SquirrelJoin: Network-Aware Distributed Join Processing with Lazy Partitioning

Lukas Rupprecht
Imperial College London
lr12@imperial.ac.uk

William Culhane
Imperial College London
w.culhane@imperial.ac.uk

Peter Pietzuch
Imperial College London
prp@imperial.ac.uk

ABSTRACT

To execute distributed joins in parallel on compute clusters, systems partition and exchange data records between workers. With large datasets, workers spend a considerable amount of time transferring data over the network. When compute clusters are shared among multiple applications, workers must compete for network bandwidth with other applications. These variances in the available network bandwidth lead to *network skew*, which causes straggling workers to prolong the join completion time.

We describe *SquirrelJoin*, a distributed join processing technique that uses *lazy partitioning* to adapt to transient network skew in clusters. Workers maintain in-memory *lazy partitions* to withhold a subset of records, i.e. not sending them immediately to other workers for processing. Lazy partitions are then assigned dynamically to other workers based on network conditions: each worker takes periodic throughput measurements to estimate its completion time, and lazy partitions are allocated as to minimise the join completion time. We implement SquirrelJoin as part of the Apache Flink distributed dataflow framework and show that, under transient network contention in a shared compute cluster, SquirrelJoin speeds up join completion times by up to 2.9× with only a small, fixed overhead.

1 Introduction

With the recent explosion of big data analytics, users frequently want to execute parallel joins over large datasets to combine and analyse data from different sources. For this, organisations deploy massively-parallel, shared-nothing distributed dataflow systems such as Hadoop [39], Spark [4] or Flink [18], which perform join computation on large compute clusters.

As the cost of acquiring and operating large compute clusters becomes substantial, organisations increasingly share clusters between different applications. Cluster managers such as Mesos [23] or YARN [42] successfully isolate CPU and memory resources for each application through virtual machines (VMs) or, more recently, containers. Network bandwidth, however, remains a contended resource. In particular, the available edge bandwidth of cluster nodes, typically limited by a 1 Gbps or 10 Gbps NIC, can become a bottleneck, affecting the performance of data processing jobs [34, 45]. The uneven distribution of network flows, and thus available bandwidth, across nodes creates what we refer to as *network skew*.

The performance of distributed join processing is particularly sensitive to network skew. In a distributed *repartition join* [9], workers read data records from input partitions in parallel and repartition them based on a join key, i.e. transfer them to other workers that then apply the join predicate. Redistributing a large number of data records is limited by the available network bandwidth between nodes: in practice, it accounts for up to 43% of the join completion time [30]. In a shared network, this traffic competes with other network flows for edge bandwidth. Network skew thus introduces *straggling* workers, which increases join completion time due to the late-arrival of results or head-of-line blocking [36]. The join computation only completes after the last worker has finished processing.

Previous work has focused on *data skew* [44], i.e. the imbalance of data processed by nodes due to an uneven key distribution in the join. Prior solutions for data skew cannot help with network skew: they either rely on prior knowledge of the skew [46, 33], which is infeasible when organic background traffic causes network skew, or they migrate state between workers across the network to mitigate data skew [16, 36], which would only exacerbate network skew when the network is the bottleneck.

Addressing network skew therefore requires a new approach. We describe a runtime mitigation technique for network skew based on **lazy partitioning**. The key idea behind lazy partitioning is that, instead of using a single hash-based mapping of join keys to workers, workers retain some of the data records in *lazy partitions* maintained in memory. Lazy partitions are then flexibly assigned at runtime to workers in reaction to network skew conditions.

When realising this idea, we overcome three challenges: how to (i) measure network skew in the cluster; (ii) determine an optimal allocation of lazy partitions to workers in response to network skew; and (iii) minimise the overhead of maintaining lazy partitions when there is no network skew in the cluster.

We describe **SquirrelJoin**, a new distributed join processing technique to mitigate network skew by retaining records at workers.[1] SquirrelJoin is lightweight and compatible with existing distributed join algorithm implementations in dataflow systems. In SquirrelJoin, a coordinator continuously measures the throughput of workers participating in the join processing to produce estimated completion times. Network skew is detected if the estimated completion time of a worker is statistically significantly higher than that of the rest.

While network skew can affect either receiving or sending workers, SquirrelJoin only focuses on *receiver-side* skew because our theoretical analysis shows that any mitigation of sender-side skew can offer only a limited benefit in practice. To mitigate receiver-side

[1] similar to how squirrels stash away food before winter

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 10, No. 11
Copyright 2017 VLDB Endowment 2150-8097/17/07.

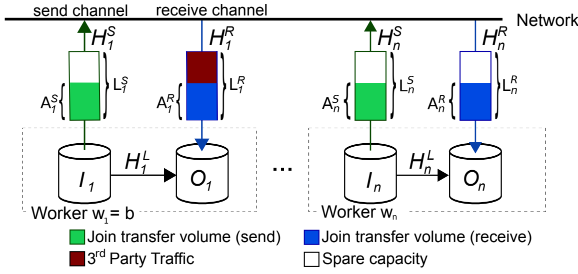

Figure 1: Distributed join processing model

skew, SquirrelJoin assigns proportionally more data from lazy partitions to faster receivers. As a result, workers affected by receiver-side skew have to process less data, thus equalising data processing across the cluster with network skew.

We demonstrate the feasibility of SquirrelJoin through an implementation as part of *Apache Flink* [18], a popular distributed dataflow system for join processing. In our implementation, all skew detection and assignment decisions occur asynchronously and do not require further synchronisation between senders, reducing SquirrelJoin's overhead to a minimum.

After describing distributed join processing and the problem of network skew in clusters (§2), we make the following contributions:

- we formally analyse the effectiveness of *record reassignment* to mitigate network skew at receivers and senders, respectively. Our analysis informs our use of a simple, yet effective, way to balance receiver-side network skew in SquirrelJoin (§3);
- we describe *lazy partitioning* and an algorithm for partition assignment to dynamically adapt receiver partitions to network skew (§4);
- we describe *SquirrelJoin*, a distributed join processing technique that mitigates network skew. SquirrelJoin collects robust progress metrics to make decisions about the assignment of lazy partitions (§5).

We evaluate SquirrelJoin using Apache Flink and a variety of network skew scenarios on a shared cluster, using interfering real-world applications and synthetic micro-benchmarks (§6). Our results show that SquirrelJoin achieves speed-ups of up to 2.9×, while adding a fixed 10 s increase in job completion time without network skew. After a discussion of related work (§7), the paper concludes (§8).

2 Network Skew in Join Processing

We start by modelling distributed join processing in a compute cluster (§2.1). We detail the issue of network skew in shared clusters (§2.2) and why existing approaches fail to address it (§2.3).

2.1 Distributed join processing

Join model. We model an *equi-join*, $S \bowtie_J T$, over relational tables, S and T, with records, $r \in \mathbb{R}$, each with a join attribute J. We use $J(r)$ to refer to the value of the join attribute for record r.

When faced with large amounts of data, join processing can be distributed to exploit data parallelism on a cluster of n nodes. We assume that the records from S and T are partitioned horizontally into a set of *input partitions*, $I = \{I_1, I_2, \dots, I_n\}$. The set of records across all partitions is equal to the set of records across S and T. Each node i stores an input partition $I_i = \{r_a, r_b, \dots\}$ locally. To balance load, input partitions are typically equally sized [46].

We consider an implementation of the equi-join as a hash-based *repartition join* [9], in which the n input partitions $\{I_1, \dots, I_n\}$ with records from S and T are *repartitioned* according to the join attribute J into n *output partitions*, $\{O_1, \dots, O_n\}$. Each output partition contains all records from both tables with the same value for the join attribute J, so the join predicate can be evaluated local to an output partition. Each node i maintains one output partition O_i.

As shown in Fig. 1, a node i executes a *join worker* $w_i \in \mathbb{W}$ with a *sender* and a *receiver* process: the sender process reads each record from the local partition I_i, determines the output partition O_j, and sends the record to the receiver process of the responsible worker w_j. The receiver process of w_j applies the join predicate to O_j, which contains all records from S and T with the same value for the join attribute J, to generate the join result.

In *symmetric* hash joins, such as XJoin [41], sender processes repartition records from S and T simultaneously, which requires receiver processes maintain all received records until the join completes. To reduce memory usage, *two-phase* repartition joins, such as the GRACE hash join [25] or the hybrid hash join [15], repartition S and T *sequentially*. Assuming that S is the smaller table, (i) in a *build* phase, sender processes first repartition records from S, and the receiver processes store the received records; and (ii) in a *probe* phase, sender processes repartition the records from T, and receiver processes probe their stored records from S for matches.

Network model. We model the network as a set of *channels* $H \in \mathbb{H}$, between nodes, each with the same maximum *capacity* C. In a network with full-bisection bandwidth, the capacity C is determined by the network interface card (NIC), e.g. 1 Gbps. We denote the overall join completion time as Δ. The *maximum channel volume* L is then the amount of data that can be transferred over a channel during Δ, $L = \Delta C$. For example, the maximum channel volume for a 1 Gbps channel with a join that runs for 100 seconds is 100 Gb.

In a full-duplex network, nodes have independent *send* and *receive* channels, H^S and H^R, respectively (see Fig. 1). The sender process of worker w_i can use channel H_i^S to send records to other workers; its receiver process can use H_i^R to receive records. Note that the send and receive channels of a node are shared by the join worker's sender and receiver processes with other applications executing on the same node. For convenience, we assume that node i also has a local channel H_i^L that its sender process can use to send records to its receiver process. The local channel does not incur network traffic by using inter-process communication (IPC).

