
yelp
Spark+AI Summit
April 24, 2019

Smart join algorithms
for fighting skew at scale

Andrew Clegg, Applied Machine Learning Group @ Yelp

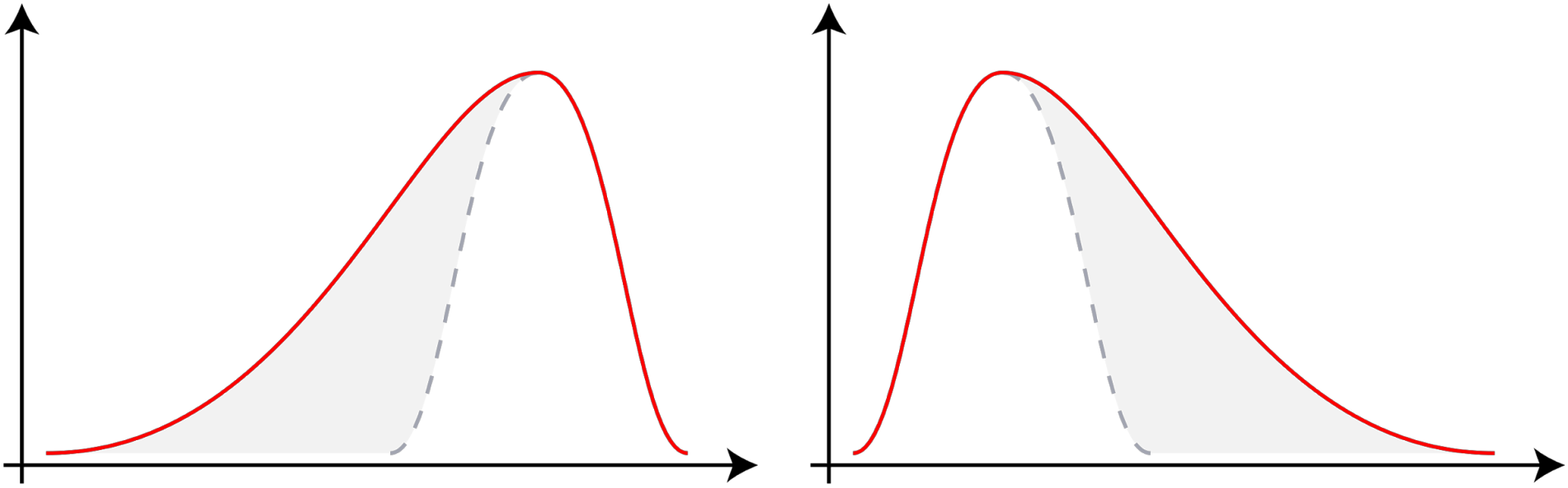
@andrew_clegg

Overview

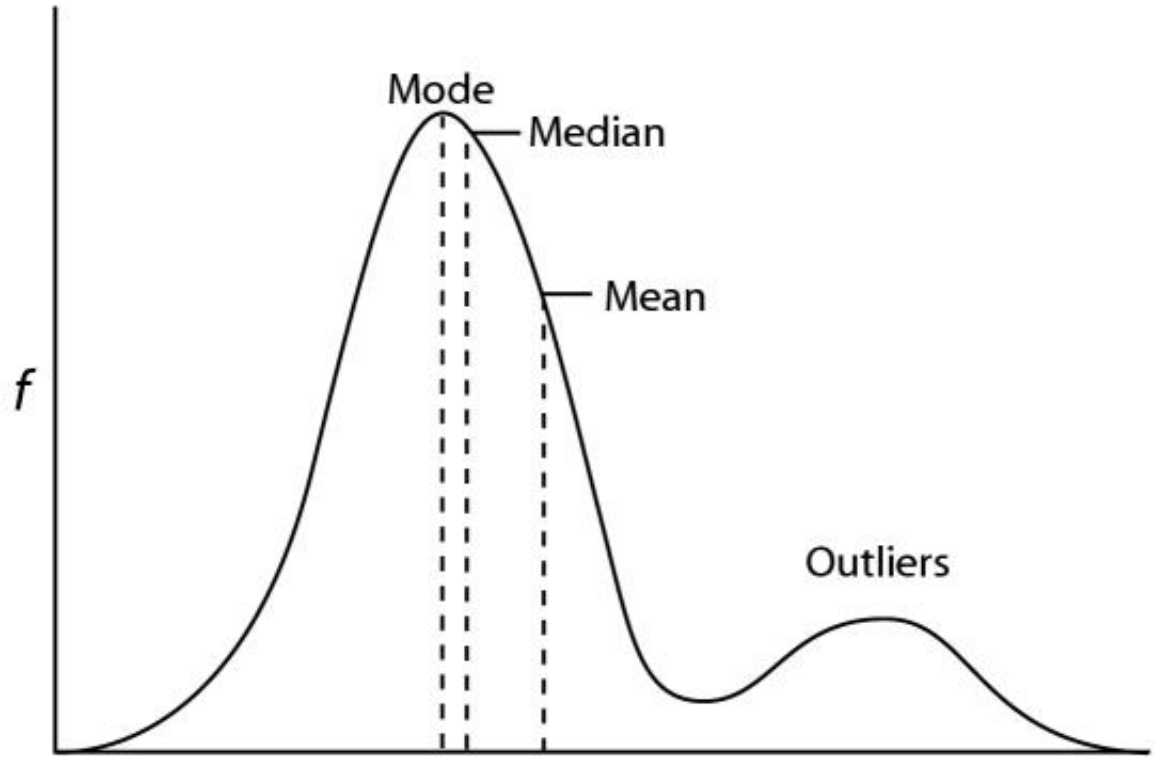
- **Data skew**, outliers, power laws, and their symptoms
- How do **joins** work in Spark, and why skewed data hurts
- Joining skewed data **more efficiently**
- Extensions and tips

