
BEAM
SUMMIT

Parallelizing Skewed Hbase regions using Splittable Dofn

Prathap Reddy
Google

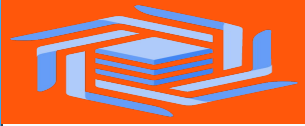
BEAM SUMMIT
NYC 2023

Agenda

- HBase and BigTable Overview
- HBase Snapshot Storage Structure
- Import Snapshots Pipeline
- Challenges & Resolutions

HBase

- Open Source Distributed Scalable Big Data Store
- Random read/write access patterns
- Automatic sharding of tables across regions
- Server side processing using Coprocessors



Bigtable

- Fully managed by Google
- High availability and automatic replication
- Auto Scaling based on application traffic
- Enterprise grade security and control

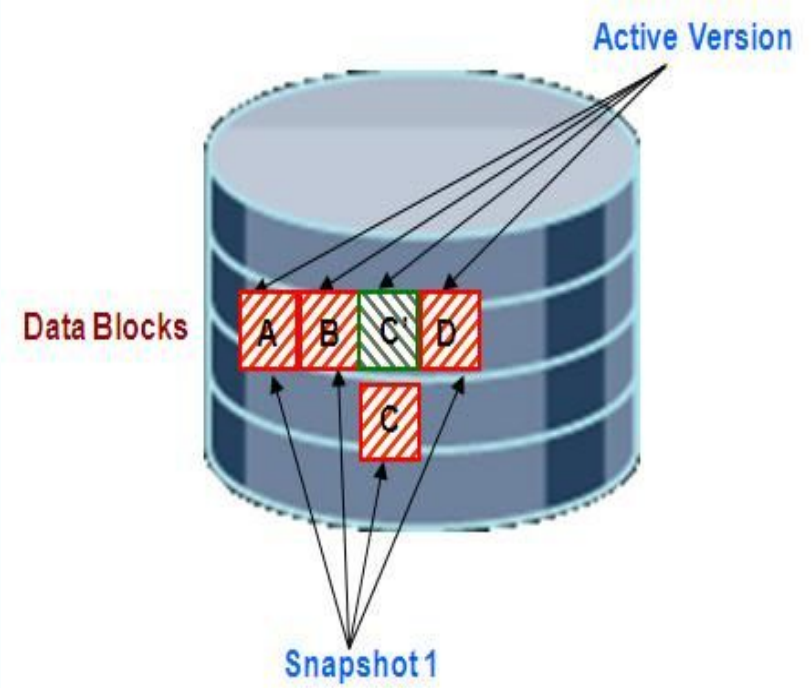
Hbase Snapshots

- Representation of table at point in time
- Zero Data Copying
- Minimal impact on region servers
- Creating Snapshot

```
hbase> snapshot 'tableName', 'snapshotName'
```

- Export Snapshot to Google Cloud Storage

```
hbase> hbase \
org.apache.hadoop.hbase.snapshot.ExportSnapshot \
-snapshot $SNAPSHOT_NAME \
-copy-to  $BUCKET_NAME$SNAPSHOT_EXPORT_PATH/data \
-mappers  $NUM_MAPPERS
```



Hbase Storage Structure

```
Table                   (HBase table)
    Region              (Regions for the table)
        Store           (Store per ColumnFamily for each Region for the table)
            MemStore    (MemStore for each Store for each Region for the table)
            StoreFile   (StoreFiles for each Store for each Region for the table)
                Block   (Blocks within a StoreFile within a Store for each Region for the table)
```