For a channel H_i, we define the *join transfer volume* A_i^S and A_i^R as the actual amount of data that worker w_i has transferred on send and receive channels respectively. The *join channel utilisation* U_i is then the ratio between the join transfer volume and the maximum channel volume, e.g. $U_i^R = A_i^R / L$. In the ideal case of equal-sized input and output partitions and no CPU bottlenecks, all workers send and receive the same amount of data, so, without network skew, all channels have the same join channel utilisation,

$$\forall w_i, w_j \in \mathbb{W} \,:\, U_i^S = U_j^S = U_i^R = U_j^R = 1.$$

If $U_i < 1$, (i) the channel is under-utilised because the join computation on worker w_i is limited by another resource (e.g. CPU) or waits for records from another worker; or (ii) the channel is congested because some of its capacity is used by other application traffic on node i. In the latter case, the channel may be a bottleneck, making w_i a *straggling* worker. If different channels experience different degrees of congestion, we refer to this as *network skew*.

2.2 Network skew in shared clusters

For many organisations, compute clusters are a source of major capital and operational expenses. To amortise costs, organisations are moving away from dedicated clusters for specific data processing tasks towards consolidating multiple applications on the same cluster [23]. These cluster applications then share the network.

As a result, *network skew* due to background network traffic is increasingly common. Background traffic created by cluster applications may be concentrated on a subset of nodes, which reduces the available bandwidth to applications on some nodes, while leaving others unaffected [45, 27]. It also varies significantly over time in different dimensions, which means that static deployment changes or fixed bandwidth allocations are insufficient:

Duration. The distribution of flow sizes for cluster traffic covers a wide range from 10 KB to 1 GB [21, 8]. The majority of flows are small, but larger "elephant" flows account for most of the data. Microsoft reports that in one cluster 99% of flows were smaller than 100 MB, but 90% of the data was transferred in the remaining 1% of flows [21]. That means that most of the network resources are utilised by large flows, which can cause long interference periods.

Severity. Besides the flow size, the number of concurrent flows on a node also varies. This determines the *severity* of the interference, as more background flows mean less bandwidth available for join processing. In the same Microsoft cluster, machines have around 10 concurrent incoming and outgoing flows in the median. However, there is a long tail, and 5% of the time, machines experience more than 80 concurrent flows [21]; results from another cluster show a median of 36 concurrent flows with a 99.99th percentile of over 1600 [2]. These heavy tails can cause a significant decrease in the available bandwidth for join processing.

Variability. Cluster traffic is highly variable in terms of flow arrival times [21, 24, 8]. Greenberg et al. clustered the traffic matrices from a Microsoft data centre over a day but failed to find a representative subset to predict traffic accurately [21]. Even in periods of stable network traffic, the nodes involved in the network transfers experience high churn. Thus it is hard to predict both traffic volume and its location, making the avoidance of network skew a priori impossible.

The above characteristics of background traffic lead to frequent periods of network congestion, resulting in network skew. In one day, a 1500-machine cluster saw 665 episodes of congestion of between 10 s and 382 s each: 86% of links experienced congestion of more than 10 s, and 15% experienced congestion exceeding 100 s [24]. Hot spots were most severe in clusters with data analysis jobs: approximately 20% of the links in the network core experienced congestion for more than 50% of the time [8]. An important observation is that longer lasting congestion periods do not affect network links evenly but are localised to a small subset of links. This is the prime condition that leads to network skew.

2.3 Addressing other types of skew

Instead of network skew, existing solutions have focused on two other types of skew in distributed join processing:

Data skew is caused by an unequal key distribution in the input data. This may lead to an imbalanced partitioning of data: workers that must now perform more computation than others become stragglers.

Approaches to address data skew can be divided into static and dynamic techniques. *Static* techniques [33, 29, 32] attempt to apply more balanced partitioning functions that account for the skewed data distribution. For example, Rödiger et al. [32] sample a small subset of the data before the join processing to determine the heavy hitters and apply selective broadcast to distribute them across many nodes. Such an approach, however, is not applicable to network skew, which appears and also disappears dynamically at runtime.

Dynamic techniques react to data skew as it develops and then change the work assignment on-the-fly. Some existing proposals [26] assume that keys are ordered and new keys can be assigned to arbitrary workers, but this is not the case for most datasets. More general solutions instead migrate already-partitioned data between nodes at runtime [5, 16, 36]. Under network skew, a mitigation approach that migrates data between workers is not possible because the network itself is the cause of the skew and thus a constrained resource. Sending more data as part of the migration would further exacerbate the problem of network skew.

Resource skew is caused by the non-uniform allocation of node resources, such as CPU or disk, to workers. In practice, resource skew is often addressed by isolating resources: for example, cluster resource managers, such as Mesos [23] and YARN [42], can provide workers with guaranteed CPU resources through *containers*, thus reducing interference that would lead to resource skew.

While network skew is technically a type of resource skew, existing isolation techniques such as containers do not offer guarantees regarding available network bandwidth between nodes. In addition, network skew introduces *unidirectional* bottlenecks due to TCP congestion control and full-duplex network links—a congested network link may only affect the sender or receiver process of a worker, whereas CPU bottlenecks affect the whole node.

While a number of approaches exist to isolate network bandwidth [6, 28, 31, 11], they cannot fully solve network skew: (i) *static* approaches [6, 28] assign bandwidth to cluster users a priori, leading to under-utilisation if demands are set too high; and (ii) *work-conserving* approaches [31, 11] dynamically allocate bandwidth to achieve high utilisation while providing minimum guarantees. However, they either require setting a minimum bandwidth value [31], which is hard to determine correctly, or implement some form of max-min fairness, which is susceptible to network skew [11]. In addition, these advanced techniques are currently not available in public cloud environments. In general, network isolation approaches can enforce sharing constraints on congested links but are unable to avoid these links completely, leading to a sub-optimal network utilisation. Thus, mitigating network skew effectively requires explicit support as part of data processing applications.

3 Record Reassignment Under Network Skew

Our goal is to mitigate the effect of network skew on the completion time of join workers. The basic idea follows: in response to network skew, we *reassign* records among the output partitions, $\{O_1, \ldots, O_n\}$, maintained by different workers, and shift traffic from *congested* to *under-utilised* channels. We adjust the repartitioning performed by the join by *reassigning* all records that share the same value for the join attribute J, $\{r \mid J(r) = \alpha\}$, from an output partition O_i to a partition O_j before any records are sent via the network.

By modelling this record reassignment, we want to establish an upper bound on the improvement in job completion time that it can provide with network bottlenecks. We want to find the reassignment that achieves the maximum decrease in join completion time, which happens when the load on the most congested channel is decreased as much as possible without introducing a new straggling worker. We separately consider the scenarios with (i) congested receive channels and (ii) congested send channels.

For our theoretical analysis, we assume: (1) a uniform data distribution of the join attribute J results in equally-sized output partitions O across nodes, $\forall i, j \in \mathbb{W}, |I_i| = |I_j| = |O_i| = |O_j|$. While good hash functions typically achieve nearly balanced partition sizes [46, 33], we only make this assumption to calculate a ceiling of the potential benefit of reassignment. Data skew may change channel utilisation and the granularity at which we make reassignments (see §4.2); (2) a single network bottleneck dominates join completion time. Reassignment can only help mitigate skew as long as another contested resource, e.g. the CPU, does not become the dominating bottleneck; and (3) a priori knowledge of network skew.

(1) Congested receive channels. First we consider m straggling workers, $\mathbb{M} = \{w_{b_1}, w_{b_2}, \ldots, w_{b_m}\}$, with congested *receive* channels of the same magnitude, i.e. their network bandwidth is limited equally. The $n-m$ receiver processes on the other workers, and all n sender processes can utilise the full channel capacity. Since all straggling workers are symmetric, we use any $w_b \in \mathbb{M}$ to determine the *bottleneck join channel utilisation*, $U_b^R = A_b^R / L$.

We want to minimise the runtime of the straggling workers by reassigning records from straggling to non-straggling workers. The set of reassigned records is $R_{\mathbb{M}} = \bigcup_{i=1}^{m} O_{b_i} \setminus \widehat{O}_{b_i}$ where O_{b_i} are the records assigned to worker w_{b_i} before reassignment, and $\widehat{O}_{b_i}$ are the records after reassignment. Reassigned records are distributed evenly among non-straggling workers,

$$\forall r \in O_{\mathbb{M}}, \exists w_i \in \mathbb{W} \backslash \mathbb{M} \,:\, r \in \widehat{O}_i \text{ and } \forall w_i, w_j \in \mathbb{W} \backslash \mathbb{M} \,:\, |\widehat{O}_i| = |\widehat{O}_j|.$$

We refer to the amount of data at worker $w_b \in \mathbb{M}$ after reassignment as $\widehat{A}_b^R$. Now the fraction of records k that is reassigned from the straggling worker w_b is

$$k = (A_b^R - \widehat{A}_b^R) / A_b^R.$$

As a result of the reassignment, the join completion time Δ,

$$\Delta = \frac{\widehat{A}_b^R}{U_b^R C} = \frac{A_b^R (1-k)}{U_b C},$$

is decreased: while worker w_b remains a straggler due to its congested receive channel, U_b is constant, which means that $\widehat{A}_b^R$ is reduced by a factor of $(1-k)$.

