, K.L.: Elastic scaling of data parallel operators in stream processing. In: Proceedings of the International Parallel and Distributed Processing Symposium (IEEE IPDPS) (2009)
24. Schneider, S., Hirzel, M., Gedik, B., Wu, K.L.: Auto-parallelizing stateful distributed streaming application. In: Proceedigns of the International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 53–64 (2012)
25. Shah, M.A., Hellerstein, J.M., Chandrasekaran, S., Franklin, M.J.: Flux: An adaptive partitioning operator for continuous query systems. In: Proceedings of the International Conference on Data Engineering (IEEE ICDE) (2003)
26. Shatdal, A., Naughton, J.: Adaptive parallel aggregation algorithms. In: Proceedings of the International Conference on Management of Data (ACM SIGMOD) (1995)
27. Storm project. http://storm-project.net/ (2012). Retrieved May 2012
28. StreamBase Systems. http://www.streambase.com (2012). Retrieved May 2012
29. Walton, C., Dale, A., Jenevein, R.: A taxonomy and performance model of data skew effects in parallel joins. In: Proceedings of the Very Large Data Bases Conference (VLDB) (1991)
30. Xu, Y., Kostamaa, P.: Efficient outer join data skew handling in parallel dbms. In: Proceedings of the Very Large Data Bases Conference (VLDB) (2009)