Data skew

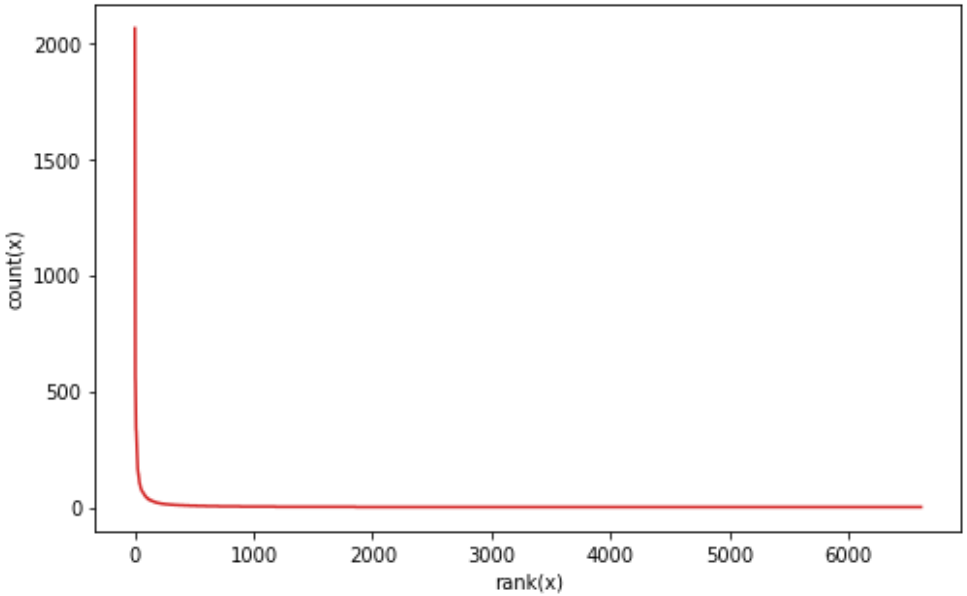
Classic skewed distributions

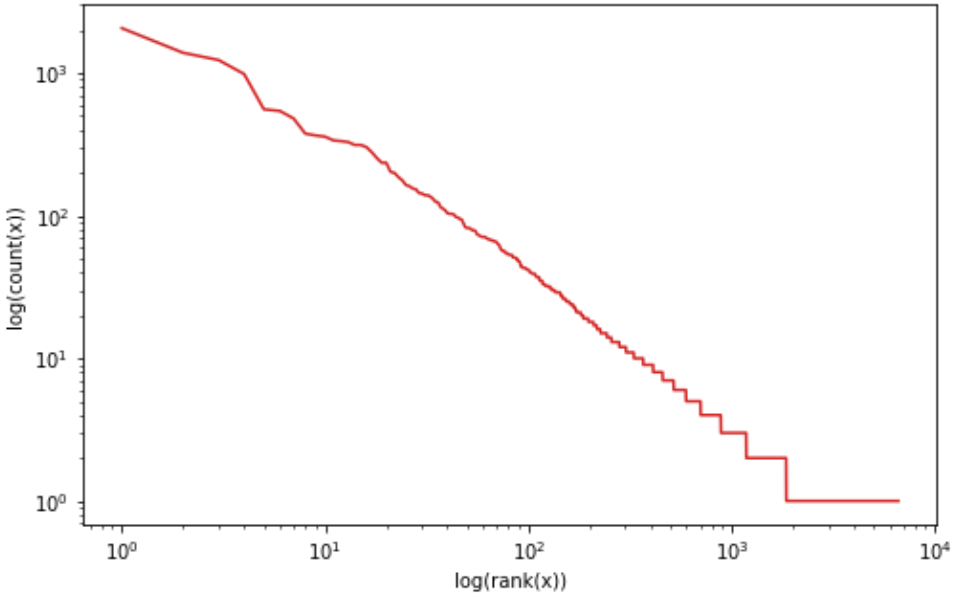


Outliers



Power laws





Word rank vs. word frequency in an English text corpus

Power laws are **everywhere** (approximately)

- **Electrostatic** and **gravitational** forces (inverse square law)
- Distribution of **earthquake magnitudes**
- ‘80/20 rule’ in **distribution of income** (Pareto principle)
- Relationship between **body size and metabolism** (Kleiber’s law)

Power laws are **everywhere** (approximately)

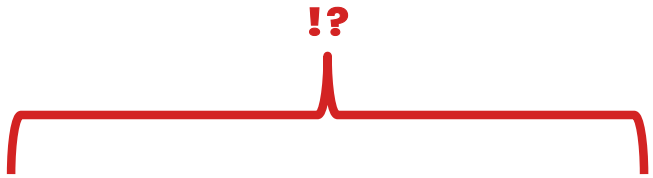
- **Word frequencies** in natural language corpora (Zipf's law)
- Degree distribution in **social networks** ('Bieber problem')
- **Participation inequality** on wikis and forum sites ('1% rule')
- Popularity of **websites** and their **content**

Why is this a problem?

Popularity hotspots: symptoms and fixes

- **Hot shards** in databases — salt keys, change schema
- **Slow load times** for certain users — look for $O(n^2)$ mistakes
- **Hot mappers** in map-only tasks — repartition randomly
- **Hot reducers** during joins and aggregations … ?