Next we determine the maximum fraction of records k to reassign from the straggling workers before other channels become congested, i.e. reach a join channel utilisation of 1. For that, we analyse the change in the join transfer volume for each of the remaining non-congested channels:

Case (1): Receive channels on non-straggling workers. Worker w_i receives $(n-1)/n$ of its records via H_i^R, and the rest via H_i^L. A fraction of k records from each of the m straggling workers is reassigned evenly across all non-straggling workers, increasing the size of O_i at w_i by $km/(n-m)$. A fraction $(n-1)/n$ of the newly assigned records therefore increase the load on channel H_i^R:

$$\widehat{A}_i^R = A_i^R \left(1 + \frac{km}{(n-m)}\right) \tag{1}$$

Case (2): Send channels on straggling workers. After reassignment, each straggling worker w_b must send additional records via H_b^S. Originally, $(n-1)/n$ of the records in I_b were sent via H_b^S, while the remaining $1/n$ records used H_b^L. Now an additional fraction k of the records from H_b^L are also sent via H_b^S:

$$\widehat{A}_b^S = A_b^S \left(1 + \frac{k}{(n-1)}\right) \tag{2}$$

Case (3): Send channels on non-straggling workers. Finally, the reassignment shifts traffic on a non-straggling worker w_i from H^S to H^L, thus decreasing A_i^S. Of the fraction of records in I_i originally sent via H_i^S, $(n-1)/n$, only $km/((n-m)(n-1))$ are reassigned to O_i and now use H_i^L. The remaining reassigned records still use H_i^S despite having new output partitions, yielding:

$$\widehat{A}_i^S = A_i^S \left(1 - \frac{km}{(n-m)(n-1)}\right) \tag{3}$$

Considering the join transfer volumes after reassignment for these three cases, the receive channels on non-straggling workers, given by Eq. (1), experience the highest relative increase. To find the value of k that maximises the reduction in join completion time, we observe that a new bottleneck forms when U_i^R=1 for $w_i \notin \mathbb{M}$. Since $U_i^R = A_i^R L$ by definition, after substituting 1 for U_i^R, we can solve $\widehat{A}_i^R = \widehat{L}$ for k to find when this bottleneck appears:

$$A_i^R \left(1 + \frac{km}{(n-m)}\right) = \frac{A_i^R}{U_b}(1-k) \quad \Rightarrow \quad k = \frac{1-U_b}{1 + \frac{U_b m}{(n-m)}} \tag{4}$$

Eq. (4) gives the optimal value of the fraction of records k to reassign: reassigning fewer is suboptimal for the original bottleneck; and reassigning more exacerbates a new bottleneck.

Example: Consider a distributed join with 32 workers with one straggling worker whose receive channel utilisation is limited to half the channel. Eq. (4) predicts a maximum improvement of 49.2% from record reassignment. The benefit decreases slightly as more straggling workers are added: for m=5, the maximum improvement is 45.7%. A 50% improvement would be equal to the case without network skew, so reassignment offsets most, but not all, of the effect of network skew among receive channels.

(2) Congested send channels. Next we perform the same analysis for m straggling workers, $\mathbb{M} = \{w_{b_1}, w_{b_2}, \ldots, w_{b_m}\}$ with congested *send* channels, limited to the same join channel utilisation, $U_b^S = A_b^S / L$. Since records cannot be reassigned between input partitions, the only way to reduce the load on a send channel H_b^S is for worker w_b to process more records locally, i.e. reassign them to O_b. Analogous to the congested receive channels above, we reduce the join completion time Δ by $(1-k)$ by reassigning a fraction

$$k = (A_b^S - \widehat{A}_b^S) / A_b^S.$$

Next we analyse the effect on the remaining three types of channels to calculate the maximum improvement of this reassignment:

Case (1): Send channels on non-straggling workers. After reassignment, some sent records change output partitions, but they are still sent via H_i^S. An extra $km(n-1)/(n-m)$ of the records originally processed locally are now also sent via H_i^S. Given that $(n-1)/n$ of the records sent use H_i^S, and $1/n$ use H_i^L, the new join transfer volume for w_i increases:

$$\widehat{A}_i^S = A_i^S \left(1 + \frac{km}{n-m}\right) \tag{5}$$

Case (2): Receive channels on straggling workers. Before reassignment, each straggling worker $w_b \in \mathbb{M}$ received a fraction of $(n-1)/n$ records in O_b via H_b^R. After reassignment, the size of O_b increases by k, of which $(n-1)/n$ arrive via H_b^R:

$$\widehat{A}_b^R = A_b^R (1 + k(n-1)) \tag{6}$$

Case (3): Receive channels on non-straggling workers. Finally, records are taken evenly from all non-straggling workers $w_i \notin \mathbb{M}$. $k(n-1)/(n-m)$ of the records originally assigned to each non-straggling worker are reassigned to each w_b:

$$\widehat{A}_i^R = A_i^R \left(1 - km \frac{n-1}{n-m}\right) \tag{7}$$

Receive channels on straggling workers show the highest join transfer volume increase. Similar to before, we assume a new bottleneck is created when U_b^R=1, and solve for k to find the highest speed-up:

$$A_b^S (1 + k(n-1)) = \frac{A_b^S}{U_b}(1-k) \quad \Rightarrow \quad k = \frac{1-U_b}{U_b(n-1)+1} \tag{8}$$

Compared to congested receive channels, record reassignment is less effective for congested send channels because the reassignment quickly overloads the receive channels of the straggling workers.

Example: With 32 workers and one congested send channel (50% capacity), the maximum reduction in completion time after record reassignment is 3% according to Eq. 8: only $|O_i|(n-m)/n$ records are available at non-congested workers; if $mk|O_b| > O_i(n-m)/n$, k is limited by $(n-m)/mk$, leading to even less improvement.

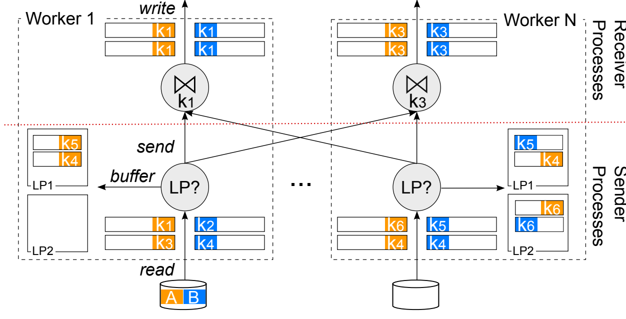

Figure 2: Lazy partitioning (Some lazy partitions are already assigned.)

We conclude from the above analysis that record reassignment can effectively offset network skew among receive channels, but is significantly less effective against skew on send channels. Thus our solution focuses on receive channels, but our evaluation with real-world applications generating background traffic (§6.3) shows that their traffic patterns frequently create receive-side bottlenecks.

4 Lazy Partitioning

We now introduce lazy partitioning to adapt to network skew in distributed join processing. Based on the analysis in §3, we focus on receiver-side skew. We describe lazy partitions (§4.1), how skew is detected and rebalanced (§4.2), and how lazy partitioning behaves without skew (§4.3). We will explore the behaviour of all parameters introduced in this section in §6.6, justifying their default values.

4.1 Lazy partitions

We introduce *lazy partitioning* to implement a practical form of record reassignment. Record reassignment can effectively offset network skew in receiver channels, but assignments must be recorded to ensure consistency if keys appear multiple times. A data structure recording assignments grows with the number of entries, so tracking the output partition of each individual key of large datasets entails significant overhead. Instead we group records into fine-grained partitions: any input partition consists of multiple smaller partitions, $I_i = \{p_{i1}, p_{i2}, \ldots\}$, and each smaller partition is treated as an atomic unit; correspondingly output partitions consist of these same smaller partitions, $O_j = \{p_{j1}, p_{j2}, \ldots\}$. Each record appears in one of these partitions based on its join key.

We call these partitions *lazy partitions* because we can withhold assignment to output partitions until the job is underway: workers buffer records at sender processes prior to an assignment decision that can offset network skew. $O_\varnothing$ denotes the set of lazy partitions not yet allocated to an output partition. Initially, this set contains all records. Assigning a lazy partition $p \in O_\varnothing$ to the output partition on worker w_i removes all partitions with the same join keys across all workers from $O_\varnothing$ and adds them to O_i. When a lazy partition is assigned, all its buffered records are sent to the assigned worker, and future records are sent directly without buffering. At join completion, all lazy partitions must have been assigned, i.e. $O_\varnothing = \varnothing$.

Fig. 2 shows an example of lazy partitioning. Keys k_4, k_5 and k_6 are buffered in lazy partitions. Records with those keys are thus kept by the sender processes, partitioned into two lazy partitions, which can be assigned later. The lazy partitions with keys k_1, k_2 and k_3 have already been assigned to output partitions. When a sender process reads a new record with a given key, it either buffers the record or sends it immediately, depending on whether the key's corresponding lazy partition has been assigned.

Algorithm 1: Detect skew and compute assignment

Input : $\mathbb{W}$: set, $O_\varnothing$: set, τ_{skew}: const, τ_{assign}: const, t: now()

1 $\delta_{slowest} \leftarrow \max_{w_i \in \mathbb{W}} \{\frac{\rho(O_i,t)}{r(i,t)}\}$
2 $\mathbb{W}_{finished} \leftarrow \varnothing$
3 **if** $\min_{w_i \in \mathbb{W}} \{\frac{\rho(O_i,t)}{r(i,t)}\} < \delta_{slowest}(1-\tau_{skew})$ **then**
4   **foreach** $w_i \in \mathbb{W}$ **do**
5         $\delta_i \leftarrow \frac{\rho(O_i,t)}{r(i,t)}$
6   **while** $w_i \notin \mathbb{W}_{finished} \wedge \exists\{p_1 \ldots p_{|\mathbb{W}|}\} \in O_\varnothing$ **do**
7             $\delta'_i \leftarrow \frac{\rho(O_i,t)}{r(i,t)} + \frac{|\{p_1 \ldots p_{|\mathbb{W}|}\}|}{r(i,t)}$
8   **if** $\delta'_i > \delta_{slowest} \vee \delta'_i - \delta_i > \tau_{assign}$ **then**
9                 $\mathbb{W}_{finished} \leftarrow \mathbb{W}_{finished} \cup \{w_i\}$
10   **else**
11                 $O_\varnothing \leftarrow O_\varnothing \setminus \{p_1 \ldots p_{|\mathbb{W}|}\}$
12                 $O_i \leftarrow O_i \cup \{p_1 \ldots p_{|\mathbb{W}|}\}$

Lazy partitions are assigned in two cases: (i) when network skew is detected and a rebalancing decisions is made (§4.2); and (ii) when the lazy partitions at a worker exhaust the allocated memory (§4.3).