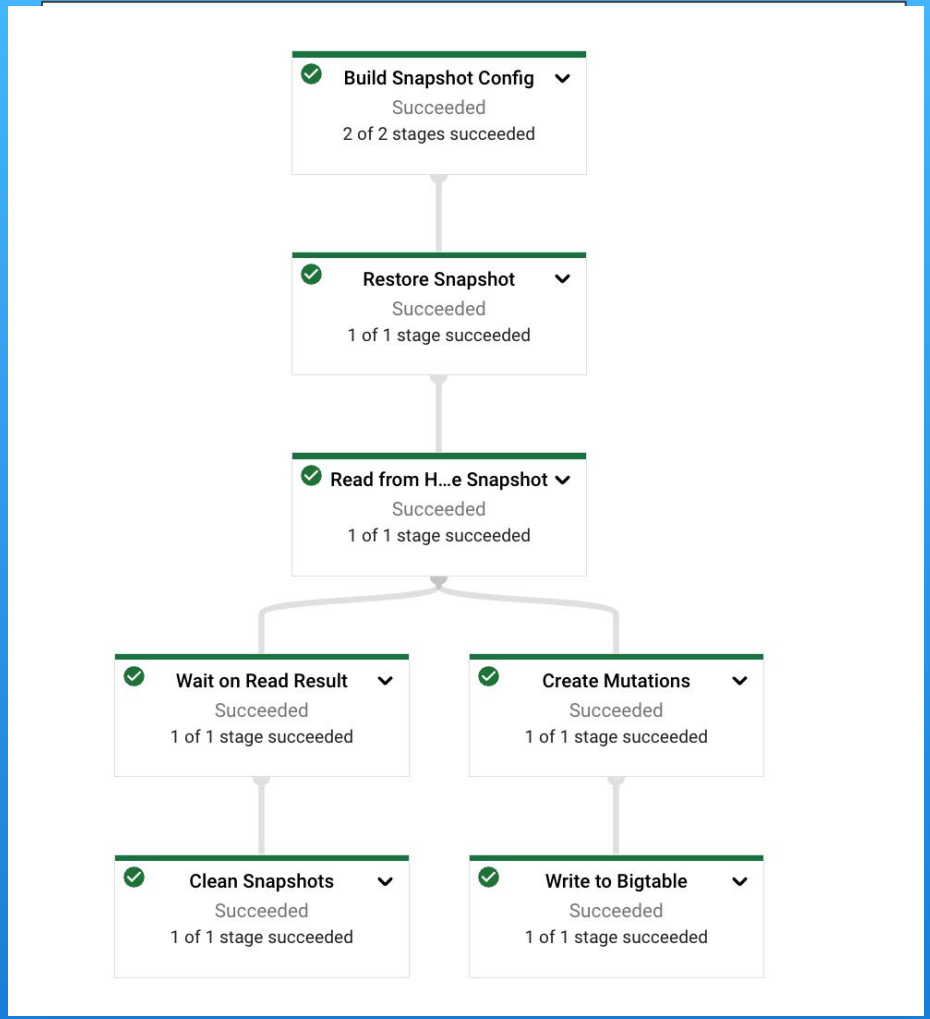
* *Region represents a key range (startKey - endKey) and may live on a different region server*

* *Store Files are also known as Hfiles*

Importing to BigTable (v1)

- Build Snapshot Config
- Read Snapshot (HadoopFormatIO)
- Create Mutation
- Write to Bigtable

* Pipeline Source



Challenges

- Skewed regions
- Single Table Snapshots

Importing to BigTable (v2)

- Read multiple Snapshot Configs
- List Regions
- Read Region Splits (in parallel)
- Create mutation
- Write to multiple tables in Bigtable

* Snapshot config provides snapshot name, source path and target table name

Splittable Dofn

- Powerful abstraction with support to split each element of work

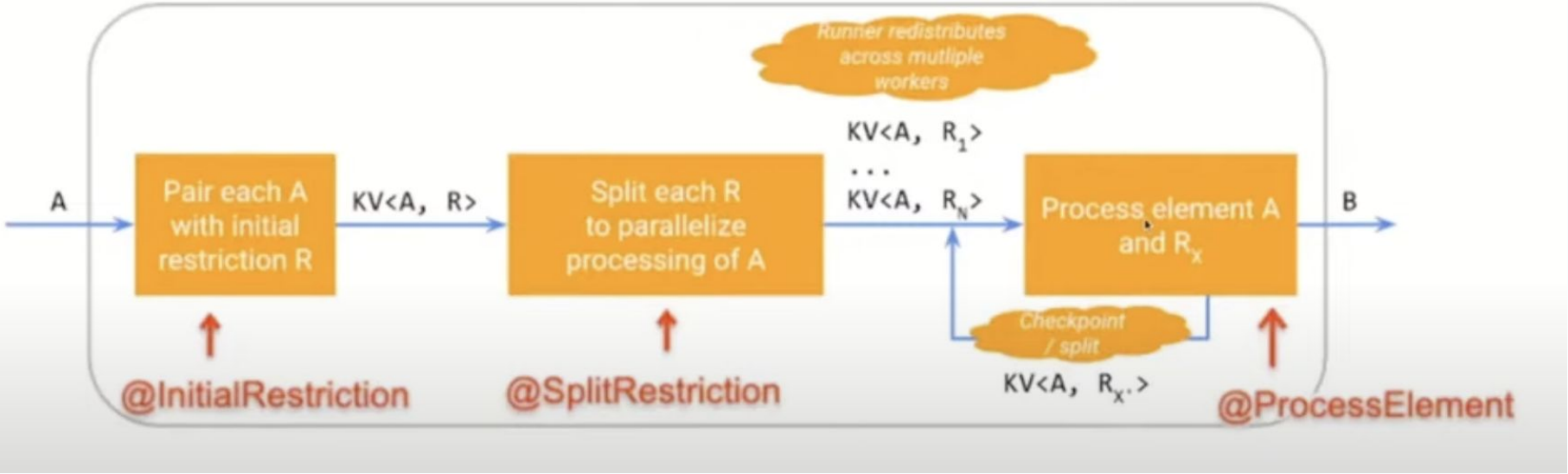
  (element, restriction) -> (element,restriction_1) + (element, restriction_2)