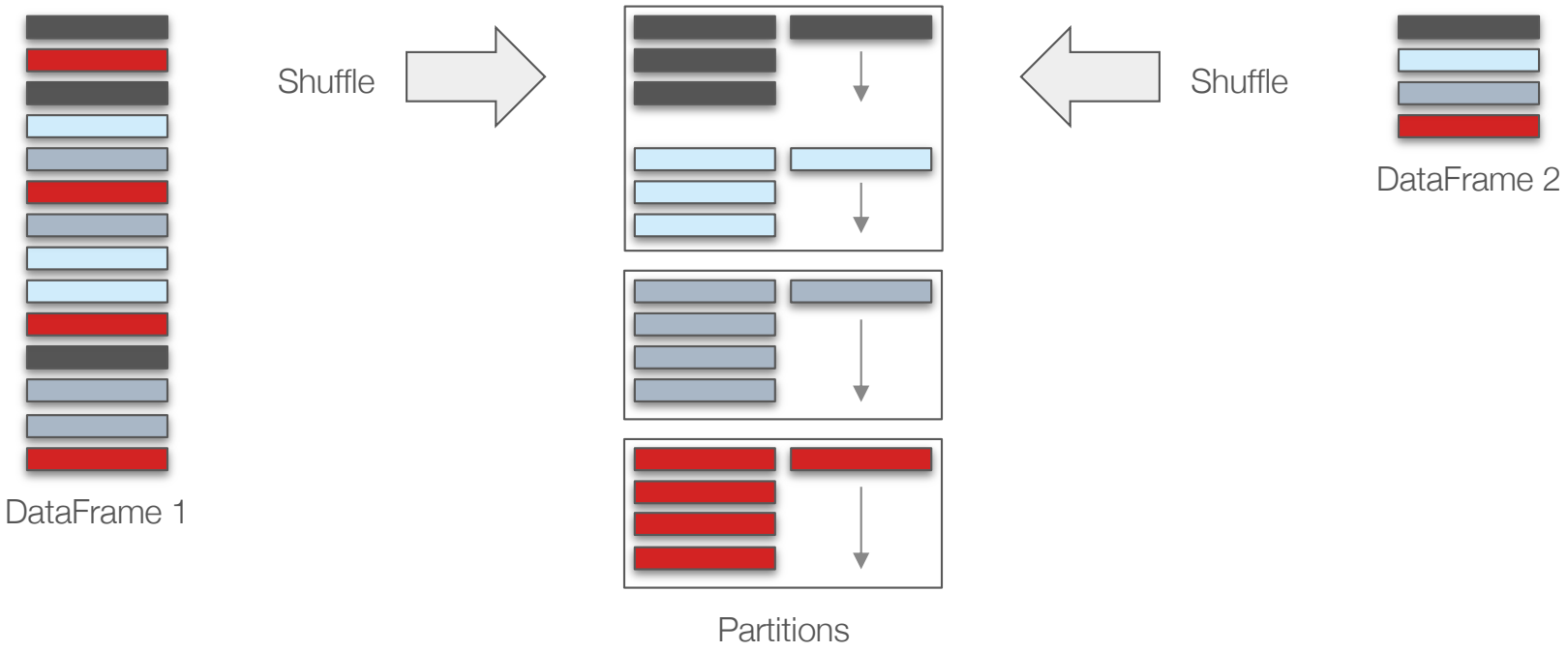
Diagnosing hot executors



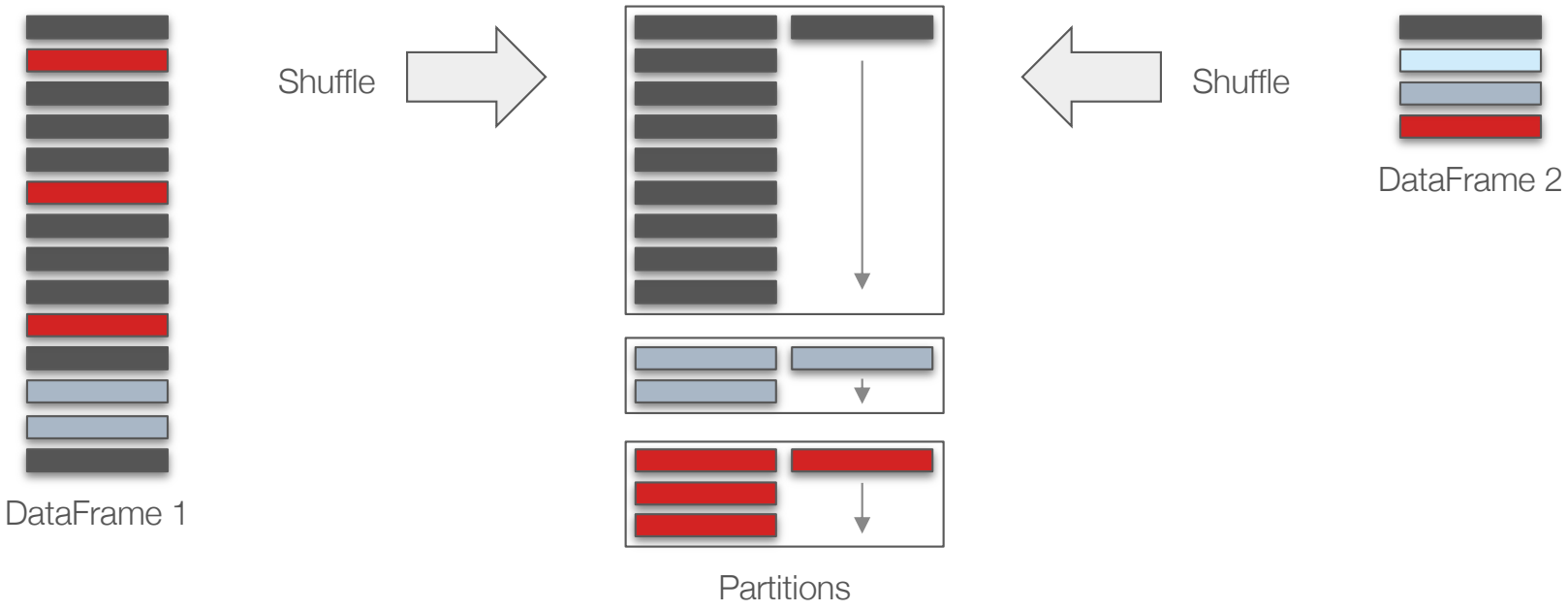
Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0 ms	3 s	5 s	10 s	55 min
Scheduler Delay	8 ms	14 ms	19 ms	42 ms	1.1 h
Task Deserialization Time	0 ms	83 ms	0.1 s	0.5 s	2 s
GC Time	0 ms	0.2 s	0.4 s	1 s	3.9 min
Result Serialization Time	0 ms	0 ms	0 ms	1 ms	3 ms
Getting Result Time	0 ms	0 ms	0 ms	0 ms	0 ms
Peak Execution Memory	0.0 B	208.0 MB	464.0 MB	1552.0 MB	3.4 GB
Shuffle Read Blocked Time	0 ms	5 ms	0.2 s	0.7 s	2 s
Shuffle Read Size / Records	3.4 MB / 709003	7.6 MB / 2353476	12.1 MB / 6699402	25.3 MB / 23890002	1740.2 MB / 17593730440
Shuffle Remote Reads	3.4 MB	7.5 MB	12.0 MB	25.3 MB	1740.2 MB
Shuffle Write Size / Records	0.0 B / 0	59.0 B / 1	59.0 B / 1	59.0 B / 1	59.0 B / 1
Shuffle spill (memory)	0.0 B	0.0 B	0.0 B	0.0 B	343.9 GB
Shuffle spill (disk)	0.0 B	0.0 B	0.0 B	0.0 B	2.4 GB