4.2 Detecting and balancing skew

Correctly deciding *when* lazy partitions should be assigned to mitigate network skew is critical. *Late detection* policies [26] make decisions after some workers have already finished, and shift load to these workers. Such an approach, however, suffers from head-of-line blocking and would also prevent the reaction to *transient* network skew during the join execution. Instead, we want to make assignment decisions as early as possible, which requires periodic measurements to detect straggling workers.

As the analysis from §3 has led us to focus on the receiver side, we need a measure of progress for each receiver process. Receiver processes continuously monitor their throughput in terms of received data records per second and use it to estimate their completion time. Significant differences in these estimates indicate network skew and trigger an assignment decision.

Each worker maintains a counter γ of the number of processed records. We denote the number of records still to be received by worker i at time t in the assigned partitions O_i by $\rho(O_i, t)$. Assuming that the total number of records is known in advance, e.g. from statistics maintained by a query optimiser, it is possible to compute $\rho(O_i, t) = \rho(O_i, 0) - \gamma$. Using ρ, the rate $r(i, t)$ of a worker w_i at time t is $r(i, t) = (\rho(O_i, t) - \rho(O_i, t'))/(t - t')$, where t' is the time of the last measurement. We denote $t - t'$ as the *assignment interval* τ_{assign}. The node completion time δ_i is then given by:

$$\delta_i = \frac{\rho(O_i, t)}{r(i, t)} \tag{9}$$

We use the estimated completion times of workers to compute a lazy partition assignment. Our algorithm greedily assigns lazy partitions to workers until their new estimated completion time either equals that of the slowest worker or exceeds a specified threshold. Alg. 1 detects network skew and computes the lazy partition assignment using the set of all workers, the set of unassigned lazy partitions, and sensitivity thresholds τ_{skew} and τ_{assign}. A master node executes the algorithm and broadcasts its result to all workers.

First, the slowest worker (line 1) is compared to the fastest worker (line 3). Repartitioning only occurs if the estimated completion time of the fastest worker is less than the *skew threshold*, τ_{skew}, defined as a fraction of the slowest worker. τ_{skew} controls the sensitivity of the approach: smaller values cause more rebalancing decisions, while larger values cause slower reaction. By default, τ_{skew} is set to 5%, which gives robust behaviour in practice (see §6.6).

Algorithm 2: Consume lazy partitions

Input : $\mathbb{W}$: set, $O_{\varnothing}$: set, $\tau_{consume}$: const, t: now()
1 $l \leftarrow \lceil |O_{\varnothing}| \tau_{consume} \rceil$
2 $\delta_{slowest} \leftarrow \min_{w_i \in \mathbb{W}} (r(i,t))$
3 $q = l / \sum_{i=0}^{n} l_i$
4 **foreach** $w_i \in \mathbb{W}$ **do**
5 $\quad l_i = \lceil q \frac{r(i,t)}{\delta_{slowest}} \rceil$
6 $\quad$ **while** $l_i > 0 \land \exists \{p_1 \dots p_{|\mathbb{W}|}\} \in O_{\varnothing}$ **do**
7 $\quad\quad O_i \leftarrow O_i \cup \{p_1 \dots p_{|\mathbb{W}|}\}$
8 $\quad\quad l_i \leftarrow l_i - 1$

If network skew is detected, the algorithm makes an assignment. Since all sender processes use the same partitioning scheme, the algorithm assigns the equivalent partition for each sender process to ensure that workers receive all records that must be joined with each other (line 6). A new estimated completion time is computed for a worker, assuming the given set of lazy partition is assigned to it (line 7). If the estimated completion time exceeds that of the slowest worker or the cumulative increase for this round exceeds a threshold, τ_{assign}, the worker is marked as finished for this round (lines 8–9); otherwise the lazy partition is assigned (lines 10–12).

The complexity of the detection algorithm is O($\mathbb{W}$) because it must scan all workers to find the slowest one. The reassignment heuristic has a worst case complexity of O($O_{\varnothing}$), which occurs when all lazy partitions are assigned in one step.

Using the assignment interval τ_{assign} as an assignment threshold makes the algorithm robust to transient network skew. Fully rebalancing perceived network skew can cause an imbalance if the skew disappears before the join computation finishes. In this case, faster workers may receive too many lazy partitions relative to straggling workers, while consuming lazy partitions that could be used to address such an imbalance. Using an assignment threshold avoids this, and we always assign enough to last at least until the next assignment decision. By default, τ_{assign} is 5 s, and we explore this parameter in §6.6.

Alg. 1 takes into account the number of unsent records in the lazy partitions when making assignments (line 7). The total size of a lazy partition is obtained by linearly extrapolating its current size according to the fraction of input that has not yet been seen. For example, if a lazy partition contains l records and 50% of the input data was processed, the lazy partition accounts for approximately $2l$ records. This makes our approach aware of data skew: the algorithm tracks the aggregate size of the records assigned to each output partition until a threshold is met. It therefore adapts the assignments to offset any differences in the sizes of lazy partitions caused by data skew. Reassignment decisions involving large lazy partitions will assign fewer partitions than those involving many smaller partitions.

4.3 Consumption without network skew

Without network skew, lazy partitions must eventually be assigned to receiver processes to ensure join completion. Each worker has a finite amount of memory M to store lazy partitions. S_i denotes the *lazy partition size*, i.e. the memory required to store all lazy partitions on worker w_i. Once any worker has filled up its memory, $\exists w_i \in \mathbb{W} : S_i = M$, it must *consume* a subset of its lazy partitions to make room for more records.

Alg. 2 formalises the consumption. It first determines the number of lazy partitions that should be consumed (line 1) and then calculates the rate of the slowest worker (line 2). Each worker receives a weighted number of lazy partitions according to this ratio: (i) the average weight is computed (line 3); and (ii), for each worker, the amount to assign is determined (line 5) and assigned (lines 6–8).

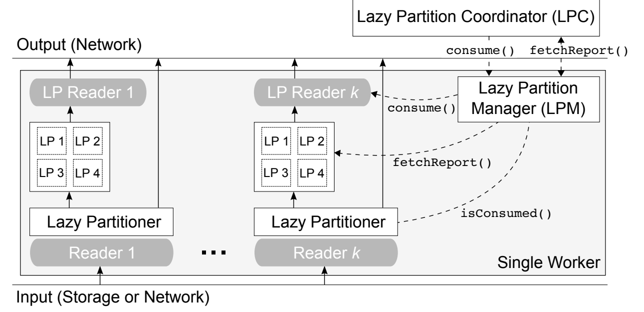

Figure 3: SquirrelJoin architecture (Solid lines represent the data plane; dashed lines represent the control plane.)

This size-weighted assignment accounts for data skew because the number of records is assigned according to the combined sizes, rather than the number of the lazy partitions containing the records. This adds to the join transfer volume by the same proportion for each receiver channel, which increases the estimated completion time δ_i of all channels by the same amount. Because lazy partitions are assigned in groups, the nominal sizes of individual partitions do not affect the increase in δ_i.

A *consumption threshold* $\tau_{consume}$ specifies how many lazy partitions should be consumed when a worker exhausts its memory. The threshold is specified as a percentage over all lazy partitions that have not been assigned yet—as more lazy partitions are assigned, the absolute amount therefore decreases. Ideally, $\tau_{consume}$ should be small to retain as many lazy partitions as possible for later use in rebalancing. If $\tau_{consume}$ is too small, however, workers consume lazy partitions too frequently, increasing overhead. As we show empirically in §6.6, $\tau_{consume} = 5\%$ is a robust default.

$\tau_{consume}$ also determines the number of lazy partitions required. The consumption algorithm selects a set of lazy partitions based on their combined size (line 1). If there are too few lazy partitions, at least one lazy partition cannot be assigned to each worker. Thus the number of partitions required has a lower bound of $|\mathbb{W}|/\tau_{consume}$ when $O_{\varnothing}$ contains all records. As more assignments are made, the size of $O_{\varnothing}$ shrinks geometrically, so it is advisable to have more lazy partitions to ensure an even distribution in later assignments. The upper bound is one partition per key, but there is a significant overhead in tracking assignment decisions for each key.

For example, with 64 workers and $\tau_{consume}$ of 5%, the minimum number of lazy partitions required is 1280. In our experiments, we use approximately three times this number (3200 lazy partitions) to stay well above this calculated minimum.

5 SquirrelJoin Implementation

We now describe SquirrelJoin, an implementation of lazy partitioning on top of the hash-based repartition join algorithm in Apache Flink [18]. SquirrelJoin addresses two challenges: (i) minimising overhead for maintaining and consuming lazy partitions (§5.1); and (ii) creating robust completion time estimates (§5.2).

5.1 Architecture

SquirrelJoin extends the *task-based* architecture of systems such as Flink or Spark. Each worker has multiple readers that read input data records in parallel and send them to receiver processes.

Fig. 3 shows the SquirrelJoin architecture consisting of (i) a *lazy partition coordinator* to make assignment decisions, and (ii) *lazy partition managers* to manage the lazy partitioning on workers. Workers have (i) *lazy partitions* to store records in memory; (ii) *lazy*

partitioners to determine if records should be stored in a lazy partition; and (iii) *lazy partition readers* to consume lazy partitions after assignment. The components communicate via control messages.

