Sort Merge Bucket support in Beam

https://issues.apache.org/jira/browse/BEAM-6766

Claire McGinty (clairem@spotify.com), Neville Li (neville@spotify.com)

Overview

SMB is an optimization for joining on predetermined keys across datasets. By grouping and sorting keys in upstream (producer) jobs and writing to bucketed files, downstream (consumer) jobs can join data by reading and merging matching buckets, and thus avoiding shuffle.

One of the most common user operations in Map/Reduce jobs is the join. In Beam jobs, this is commonly implemented as a union of 2+ datasets followed by a `GroupByKey` operation on the desired join key (`CoGroupByKey` abstracts both of these operations). For large datasets with many keys, the shuffle incurred by the CoGBK operation can be the most expensive step of the job in terms of wall time and resource consumption, especially for hot datasets read by many consumer jobs.

The shuffle step is unavoidable if we have no knowledge of the underlying data format and can only access it as an unordered PCollection. However, for file-based input that are read in sequentially, if we know the most common join key in advance, upstream jobs can write records in a predictable way that allows downstream jobs to avoid the CoGBK shuffle phase entirely. This is the Sort Merge Bucket algorithm[1]: instead of the sink collection being randomly sharded and written to FS, we can (1) assign each record to a specific bucket based on a hash of its join key, (2) within each bucket, write records sorted by that key, and (3) write each bucket to a separate file whose filename corresponds to the bucket ID. This allows us to perform a Source+CoGBK-like operation just by sequentially reading individual files with the same bucket ID and merge sort records, without actually inccuring a CoGBK/shuffle.

We foresee this being especially helpful for user event data, since user ID is a very common join key and is likely to be well-distributed without significant skew.

This proposal is for a SMB Sink and Source, where the Sink writes a single PCollection to bucketed, sorted files, and the Source reads in 2+ directories and returns a single PCollection representing the cogrouped inputs.

SMB is the ideal solution for producer and consumer of common data sets. Imagine the following scenario:

- A producer pipeline writes 2 datasets about user behavior daily, say user accounts and user events
- 100 consumer pipelines read them daily and join by user ID, which incurs the same shuffle 100 times
- With SMB the producer writes both datasets in files bucketed & sorted by user ID
- Each consumer reads pairs of files within the same bucket and merge sort records of the same user ID

[1] SMB is currently already supported in Hive and Spark SQL.

Table of Contents

Goals

List a set of goals or requirements that this feature must solve.

- `smb` Gradle module within `beam/sdks/java/extensions`.
- Sink for writing data as sorted buckets, and corresponding metadata (i.e. # buckets, join key, hash function).
- Source that performs a join across 2+ inputs *only if* they are SMB-compatible (i.e., same join key, hashing function) with significant reduction in wall time and compute resources compared to union+GBK implementation.
- Common file-based IO types, such as Avro, must be supported out of the box, with flexibility for users to implement their own IO.
- This implementation should be something users can easily swap in for their existing sinks and sources with no negative performance implications. I.e., SMB sinks should be well-distributed enough that regular union+GBK can still be used.

Non-goals

State any explicit non-goals, along with reasons why they are not considered for this design.

- This proposal is limited to file-based IOs.
- This implementation is limited to the Java SDK.

Current Status

We have a working implementation currently in PR, shipped with benchmarks on artificial data. We plan to split the PR according to each of the proposal items below. In internal tests we've seen 3-5x speedup with significantly less PD and vCPU usage, as measured by Dataflow.

More modular PRs:

- SMB Metadata apache/beam#8823
- FileOperations apache/beam#8824

Proposal

SMB Metadata

(currently in PR)

In order to perform an SMB join, we need information on how the inputs were written to verify that they're compatible. For example, if two datasets are sorted on a different key, we should be able to fail the job at launch time. Additionally, we need to know how to extract the key field from the PCollection element type, in order to compute key groups.

We propose that, for a `PCollection<V>` with key `K`, the SMB sink will additionally write a `metadata.json` file to the output directory with the following fields:

Field	Type	Description	Restrictions
version	`int`	Major version of SMB implementation[2]	
numBuckets	`Int`	Number of output buckets	power of 2[3]
numShards	`int`	Number of shards in each bucket[4]	
keyClass	`Class<K>`	Class of extracted key field, required to encode/decode to bytes	
hashType	Enum representing `com.google.common.hash.HashFunction`	Hash function to apply to key bytes, which we will then modulo `numBuckets` to assign each record a bucket ID	

[2] In case we need to make breaking changes to output format of SMB Sink
[3] Power of 2 so that inputs of different bucket size can be joined as long as they share the same key semantic/encoding and hashing function
[4] Randomly assigned to handle potential hotkey situation

extractKeyFn	? Depends on impl	Implementation of this is specific to IO type. For example, for Avro IO, metadata might just include the key field name, which is enough information to extract key from a record	

Each IO type would extend a base `BucketMetadata` class with any information specifically required to reconstruct a function `T -> K` that maps elements to their keys.

File Operations

(currently in PR)

We would like to offer pre-written SMB implementations for file IOs like Avro, JSON, or TensorFlow TFRecord. We can't re-use existing sinks/sources and risk runner-determined shuffling/sharding after we've done our own bucketing/sorting. Furthermore, most of the existing Beam code that perform reads/writes at a per-record granularity are package private and can't be re-used. We plan to define an abstract class `FileOperations<V>`, where `V` is the `PCollection` element type, that custom IO types must implement:

```
public abstract class FileOperations<V> implements Serializable {
	public abstract static class Reader<V> implements Serializable {
		public abstract void prepareRead(ReadableByteChannel channel) throws Exception;

		public abstract V read() throws Exception;

		public abstract void finishRead() throws Exception;
	}

	public abstract static class Writer<V> implements Serializable {
		public abstract String getMimeType();

		public abstract void prepareWrite(WritableByteChannel channel) throws Exception;

		public abstract void write(V value) throws Exception;
	}
}
```

SMB Sink

(current code branch)

We have modeled the SMB sink after Beam's `WriteFiles` transform. It extends `PTransform<PCollection<V>, WriteResult>` where `WriteResult` is a `POutput` containing information on final file destinations. This composite PTransform has 4 steps:

1) Extract keys of type `K` from every element `V`. Encode each `K` to a byte array and apply the hashing function to assign each `V` an integer bucket ID and a random shard ID.
2) Group by bucket/shard ID.