Spark joins
under the hood

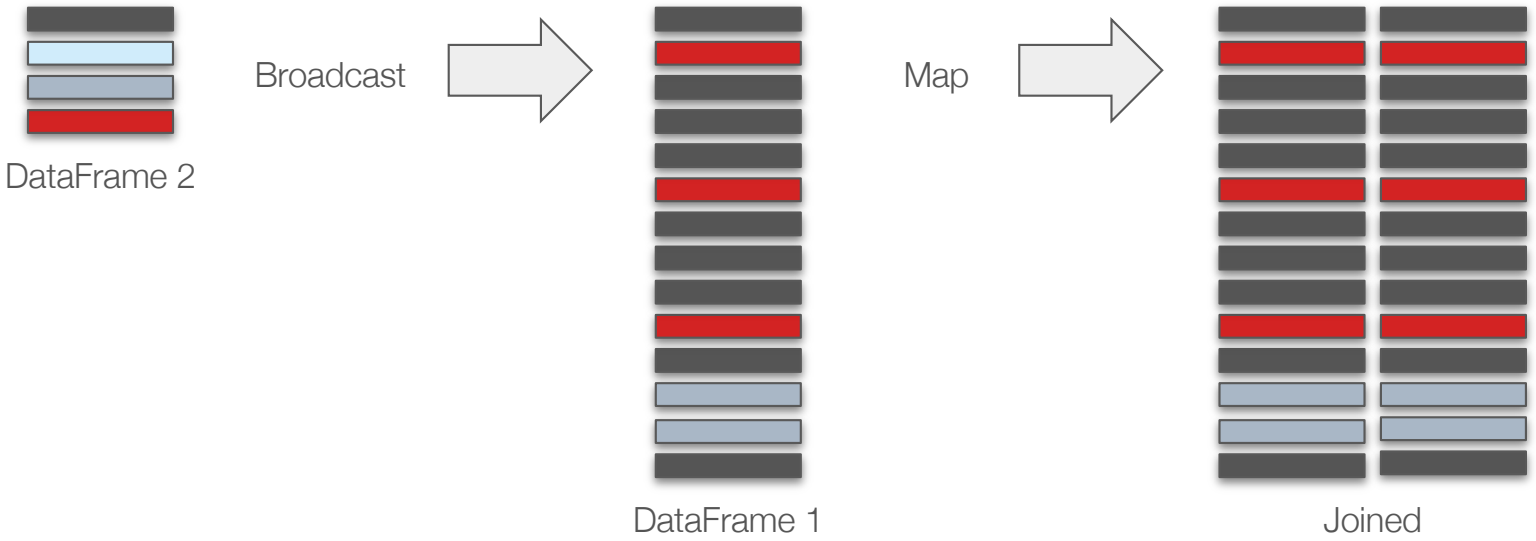
Shuffled hash join



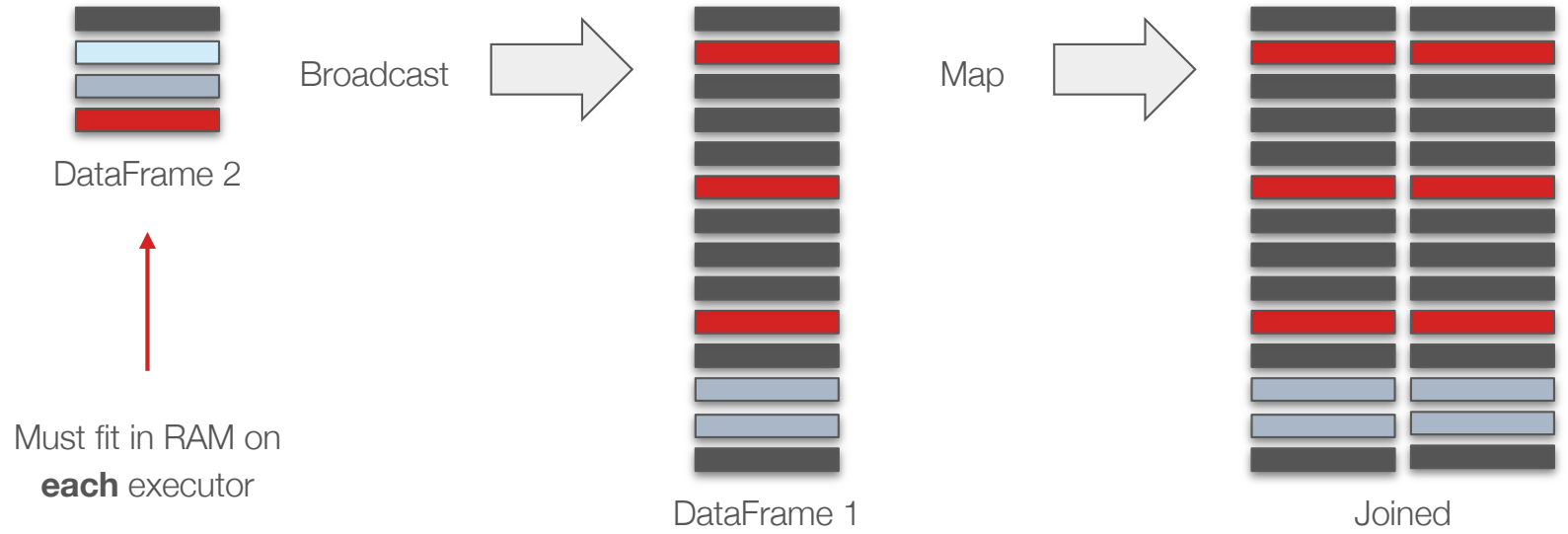
Shuffled hash join with very skewed data