Lazy Partition Coordinator (LPC). The LPC is part of the Flink master node and coordinates the lazy partition assignment. It periodically retrieves current progress rates from the LPMs on the workers and runs Algorithm 1 to detect and rebalance skew. It also polls LPMs to retrieve the lazy partition sizes and consumes partitions as per Algorithm 2 when workers exhaust memory. The LPC broadcasts all assignment decisions to the LPMs.

Lazy Partition Managers (LPMs). Each worker has an LPM that reports aggregate statistics from local lazy partitions to the LPC and stores assignment decisions received from the LPC. The decisions are queried by local lazy partitioners to identify the receiver process for a data record whose lazy partition was already assigned.

Lazy Partitioners. Flink exploits the parallelism of multi-core CPUs by running multiple reader tasks at a worker. In SquirrelJoin, each reader is associated with a lazy partitioner, which decides if a data record should be stored in a lazy partition or sent to a receiver process. This decision must be done consistently across all readers on all workers. Lazy partitioners can query the LPM to determine if the lazy partition for a given record has been assigned or not. To reduce the performance impact of adding records to lazy partitions, each reader maintains its own set of lazy partitions. This avoids the need for synchronisation between reader tasks and reduces the latency added by the lazy partitioner.

Lazy Partitions. Lazy partitions are stored in a hash table with a unique identifier per partition as the key and the stored records as the value. Records are stored in byte arrays in serialised form to reduce instantiated objects and garbage collection overheads.

Lazy Partition (LP) Readers. After a lazy partition has been assigned, a LP reader reads and sends records from the assigned lazy partitions to the specified destinations. By consuming lazy partitions concurrently, readers do not require extra logic for receiving and parsing assignment decisions, reducing delay on the critical path.

5.2 Mitigating estimation error

The LPC requires good time estimates to make good assignment decisions. Multiple effects create errors in the estimation, such as the measurement granularity or transient network effects. For example, we found TCP unfairness to be a major error source: TCP implements max-min flow fairness, i.e. each network flow on a shared link receives an equal amount of bandwidth, but TCP converges slowly under congestion [2]. Such estimation errors lead to wrong assignment decisions, which can cause degraded performance.

SquirrelJoin uses three simple yet effective techniques to mitigate estimation errors:

Sliding window average. The LPC takes regular throughput measurements. Single measurements are noisy, so the assignment algorithm uses the average rate over the preceding τ_{assign} interval. After an assignment, there is a back-off period equal to τ_{assign} before the next assignment decision to ensure the throughput is measured after the previous assignment. If no assignment is made, the assignment algorithm executes again after the next measurement.

Friedman test. The Friedman test [19] filters statistically insignificant network skew. Progress rates measured by the LPC naturally differ, with some workers appearing faster than others, resulting in an ordering of workers. If the differences in rates are caused by measurement error, the order of processes will change randomly between measurements, but a consistent phenomenon such as network skew maintains the same ordering. The Friedman test determines if the ordering is consistent across multiple measurements. We set the detection level of the Friedman test to 1%.

Skew thresholds. In addition to the relative skew threshold τ_{skew} (see §4.2), an absolute skew threshold is used to avoid addressing insignificant skew. Even if statistically significant skew is detected and the relative difference is above t_{skew}, reacting can be wasteful if the expected absolute difference is small—instead lazy partitions could be used to mitigate more severe future network skew. Thus the LPC only reacts to network skew that is predicted to extend job completion time by more than 5 seconds.

5.3 Multi-join execution

In a query with multiple joins, each join is executed independently, i.e. it allocates and maintains its own lazy partitions and has separate LPM and LPC instances. Progress measures for one join only affect the lazy partition assignment of that join, which guarantees that a join only reacts to skew that occurs during its execution. Without data dependencies between the joins, it is straightforward to support different join trees, such as bushy or left/right-deep trees, during an n-way join. This makes SquirrelJoin compatible with existing query optimisers because it does not impose restrictions on join ordering.

6 Evaluation

We evaluate SquirrelJoin's effectiveness for different queries (§6.2) and different sources and types of background traffic (§6.3). We also explore the robustness of its measurements (§6.4) and consider its scalability (§6.5). Finally, we analyse different choices for the available parameters (§6.6).

6.1 Experimental set-up

We deploy Apache Flink [18] on a 17-node cluster (16 nodes with 4 workers each and 1 master node) on Google Compute Engine. We use "n1-standard-16" VMs with 16 CPU cores at 2.5 GHz and with 60 GB of memory. We allocate 12 GB of memory to each worker. We configure the VMs (using the Linux `tc` tool) to have virtual 1 Gbps NICs. We store input data on a RAM disk to ensure that storage I/O is not a bottleneck.

Queries. As our workload, we consider the join queries Q2, Q3, Q9, Q10, Q16, and Q21 from the TPC-H benchmark [40]. The queries cover a large variety of workloads with different numbers of joined tables, small and large input table sizes, and different run times. We project away fewer attributes to saturate the network links, and preprocess input tables, applying the selections, to observe each join in isolation. We use a scale factor of 300 for all queries except Q10, which we use for our scalability experiment. For Q10, we use a scale factor of 1000, giving each preprocessed table a size of roughly 160 GB. We use a subset of 40 GB per table and later vary the subset size (see §6.2 and §6.5). All input data is stored in HDFS. Note that we do not include the `nation` and `region` tables because they would be joined via a broadcast join.

We compare SquirrelJoin to the default hash-based repartition join implementation in Flink. Each experiment is repeated 5 times, and we report the 25th, 50th and 75th percentiles as errorbars.

6.2 Query workloads

To analyse SquirrelJoin's effectiveness, we evaluate it with different join queries under network skew and for a workload with data skew.

Join queries. We use the above TPC-H queries and run each query with and without network skew. To generate network skew, we use `iperf`, a bandwidth measurement tool, to create 30 competing TCP flows from an external machine to one VM. All flows start 1 s after the query and last for 300 s, affecting all joins.

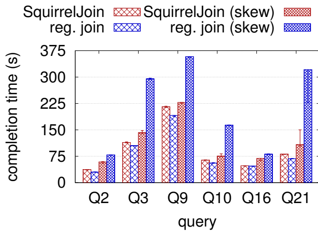

Figure 4: Queries

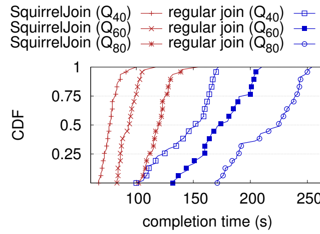

Figure 5: Probe table size

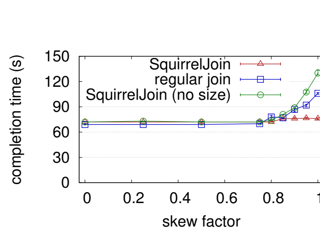

Figure 6: Data skew

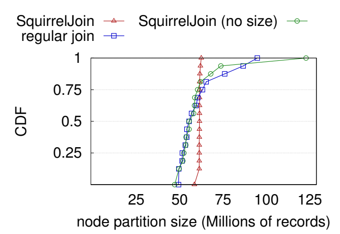

Figure 7: Received records (s=1)

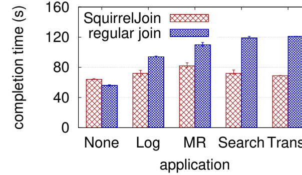

Figure 8: Background applications

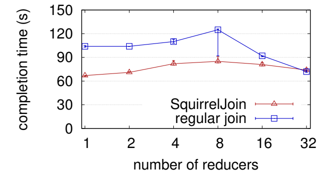

Figure 9: `Terasort` application

Fig. 4 shows that SquirrelJoin detects and balances network skew for all queries, and its performance is consistently similar to a deployment without background traffic. It achieves speed-ups of up to 2.9× over the regular join. Without network skew, the start-up phase adds a small overhead, which we analyse in §6.5.

The input tables of Q3, Q10, and Q21 are large (> 10 GB), resulting in the highest benefit achieved by SquirrelJoin. In these cases, transfers are long, and SquirrelJoin can assign most of the input data to non-congested receivers. Q9 is a 4-way join, which takes roughly 200 s to complete without network skew. Due to the longer runtime, SquirrelJoin's benefit under network skew decreases because the background traffic finishes before the query. Nevertheless, SquirrelJoin still improves completion time by 1.5×.

Note that, for queries with small input tables (Q2 and Q16), the reaction time of SquirrelJoin relative to the size of the input tables is slower. This means that the bulk of the table has already been sent before the first assignment decision, and SquirrelJoin provides less benefit. However, it still balances network skew for the remainder of the input data, leading to faster completion times.

Probe table sizes. To emulate longer running joins, we use the TPC-H query 10 and vary the size of the `lineitem` table by increasing the subset of data from the original preprocessed table. We use three differently sized `lineitem` tables, 40 GB (Q_{40}), 60 GB (Q_{60}) and 80 GB (Q_{80}). Network skew starts randomly at 1–40 s after the join and lasts for its remainder. We rerun each query 50 times and plot the cumulative distribution function (CDF).

Fig. 5 shows that, for larger probe tables, the CDF for both SquirrelJoin and the regular join shifts to the right by approximately 20 s as the overall join takes longer to complete. The shape of the CDF, however, does not change. This shows that SquirrelJoin scales with the size of the probe table and can retain lazy partitions long enough to also balance late-occurring network skew.

Compared to the regular join, whose performance depends on the start time of the network skew, SquirrelJoin is largely unaffected by this, only deviating 10%–15% from the median. SquirrelJoin does experience a short tail around the 95th percentile. In some cases, network skew starts right after a consumption decision was made. This means that fewer partitions are available for rebalancing compared to when the skew starts before the consumption decision.