3) Apply a SortValues[5] transform to sort each bucket by the key bytes of each record.
4) Write buckets and metadata to FS using the supplied `FileOperations` writer:
    a) Write to timestamped temp directory.
    b) Move temp files to final output directory. Cleanup if necessary.

SMB Source

(current code branch)

SMB Source should be modeled as a `PTransform<PBegin, PCollection<KV<K, CoGbkResult>>>`, where `K` and N is supplied by the user and `CoGbkResult` encapsulates value groups from individual inputs. It has 4 steps, where N is the number of buckets and BucketedInput represents a single join source.

1) For each supplied input directory, read its `metadata.json` file to verify source compatibility.
2) Emit N records of the format `KV<N, List<BucketedInput>>`, where `BucketedInput` contains a `TupleTag` identifier and sufficient information to read the file(s) corresponding to that bucket.
3) Apply a Reshuffle.ViaRandomKey to randomly redistribute these pairs. Each represents a non-parallelizable unit of work, i.e. read and merge sort files of bucket N across all `BucketedInput`s.
4) For each of these KV pairs, use `FileOperations` readers to begin sequentially reading from all files corresponding to that bucket, comparing each record's key bytes to confirm equality. Construct groups of type `KV<K, CoGbkResult>>` where `CoGbkResult` encapsulates one `TupleTag<V>` and `Iterable<V>` pair for each individual `BucketedInput`.

Appendix

Related Work

1. *Sort Merge Buckets: Optimizing Repeated Skewed Joins in Dataflow* - Master thesis work by Andrea Nardelli. The paper has a description of algorithms, and experiments applying this technique to various datasets. That work was hosted at Spotify.
2. Hive Bucketing in Apache Spark - slides, design doc
3. Managing "Exploding" Big Data - LinkedIn blog post about join optimization

Benchmark

We performed a simple benchmark with fairly uniformly distributed # of elements per key, SMB join is ~1.66x faster based on wall clock time (8m57s vs 14m50s), and uses about half vCPU hours (5.915 vs 11945). More in depth benchmarks can be found in Andra Nardelli's master thesis in the related work section. According to the paper, simple SMB join without skewed-adjusted algorithm is ~3-4x faster (~8m vs 27-32m) and uses a quarter or less vCPU hours (~12 vs 52-63).

[5] Using the default external buffered option that spills to disk

Unsolved Issues

A master list of Todos can be found on the Beam PR.

- Sharding per bucket, in case of extreme skew -- what is the optimal way to implement this? A few options include:
    - (Currently implemented) User specifies constant # of shards at write time; each bucket is randomly split into that many shards and sorting is local to that bucket-and-shard (easy to implement, but requires users to make a guess as to optimal #, with no finer-grained control per bucket).
    - User specifies target byte size per file and writer opens a new shard when that byte size is exceeded. Ordering is global per bucket across all sharded files (easy to reason about, at the cost of non-parellizable per-bucket writes). This is a breaking change from the current implementation.
- Automatically suggest values for # buckets and # shards so users don't have to guess.
- [**DONE**] Support nullable keys, with the assumption that most users won't be interested in including them in the join results.
    - Our proposal is to store all null-keyed records in a specially named bucket (i.e. "bucket-null-keys-shard-###-of-###"), separate from the regular bucketed records which are assigned a bucket number in the range [0, numBuckets). The default behavior of the join is to just skip reading those buckets unless explicitly specified by the user. We think this idea is relatively simple to implement as it only requires a few code changes to redirect null keyed records to a different bucket ID.
- Abstract away metadata storage. Right now it's hard coded to write to `output/metadata.json` but we can decouple it with an interface and allow pluggable storage.
- Replace `GroupByKey` → `SortValues` transforms in SMB sink with `GroupByKeyAndSortValues` (discussion) that leverages runner native secondary sort.
- [**DONE**] Reuse `CoGbkResult` for `SMBCoGbkResult` in SMB source. To do this we need to open up `CoGbkResult` construction and pass in pre-constructed `List<Iterable<?>> valueMap` and `CoGbkResultSchema schema`. We might also have to deal with the laziness & memorization of `Reiterator` (see next 2 points).
- [**WON'T FIX**: see discussion here re: not building on top of `FileBased{Sink,Source}`. Instead we'll re-use `FileIO.Sink` and `ReadableFile`.] Reuse existing `FileBased{Source,Sink}` logic for file IO. The sink abstraction should be fairly simple since it's just sequential writes. The source one is more complex since we have to deal with offset-based partial reads (see next point lazy `Iterable`).
- Make the `Iterable` in `CoGbkResult` lazy instead of materializing everything in memory. This is non-trivial and better done by refactoring and reusing offset-based IO logic in existing `FileBasedSource`. The `Iterable`s need to memorize file path, start offset, and key to re-iterate items. The other issue is we don't know the end offset of a group key until after exhausting all elements of the previous group, unless if we persist start offset of every key in the Sink?
- Merge inputs of more than one date/hour partition, e.g. multiple daily/hourly buckets with a single metadata, for cases like backfill. Probably a premature optimization and better handled with 1-to-1 joins followed by union.