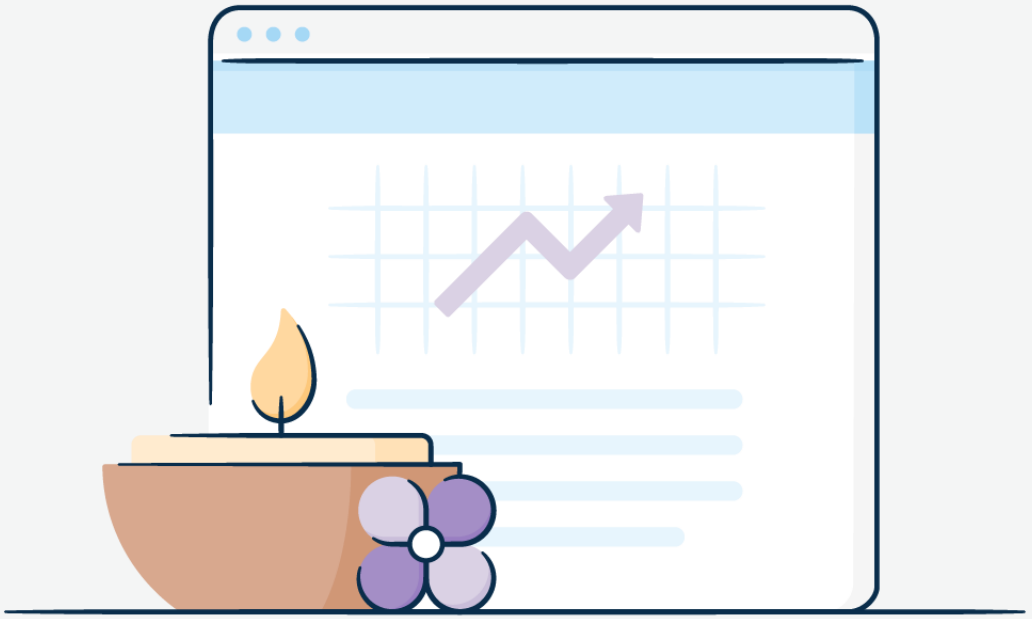
Broadcast join can help



Broadcast join can help sometimes



Joining skewed data faster

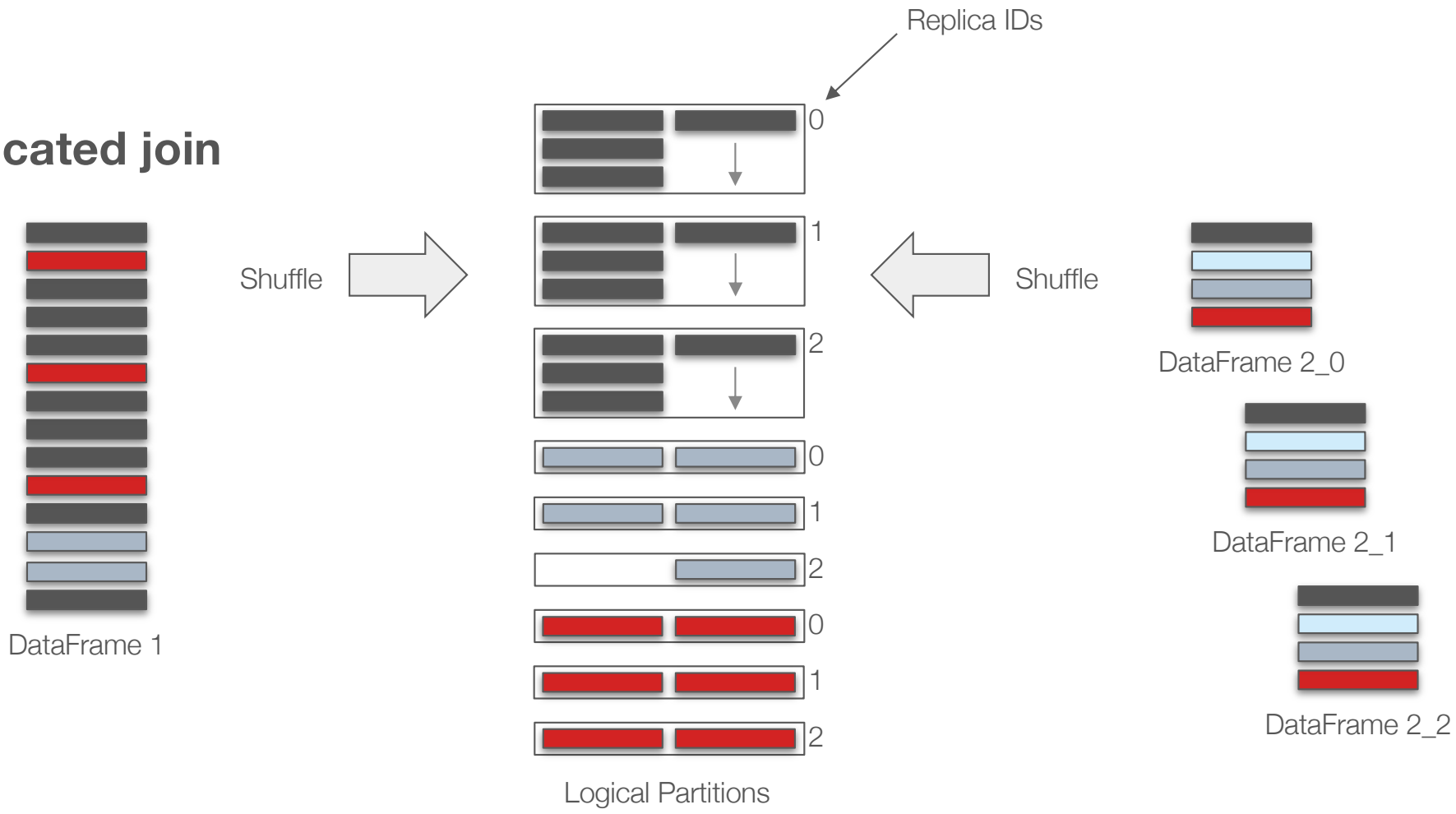


Splitting a single key across multiple tasks

- **Append random int** in [0, *R*) to each key in skewed data
- **Replicate each row** in non-skewed data, *R* times
- **Append replica ID** to original key in non-skewed data
- **Join** on this newly-generated key

R = replication factor

Replicated join

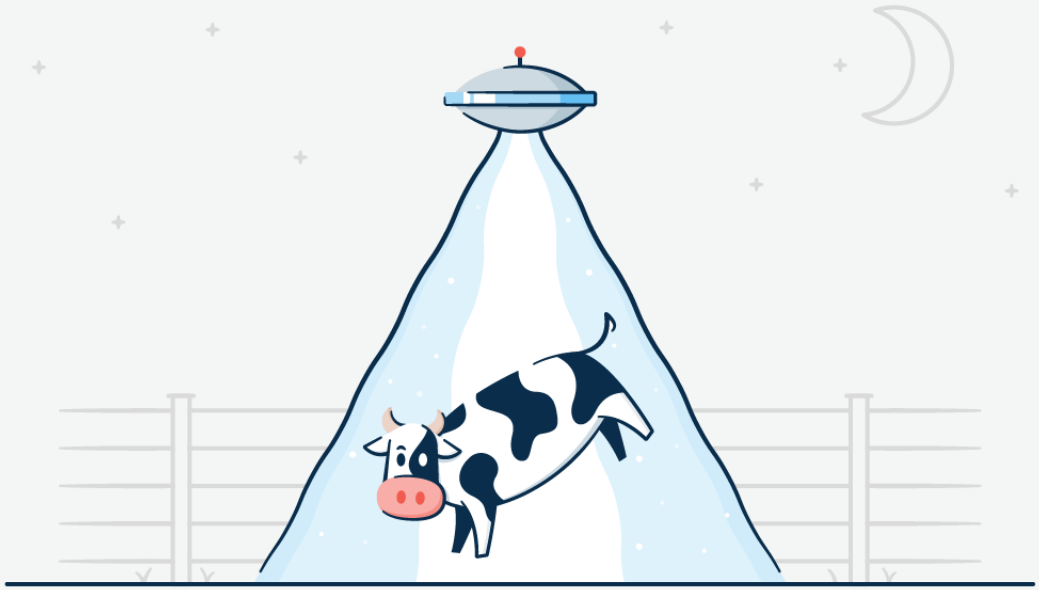


```
replication_factor = 10

# spark.range creates an 'id' column: 0 <= id < replication_factor
replication_ids = F.broadcast(
     spark.range(replication_factor).withColumnRenamed('id', 'replica_id')
)

# Replicate uniform data, one copy of each row per bucket
# composite_key looks like: 12345@3 (original_id@replica_id)
uniform_data_replicated = (
    uniform_data
    .crossJoin(replication_ids)
    .withColumn(
        'composite_key',
        F.concat('original_id', F.lit('@'), 'replica_id')
    )
)
```

```
def randint(limit):
    return F.least(
        F.floor(F.rand() * limit),
        F.lit(limit - 1),  # just to avoid unlikely edge case
    )

# Randomly assign rows in skewed data to buckets
# composite_key has same format as in uniform data
skewed_data_tagged = (
    skewed_data
    .withColumn(
        'composite_key',
        F.concat(
            'original_id',
            F.lit('@'),
            randint(replication_factor),
        )
    )
)
```

```
# Join them together on the composite key
joined = skewed_data_tagged.join(
    uniform_data_replicated,
    on='composite_key',
    how='inner',
)
```

```
# Join them together on the composite key
joined = skewed_data_tagged.join(
    uniform_data_replicated,
    on='composite_key',
    how='inner',
)
```