- Dynamic rebalancing to avoid stragglers

Splittable Dofn

- Restriction represents a portion of work (e.g: OffsetRange, ByteKeyRange)
- Similar Syntax as DoFn with an additional **RestrictionTracker** parameter to ***@ProcessElement*** method
- ***@GetInitialRestriction*** - *Represents the complete work for a given element*
- ***@SplitRestriction*** (Optional) - Supports pre-splitting initial restriction

Execution of Splittable Dofn



Splittable Dofn

```
@GetInitialRestriction
public ByteKeyRange getInitialRange(@Element RegionConfig regionConfig) {
  return ByteKeyRange.of(
      ByteKey.copyFrom(regionConfig.getRegionInfo().getStartKey()),
      ByteKey.copyFrom(regionConfig.getRegionInfo().getEndKey()));
}
```

Splittable Dofn

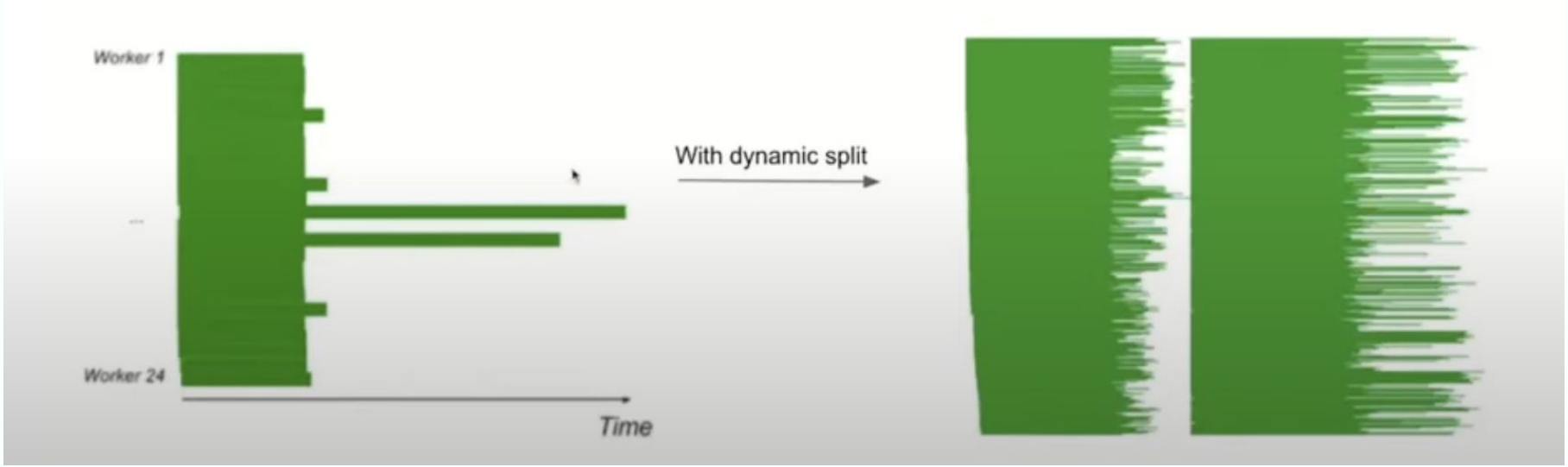
```
@SplitRestriction
public void splitRestriction(@Element RegionConfig regionConfig,
                             @Restriction ByteKeyRange range,
                             OutputReceiver<ByteKeyRange> outputReceiver) {
  int numSplits = (int) Math.ceil((double) regionConfig.getRegionSize() / BYTES_PER_SPLIT);
  if (numSplits > 1) {
    RegionSplitter.UniformSplit uniformSplit = new RegionSplitter.UniformSplit();
    byte[][] splits =
            uniformSplit.split(
                    range.getStartKey().getBytes(),
                    range.getEndKey().getBytes(),
                    getSplits(regionConfig.getRegionSize()),
                     inclusive: true);
    IntStream.range(0, splits.length - 1).forEach((int i) ->
      outputReceiver.output(
              ByteKeyRange.of(ByteKey.copyFrom(splits[i]), ByteKey.copyFrom(splits[i + 1]))));
  } else {
    outputReceiver.output(range);
  }
}
```