Data skew. Next we study how SquirrelJoin handles data skew. TPC-H data have uniformly distributed join keys, so we generate a skewed dataset according to Rödiger et al. [33]. The dataset models urban population and joins the primary key of a `city` table to a foreign key of a `people` table. The foreign key is skewed and randomly drawn from a Zipf distribution because many people live in few large cities. The `people` relation contains 1 billion records, and `city` has 100 million records. Both tables are partitioned in equally-sized input partitions across all nodes. We vary the Zipf factor s from 0 (uniform) to 1 (highly-skewed) and compare the regular join to SquirrelJoin and a version of SquirrelJoin without the size-weighted assignment that handles data skew (see §4.2).

Fig. 6 shows that the data skew only affects completion times when the skew factor s is ≥ 0.85. This is consistent with previous findings [33] that basic hash partitioning is sufficient to balance most data skew. When $s = 1$, the regular join exhibits a slow-down of 1.5×, which is similar for SquirrelJoin without size-weighted assignment. SquirrelJoin with size-weighted assignment, however, tracks the lazy partition sizes and balances the assignments to achieves a stable completion times for all skew factors. Fig. 7 confirms the even data size distribution by showing the distribution of received records across the workers in a CDF.

6.3 Network skew

To study how effective SquirrelJoin handles different types of network skew, we run TPC-H query 10 with a variety of background traffic patterns. We start off with network skew caused by background traffic from real-world applications sharing the network.

Background traffic. We use the following applications, commonly found in shared clusters, to introduce background traffic:

(1) Log aggregation. Log aggregation services collect logs from cluster machines and consolidate them at one or few aggregator servers. This "many-to-few" traffic is susceptible to generate network skew, for example, when the cluster experiences a high volume of log data due to ongoing error reporting. We deploy Facebook's Scribe [35] with one aggregator server.

(2) Data analytics. Besides Flink, a cluster may host other data processing frameworks such as Hadoop, Spark or TensorFlow. A job containing few operators with large fan-ins, e.g. a reducer during a MapReduce job or a parameter server for distributed machine learning, can generate receiver-side skew. We run a `Terasort` job [22] in MapReduce with 32 mappers and 4 reducers.

(3) Distributed search. Distributed search engines also follow a many-to-few pattern. Incoming search requests are partitioned and processed by backend servers, and the results are collected at a frontend server. Request bursts can cause substantial network skew. We use Apache Solr [37] as a distributed search system.

(4) Data transfer. Large point-to-point data transfers occur in clusters, e.g. during VM migration or large file uploads. Such transfers can be accelerated by parallelising multiple TCP connections. We emulate a VM migration operation by transferring an 8 GB Debian image to one of the Flink workers. We uses 30 TCP flows to transfer 8 GB, so the skew disappears before the join completes.

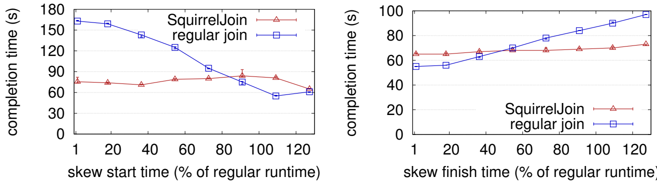

Figure 10: Network skew start time

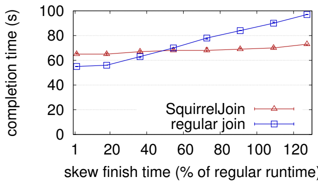

Figure 11: Network skew duration

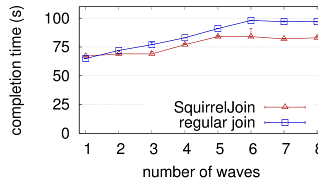

Figure 12: Waves of network skew

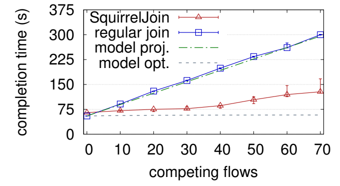

Figure 13: Network skew severity

We run Q10 with each of the interfering applications separately and plot the results in Fig. 8. Regular join experiences up to a 2× slow-down with background traffic as the number of competing TCP flows that each application generates equals the number of flows of each worker (15), thus halving the bandwidth available to Flink. SquirrelJoin detects the network skew in all cases and adapts accordingly, achieving speed-ups up to 1.7× over regular join.

When background traffic is distributed evenly across nodes, all Flink workers are affected equally, thus reducing individual stragglers and decreasing SquirrelJoin's benefit. In Fig. 9, we vary the number of reducers for a background `Terasort` MapReduce job. The gap between regular join and SquirrelJoin decreases as the traffic is spread across more reducer nodes. With more reducers, the time of the shuffle phase, and thus the duration of the background traffic, is also decreased, affecting the join less.

Another observation is that the regular join experiences a higher variance for 8 reducers. With more reducers, it is more likely that MapReduce schedules multiple reducers on a single node, thus increasing the number of competing TCP flows to that node. SquirrelJoin is resilient to such issues.

Network skew start times. To evaluate SquirrelJoin's ability to retain lazy partitions when network skew appears later during the join computation, we vary the time at which synthetic network skew begins. We delay the onset of the 30 competing `iperf` flows by different percentages of the join completion time (without background traffic) and then run them until the join completes.

Fig. 10 shows that, when the network skew occurs early, the regular join experiences a 2.9× slow-down as 30 competing TCP flows cause Flink's available bandwidth to drop to roughly 1/3. As expected, the later the network skew occurs during the join execution, the less impact it has on the join completion time until it completes before the background traffic starts (110%).

SquirrelJoin's completion time is around 80 s, regardless of network skew, yielding a maximum improvement of 2.1×. The stable completion times indicate SquirrelJoin consumes lazy partitions slowly enough to always have data available for rebalancing. SquirrelJoin cannot eliminate all effects of network skew because it sends some data records over the congested network link to estimate completion times, and it adds a start-up overhead. This means that late-occurring network skew affects SquirrelJoin longer than the regular join, explaining the difference at 110%.

Network skew finish times. Next, we investigate how SquirrelJoin behaves when network skew finishes before the join completes. This shows SquirrelJoin's robustness against short periods of network skew. We repeat the same experiment as above but now always introduce network skew when starting the join and then vary its duration, again in terms of percentage of the join completion time.

Fig. 11 shows that the regular join experiences a linear increase in completion time with longer network skew, while SquirrelJoin keeps completion times constant. We conclude that lazy partitions can handle transient network skew without overreacting.

If the network skew is short, the impact on the regular join is low, and the start-up overhead of SquirrelJoin dominates the completion time. At around 50%, which translates to approximately 30 s, the lines cross and the regular join becomes worse. As discussed in §2.2, typical network skew lasts for more than 10 s, often lasting for 100 s or more. While SquirrelJoin introduces an overhead for periods shorter than 30 s, it provides a benefit for most realistic longer occurrences of network skew.

Multiple waves of network skew. We now explore a more dynamic environment, combining the above two scenarios. We initiate different waves of transient network skew. Each wave lasts for 15 s and generates skew at a different node. We create up to 8 waves. Waves are continuous, i.e. once the previous wave has finished, the next wave starts immediately. We start the first wave after 10 s.

Fig. 12 shows that, with more waves, the completion times of the regular join increase linearly until 7 waves when it saturates. The reason is that the join finishes before wave 7 occurs and, hence, the completion time is not further affected. SquirrelJoin is stable until 3 waves and then experiences a slight increase until 5 waves, after which it is stable again at 80 s.

In summary, SquirrelJoin consistently outperforms the regular join and produces stable results in a highly dynamic environment. Its robustness mechanisms and gradual partition assignment prevent wrong decisions when skew affects nodes for short time periods.

Network skew severity. SquirrelJoin should detect and mitigate network skew independently of its severity. We now vary the number of competing TCP flows generated by `iperf`.

Fig. 13 shows the completion times for SquirrelJoin and the regular join. It also includes the times as predicted by our model for the base case without any reassignment (`model projected`) and the ideal case in which perfect knowledge about network skew is available a priori and an optimal assignment can be computed before running the join (`model optimal`).

The regular join experiences a linear increase in completion time as we increase the number of competing TCP flows. This is expected because TCP's max-min fairness proportionally splits the available network bandwidth between different flows and, hence, the available bandwidth for the join decreases with more background flows.

SquirrelJoin also experiences an increase in completion time, but the increase is less steep. The gap between the regular join and SquirrelJoin increases until SquirrelJoin reaches a speed-up of approximately 2.3× for 40 competing TCP flows. After that, the difference stays nearly constant. The reason is that even though SquirrelJoin correctly detects and rebalances the network skew, it must send a small amount of data over the shared link to obtain reliable progress measurements. The more competing network flows use that link, the longer it takes for SquirrelJoin to transmit the data.

The time predicted by `model projected` closely matches the observed time of the regular join. As the optimal model assumes perfect a priori knowledge of the network skew, it does not account for any data sent over the shared link and, hence, `model optima` and SquirrelJoin diverge slightly. The figure shows that SquirrelJoin achieves at least 70% of the maximum theoretically possible improvement for 20 or more competing flows.