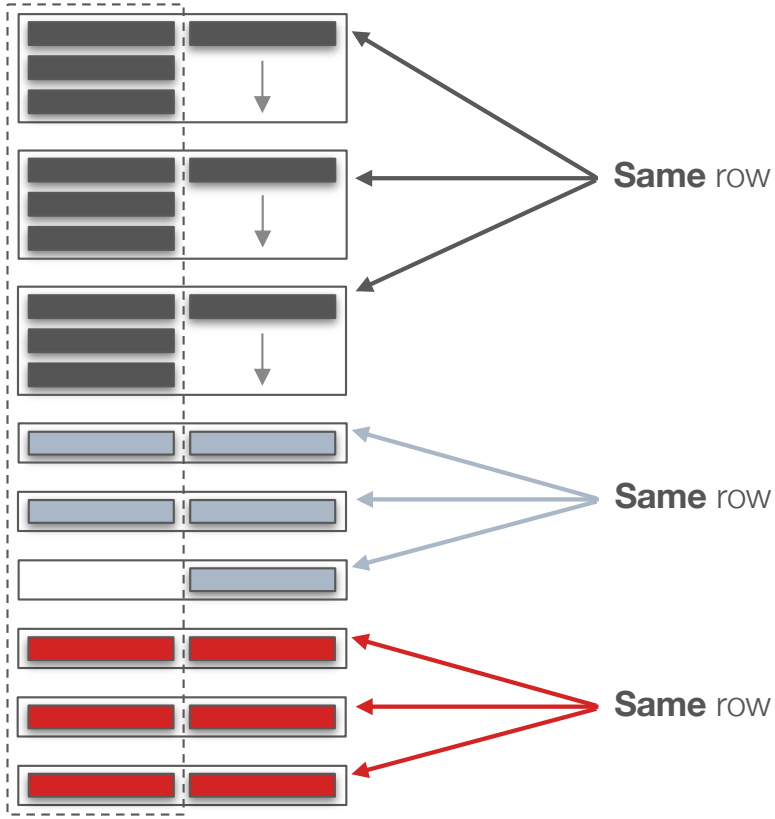


WARNING

Inner and left outer joins only

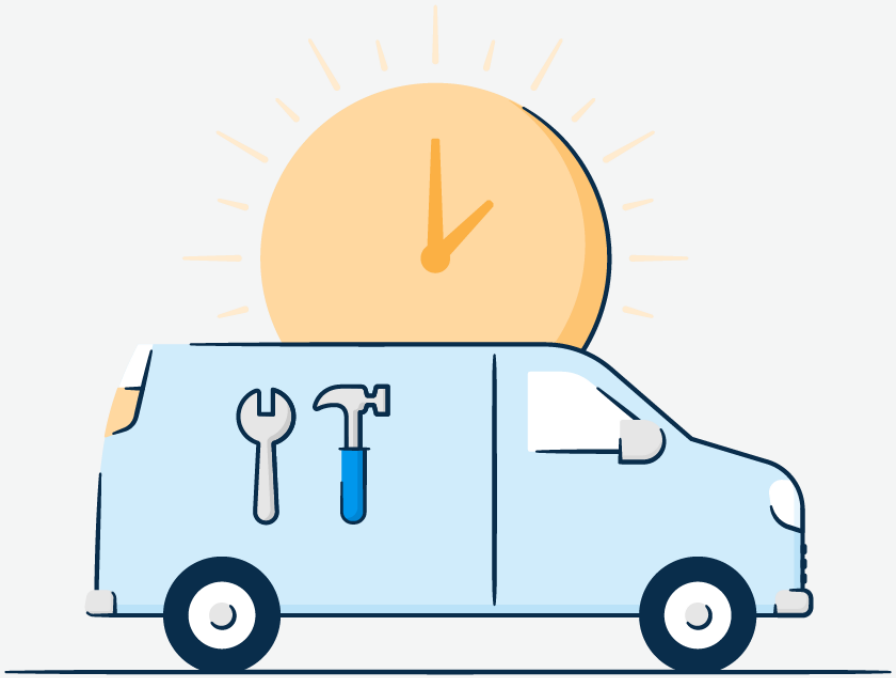
Remember duplicates…

All different rows



Experiments with synthetic data

- **100 million rows** of data with uniformly-distributed keys
- **100 billion rows** of data with Zipf-distributed keys
- Standard inner join **ran for 7+ hours** then I killed it!
- 10x replicated join **completed in 1h16m**



Can we do better?

Differential replication

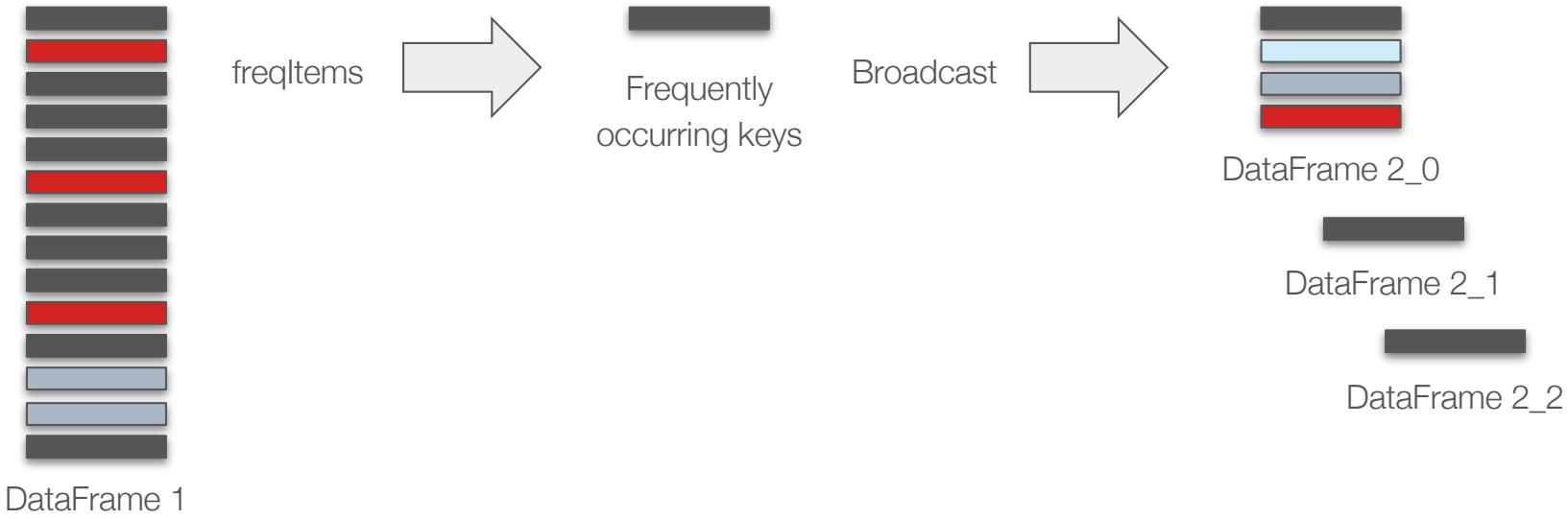
- **Very common keys** should be replicated many times
- **Rare keys** don’t need to be replicated as much (or at all?)
- **Identify frequent keys** before replication
- Use **different replication policy** for those

```
replication_factor_high = 50
replication_high = F.broadcast(
    spark
    .range(replication_factor_high)
    .withColumnRenamed('id', 'replica_id')
)
replication_factor_low = 10
replication_low = … # as above

# Determine which keys are highly over-represented
top_keys = F.broadcast(
    skewed_data
    .freqItems(['original_id'], 0.0001)  # return keys with frequency > this
    .select(
        F.explode('id_freqItems').alias('id_freqItems')
    )
)
```

```
uniform_data_top_keys = (
    uniform_data
    .join(
        top_keys,
        uniform_data.original_id == top_keys.id_freqItems,
        how='inner',
    )
    .crossJoin(replication_high)
    .withColumn(
        'composite_key',
        F.concat('original_id', F.lit('@'), 'replica_id')
    )
)
```

```
uniform_data_rest = (
    uniform_data
    .join(
        top_keys,
        uniform_data.original_id == top_keys.id_freqItems,
        how='leftanti',
    )
    .crossJoin(replication_low)
    .withColumn(
        'composite_key',
        F.concat('original_id', F.lit('@'), 'replica_id')
    )
)

uniform_data_replicated = uniform_data_top_keys.union(uniform_data_rest)
```