Splittable Dofn

```
@ProcessElement
public void processElement(
    @Element RegionConfig regionConfig,
    OutputReceiver<KV<SnapshotConfig, Result>> outputReceiver,
    RestrictionTracker<ByteKeyRange, ByteKey> tracker)
    throws Exception {
  try (ResultScanner scanner = newScanner(regionConfig, tracker.currentRestriction())) {
    for (Result result : scanner) {
      if (tracker.tryClaim(ByteKey.copyFrom(result.getRow()))) {
        outputReceiver.output(KV.of(regionConfig.getSnapshotConfig(), result));
      } else {
        break;
      }
    }
  }
  tracker.tryClaim(ByteKey.EMPTY);
}
```

Dynamic Splitting

- Splits current processing element into primary and residual parts
- Runners schedules residual part onto another instance



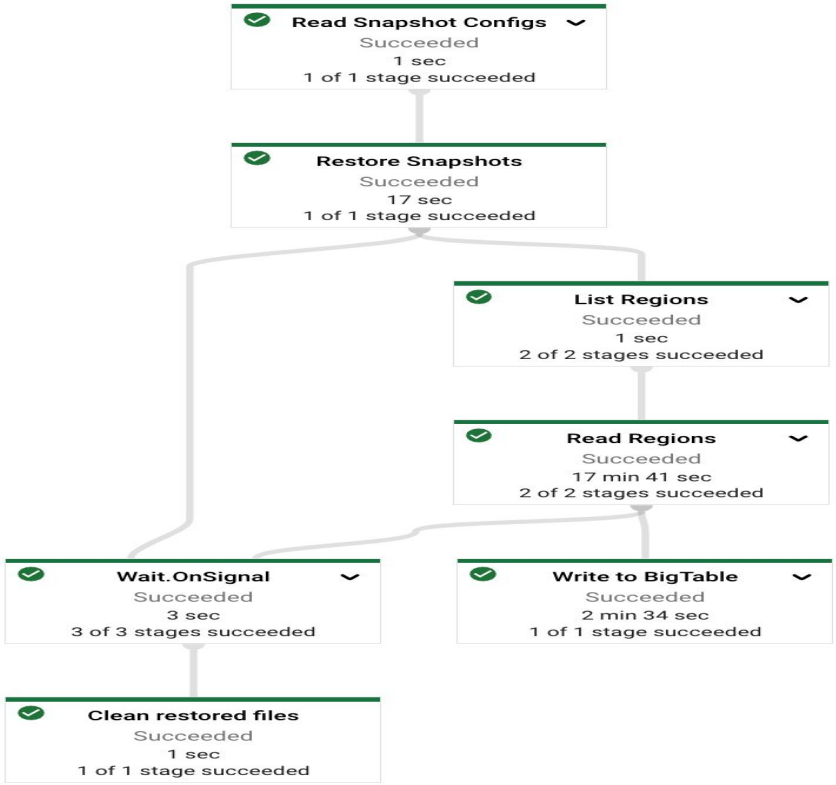
Dynamic Splitting

```
public class HbaseRegionSplitTracker extends RestrictionTracker<ByteKeyRange, ByteKey>
    implements RestrictionTracker.HasProgress {

 public HbaseRegionSplitTracker(boolean enableDynamicSplitting) {
   this.enableDynamicSplitting = enableDynamicSplitting;
 }

  public SplitResult<ByteKeyRange> trySplit(double fractionOfRemainder) {
    return enableDynamicSplitting ? this.byteKeyRangeTracker.trySplit(fractionOfRemainder) : null;
  }
}
```

Pipeline Graph



Benchmark Tests

- Snapshot Datasets
  - 104 GB with 19 regions (6 regions of 3.5 GB in size and remaining 13 regions are approximately 7 GB)
  - 875 GB with 14 regions (Mixed region sizes varying from 30GB to 98 GB)
- Enabled and Disabled Dynamic Splitting
- 10 - 30% improvements in Job Duration with reduced VCPU Consumption

* Beyond Initial splits enabling further splitting didn't yield significant differences

Prathap Reddy

QUESTIONS?


@prathapreddy017


https://github.com/prathapreddy123


https://www.linkedin.com/in/prathapparvathareddy