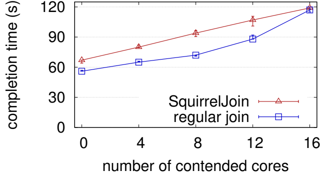

Figure 14: CPU skew

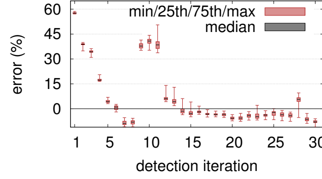

Figure 15: Estimation error

Figure 16: Cluster size

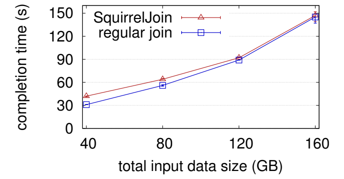

Figure 17: Input data sizes

Figure 18: Algorithm scaling

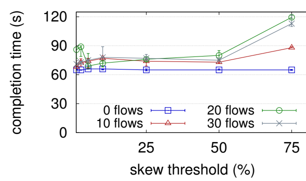

Figure 19: Parameter τ_{skew}

6.4 Robustness

Next we explore the robustness and accuracy of SquirrelJoin's measurement and assignment mechanisms under different conditions.

Uneven CPU load. We first investigate the behaviour of SquirrelJoin when stragglers are caused by imbalances in CPU utilisation. We run the join from TPC-H Query 10 and vary the number of contended CPU cores on one node using the Linux `stress` tool (see Fig. 14) without network skew.

The performance of both the regular join and SquirrelJoin decreases linearly with more contended CPU cores. SquirrelJoin performs slightly worse due to its overhead (see §6.5), but the relative difference remains almost constant. CPU utilisation is bidirectional and affects senders and receivers evenly. Hence it is not visible in the receiver rates, and SquirrelJoin does not react to it.

While SquirrelJoin is robust to CPU contention, we believe that, in practice, such situations are rare. Cluster schedulers such as Yarn or Mesos already isolate CPU resources to prevent CPU imbalances.

Estimation accuracy. We also assess the accuracy of SquirrelJoin's completion time estimates. We execute the join without network skew and record the estimates every second, comparing them to the actual time that it took to complete the join. We run 4 workers per node, giving us 64 estimates per measurement. We show the range of errors for each measurement in Fig. 15. Negative error indicates the completion time was overestimated.

Initially the errors are high (we exclude the first two data points, which are above 100%) due to the start-up effects before senders send at their full rates. Despite the high error, the variance of the error in these measurements is low, i.e. the completion time is underestimated equally for all receivers. After 6 iterations, the magnitude of the error is less than 10%, and the variance remains low.

The error increases again at around iteration 10 during the change between the two phases of the two-phase hash join. After the build phase finishes there is a short interruption between the phases before data from the probe table is sent, which affects progress rate measurements. After that, errors are consistently between 0 and -10%. To avoid wrong decisions during the phase change, SquirrelJoin does not balance network skew after the first sender and before the last sender has finished the build phase.

Our error estimates are accurate and, more importantly, have low variance. This, combined with the robustness mechanisms from §5.2, ensures good assignment decisions.

6.5 Scalability and overhead

We analyse SquirrelJoin's scalability and overhead with different cluster and input data sizes.

Cluster size. We deploy Flink with three cluster sizes (16, 24 and 32 nodes), and compare SquirrelJoin and the regular join with and without network skew. We scale the input data with the cluster size to keep the same data-to-node ratio and allocate 20 GB of memory to each Flink worker in order to handle the increase in output size due to more join matches. Since the number of network flows to each worker in the join increases with the number of nodes, we also increase the number of competing background flows, thus maintaining the same network bottleneck utilisation in all deployments.

Fig. 16 shows consistent behaviour across cluster sizes; there is little difference between SquirrelJoin and regular join without network skew while SquirrelJoin outperforms regular join under network skew. Overall, we observe a small increase in completion times as the cluster size grows: the Flink implementation experiences higher overheads in larger clusters, and there is an increase in computation required to finish processing the growing set of output records per node. These effects impact both approaches equally.

Input data size. We compare the regular join and SquirrelJoin without network skew as input tables grow. We use the same preprocessed tables as before and increase the subset of data from 20 GB to 80 GB per table until the join would start spilling.

Fig. 17 shows the same overhead of 10 s for SquirrelJoin for 40 GB and 80 GB of input data. This fixed overhead is from the start-up phase of SquirrelJoin in which the network is not used before the first assignment decision. For larger data sizes, the difference decreases until both joins exhibit the same behaviour for 160 GB of input data. For larger partitions of the input tables, more matches are found, and the join computation shifts from being network-bound to CPU-bound, reducing SquirrelJoin's impact on completion time.

Algorithmic overhead. SquirrelJoin adds overhead by executing the assignment algorithm on the LPC (see §5.1). Other overheads, such as simple computations and data structure lookups, require constant time, but the execution time of the assignment algorithm grows with the cluster size. We investigate if this can create a bottleneck by running the algorithm for various cluster sizes.

Fig. 18 shows the execution time of the algorithm for various numbers of workers (64 to 16,384). The results suggest the algorithm is unlikely to be a bottleneck. Even for extremely large clusters the times are in the range of tens of milliseconds. Note that the assignment algorithm runs in parallel with the join computation, and the system never pauses to wait for the result. At worst, the runtime of the algorithm increases the reaction latency. We show in §6.6 that this negligibly impacts the effectiveness of lazy partitioning.

6.6 Parameter choices

We finish with a sensitivity analysis of SquirrelJoin's parameters in order to establish the generality of our default values.

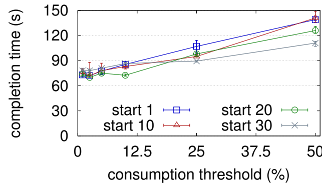

Figure 20: Parameter $\tau_{consume}$

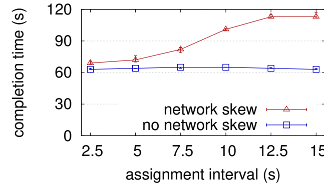

Figure 21: Parameter τ_{assign}

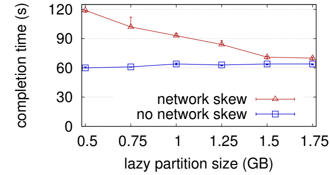

Figure 22: Lazy partition size

Skew threshold. We vary τ_{skew} from 1% to 75%, with higher values being more sensitive to network skew. We use background traffic intensities from zero to 30 competing flows to find a robust value.

Fig. 19 shows that values of τ_{skew} around 5% perform well for all intensities of background traffic. This value is not brittle in that all values between 5 and 50% perform similarly. Staying on the lower end of that range allows us to maintain more partitions to adapt to future network skew changes. Below this range, SquirrelJoin is too sensitive, leading to bad decisions—we observe high completion times for 20 competing TCP flows with τ_{skew} values of 1% and 2.5% because SquirrelJoin reacts to noise. With very large values network skew is ignored as noise, and no adjustments are made.

Consumption threshold. Next we evaluate $\tau_{consume}$. Higher values cause more lazy partitions to be consumed when the available memory fills up before network skew is detected. We vary the start time of the background traffic from 1 s to 30 s after the start of the join.

Fig. 20 shows that SquirrelJoin achieves the best performance for τ_{skew} values between 1% and 5%. Higher percentages cause a slowdown of up to 1.9× due to the earlier consumption of lazy partitions, which leaves fewer for later rebalancing. This even affects SquirrelJoin when the network skew begins shortly after the start of the join because an initial consumption is required to start sending out data and collect progress measurements.

Assignment interval. We vary the assignment interval, τ_{assign}, from 2.5 s to 15 s with and without network skew. Shorter intervals result in more reactive reassignments, but are less robust to noisy measurements and bursty traffic.

Fig. 21 shows that the low values of 2.5 s and 5 s perform better in the presence of network skew. Larger values lead to a slower reaction—more lazy partitions are assigned evenly across receivers when workers' memory fills up more often before the first assignment decision. This exacerbates the effect of the network skew while consuming lazy partitions. The assignment interval does not affect performance without background traffic because SquirrelJoin does not make assignment decisions.

We observe that even the shortest intervals are not susceptible to noise in this set-up using constant, steady background traffic. This shows our noise mitigation techniques (see §5.2) help improve the robustness of SquirrelJoin. Setting τ_{assign} to 5 s increases the number of measurements for the χ^2 approximation in the Friedman test, making SquirrelJoin more robust to transient skew.

Lazy partition size. Since the overhead depends on when the first consumption decision is taken, we evaluate the impact of the amount of memory that is allocated for storing lazy partitions. We execute the join with and without network skew from 30 competing TCP flows and vary the allocated memory from 500 MB to 1.75 GB.

As shown in Fig. 22, without network skew, less memory reduces the start-up overhead because lazy partitions are consumed more frequently. A 500 MB allocation reduces the overhead by 4 s, roughly 50% less than for sizes above 1.5 GB. Less memory, however, also limits the amount of network skew that can be rebalanced. More lazy partitions are consumed before SquirrelJoin has enough information to identify and address network skew. With 500 MB, the benefit of rebalancing decreases by 98%. At around 1.75 GB, SquirrelJoin can offset most network skew, and the overhead is still reasonably small.

7 Related Work

Distributed join processing. Kitsuregawa et al. introduced *hash-based joins* [25], which DeWitt et al. [14] parallelised. Their approach is still employed by modern massively-parallel data processing systems such as Hive [39] or SparkSQL [4]. Other work studied joins in these systems, e.g. using *semi-joins* [9] or implementing *multi-way joins* [1] for traffic reduction. These approaches, however, do not consider dynamic adaptation and thus suffer from variations in network performance.

Recent work optimises join network traffic by careful data partitioning. Rödiger et al. mitigate data skew and achieve equally sized partitions with adaptive *radix partitioning* and coordinate network traffic by scheduling communication to maximise network utilisation [33]. *FlowJoin* [32] also tries to determine optimal partitioning with minimal overhead by sampling the input at the beginning of the join and determining a distribution-aware partitioning mechanism before the join begins. *Track Join* [30] introduces a tracking phase to identify where matching tuples are stored for particular keys, then derives an optimal partitioning to minimise the total amount of traffic. Lazy partitioning at runtime is orthogonal. These techniques can improve the partitioning used as input for SquirrelJoin.