Replicate very frequent keys more often

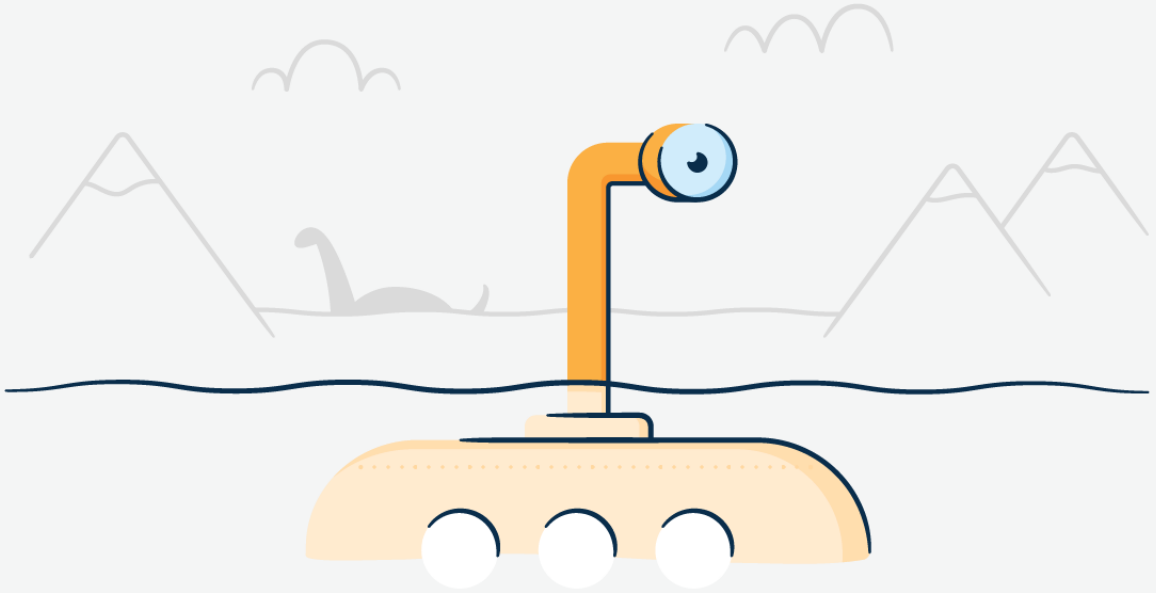


```
skewed_data_tagged = (
    skewed_data
    .join(
        top_keys,
        skewed_data.id == top_keys.id_freqItems,
        how='left',
    )
    .withColumn(
        'replica_id',
        F.when(
            F.isnull(F.col('id_freqItems')), randint(replication_factor_low),
        )
        .otherwise(randint(replication_factor_high))
    )
    .withColumn(
        'composite_key',
        F.concat('original_id', F.lit('@'), 'replica_id')
    )
)
```

Experiments with synthetic data

- **100 million rows** of data with uniformly-distributed keys
- **100 billion rows** of data with Zipf-distributed keys
- 10x replicated join **completed in 1h16m**
- 10x/50x differential replication **completed in under 1h**

Other ideas worth mentioning



Partial broadcasting

- Identify **very common keys** in skewed data
- Select these rows from uniform data and **broadcast join**
- **Rare keys** are joined the traditional way (without replication)
- **Union** the resulting joined DataFrames

Dynamic replication

- Get approximate frequency for **every key** in skewed data
- Replicate uniform data **proportional to key frequency**
- **Intuition:** Sliding scale between rarest and most common
- Can be hard to make this work in practice!

Double-sided skew

- **Append** random replica_id_that for other side, in [0, R_{that})
- **Replicate** each row R_{this} times, and append replica_id_this
- Composite key on **left**: id, replica_id_that, replica_id_this
- Composite key on **right**: id, replica_id_this, replica_id_that

Summing up

Checklist

- Is the problem just **outliers**? Can you safely ignore them?
- Try **broadcast join** if possible
- **Look at your data** to get an idea of the distribution
- **Start simple** (fixed replication factor) then iterate if necessary

Credits

- **Warren & Karau**, *High Performance Spark* (O'Reilly 2017)
- Blog by **Maria Restre**: datarus.wordpress.com
- **Scalding** developers: github.com/twitter/scalding
- Spark, ML, and distributed systems community @ **Yelp**

Thanks!

@andrew_clegg