Adaptive query processing. Other works suggest adapting query plans to changes in resource availability. *Query scrambling* [3] dynamically executes parts of a query operator tree not stalled by reads to hide network delays when reading from widely-distributed data. As this approach only changes the execution order of operators at inter-operator granularity, unlike SquirrelJoin, it does not have to ensure the consistency of operator state. Other approaches [38, 41] schedule background processing when incoming tuples for an operator are delayed, but this assumes other tasks are ready to run, which is not always the case for distributed joins.

Eddies [5, 13] act as tuple routers: they change the order in which tuples are processed by operators, essentially reordering the query plan at runtime. To undo bad routing decisions, Deshpande and Hellerstein extend eddies to modify and migrate operator state [13]—something we want to avoid due to the additional network overhead.

The *Flux* operator [36] balances transient data skew, which is similar to our approach. By keeping a single buffer per receiver at the senders, Flux changes the arrival order of incoming tuples and thereby avoids head-of-line blocking. However, it does not repartition data and is hence unable to balance more persistent skew.

Other recent approaches focus on *adaptive theta joins* [16] with provable bounds for state migration cost to minimise, but not fully compensate for, the migration effort. Xiong et al. [45] use software-defined networking (SDN) to capture the current network state and adjust query plans accordingly. This adaptation only occurs at query granularity and requires SDN support, limiting the applicability in public cloud environments.

Optimal partitioning. Gedik describes a *hybrid hash function* for flexible data partitioning, but since partitions are not entirely buffered at senders, state migration is still necessary [20]. *Resource Bricolage* [29] defines an optimisation problem to maximise utilisation in heterogeneous clusters. While our work shares the heterogeneity assumption, it focuses on adapting partitioning dynamically. *Co-partitioning* data such as in CoHadoop [17] completely avoids shuffling data but requires joins to be always on the same key—an assumption that does not hold for many analytical jobs.

SkewTune [26] mitigates the effects of data skew in MapReduce by repartitioning data across output nodes when reduce tasks finish early. This approach requires ordered input data, making the solution less general. Vernica et al. [43] use *situation-aware mappers* to implement dynamic data partitioning in MapReduce. The approach has a similar approach as lazy partitioning in that it buffers data at senders until a histogram, for determining an optimal partitioning, has been sampled. However, it is less adaptive and repartitions all records in a single assignment.

Network isolation. Controlled sharing of network bandwidth, e.g. through bandwidth guarantees to tenants in data centres [6, 7, 28], can avoid bandwidth fluctuations and stabilise application performance. This approach introduces two main challenges: (i) bandwidth guarantees cannot prevent network variations caused by multiple applications executed by the same tenant; and (ii) tenants rarely know the exact bandwidth requirements of their applications, resulting in either increased costs or lower performance. These problems make our approach still useful in clusters that enforce bandwidth guarantees.

Traffic scheduling maximises network utilisation via coordinated data transmissions [10, 12]. Chowdhury et al. decide on the location of write replicas in a distributed file system according to current bandwidth utilisation [10]. This is similar to our work, but the file granularity disallows adapting the replica destinations during writing. Existing approaches perform a priori optimisations or consider the network without regard to application semantics. This reduces possible performance gains for concrete algorithms such as distributed joins, which can benefit from altering the network flows.

8 Conclusions

We presented lazy partitioning for distributed joins, a new technique that dynamically adapts to network skew in shared compute clusters. It delays the assignment of partitions to join workers in order to permit future reassignment decisions that offset network skew.

We implemented lazy partitioning in SquirrelJoin for Apache Flink and showed its effectiveness in reacting to network skew. SquirrelJoin measures individual progress rates of join workers and makes assignment decisions based on robust completion time estimates, thus maximising network utilisation and avoiding congested network paths. It therefore significantly improves join completion times in shared compute clusters under background network traffic with minimal overhead when no skew exists.

Acknowledgements. This work was in part supported by grant EP/K032968 ("NaaS: Network-as-a-Service in the Cloud") from the UK Engineering and Physical Sciences Research Council (EPSRC).

9 References

[1] F. N. Afrati and J. D. Ullman. Optimizing Joins in a Map-Reduce Environment. In *EDBT*, 2010.

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, et al. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[3] L. Amsaleg, A. Tomasic, M. J. Franklin, and T. Urhan. Scrambling Query Plans to Cope with Unexpected Delays. In *PDIS*, 1996.

[4] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, et al. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, 2015.

[5] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *SIGMOD*, 2000.

[6] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards Predictable Datacenter Networks. In *SIGCOMM*, 2011.

[7] H. Ballani, K. Jang, T. Karagiannis, C. Kim, et al. Chatty Tenants and the Cloud Network Sharing Problem. In *NSDI*, 2013.

[8] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, 2010.

[9] S. Blanas, J. M. Patel, V. Ercegovac, et al. A Comparison of Join Algorithms for Log Processing in MapReduce. In *SIGMOD*, 2010.

[10] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging Endpoint Flexibility in Data-intensive Clusters. In *SIGCOMM*, 2013.

[11] M. Chowdhury, Z. Liu, A. Ghodsi, et al. HUG: Multi-Resource Fairness for Corr. and Elastic Demands. In *NSDI*, 2016.

[12] M. Chowdhury and I. Stoica. Efficient Coflow Scheduling Without Prior Knowledge. In *SIGCOMM*, 2015.

[13] A. Deshpande and J. M. Hellerstein. Lifting the Burden of History from Adaptive Query Processing. In *VLDB*, 2004.

[14] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, et al. The Gamma Database Machine Project. *TKDE*, 2(1), 1990.

[15] D. J. DeWitt, R. H. Katz, F. Olken, et al. Implementation Techniques for Main Memory Database Systems. In *SIGMOD*, 1984.

[16] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch. Scalable and Adaptive Online Joins. *PVLDB*, 7(6), 2014.

[17] M. Y. Eltabakh, Y. Tian, F. Özcan, et al. CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop. *PVLDB*, 4(9), 2011.

[18] Apache Flink. `https://flink.apache.org`, 2017.

[19] M. Friedman. The Use of Ranks to Avoid the Assumption of Normality Implicit in the Analysis of Variance. *Journal of the Am. Stat. Assoc.*, 32(200), 1937.

[20] B. Gedik. Partitioning Functions for Stateful Data Parallelism in Stream Processing. *The VLDB Journal*, 23(4), 2014.

[21] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, et al. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[22] The HiBench Suite. `https://github.com/intel-hadoop/HiBench`, 2017.

[23] B. Hindman, A. Konwinski, M. Zaharia, et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, 2011.

[24] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, et al. The Nature of Data Center Traffic: Measurements & Analysis. In *IMC*, 2009.

[25] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of Hash to Database Machine and its Arch. *New Generation Comp.*, 1(1), 1983.

[26] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: Mitigating Skew in MapReduce Applications. In *SIGMOD*, 2012.

[27] K. LaCurts, S. Deng, A. Goyal, and H. Balakrishnan. Choreo: Network-aware Task Placement for Cloud App. In *IMC*, 2013.

[28] J. Lee, Y. Turner, M. Lee, L. Popa, et al. Application-driven Bandwidth Guarantees in Datacenters. In *SIGCOMM*, 2014.

[29] J. Li, J. Naughton, and R. V. Nehme. Resource Bricolage for Parallel Database Systems. *PVLDB*, 8(1), 2014.

[30] O. Polychroniou, R. Sen, and K. A. Ross. Track Join: Distributed Joins with Minimal Network Traffic. In *SIGMOD*, 2014.

[31] L. Popa, P. Yalagandula, S. Banerjee, et al. ElasticSwitch: Practical Work-conserving Bandwidth Guarantees for Cloud Comp. In *SIGCOMM*, 2013.

[32] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. Flow-Join: Adaptive Skew Handling for Distributed Joins over High-speed Networks. In *ICDE*, 2016.

[33] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, et al. Locality-Sensitive Operators for Parallel Main-Memory Database Clusters. In *ICDE*, 2014.

[34] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime Measurements in the Cloud. *PVLDB*, 3(1-2), 2010.

[35] Facebook Scribe. `https://github.com/facebookarchive/scribe`, 2017.

[36] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *ICDE*, 2003.

[37] Apache Solr. `http://lucene.apache.org/solr`, 2017.

[38] Y. Tao, M. L. Yiu, D. Papadias, et al. RPJ: Producing Fast Join Results on Streams Through Rate-based Optimization. In *SIGMOD*, 2005.

[39] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, et al. Hive - A Petabyte Scale Data Warehouse Using Hadoop. In *ICDE*, 2010.

[40] The TPC-H Benchmark. `http://www.tpc.org/tpch/`.

[41] T. Urhan and M. J. Franklin. XJoin: A Reactively-scheduled Pipelined Join Operator. *Data Engineering Bulletin*, 23(2), 2000.

[42] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, et al. Apache Hadoop Yarn: Yet Another Resource Negotiator. In *SoCC*, 2013.

[43] R. Vernica, A. Balmin, K. S. Beyer, and V. Ercegovac. Adaptive MapReduce using Situation-aware Mappers. In *EDBT*, 2012.

[44] J. L. Wolf, P. S. Yu, J. Turek, and D. M. Dias. A parallel hash join algorithm for managing data skew. *TPDS*, 4(12), 1993.

[45] P. Xiong, H. Hacigumus, and J. F. Naughton. A Software-defined Networking Based Approach for Performance Management of Analytical Queries on Distributed Data Stores. In *SIGMOD*, 2014.

[46] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling Data Skew in Parallel Joins in Shared-nothing Systems. In *SIGMOD*, 2008.