Scaling Near-Duplicate Detection using Apache Beam

Pablo Rodriguez Defino - Cloud Consultant at Google

Deduplication at Scale

- ~30% of public web scrapes are near-duplicates, mirrors, and boilerplate.
- Redundant tokens cause model overfitting and inflate GPU compute costs.
- Separate pipelines for batch cleaning (e.g., Spark) and streaming ingest (e.g., Flink) cause signature drift due to mismatched normalizations.
- Apache Beam unifies orchestration to guarantee byte-for-byte signature parity across both execution modes.

Two-Stage Architecture

- Stage 1 (Syntactic phase): Min-LSH drops 90%+ of exact matches and boilerplate.

  ```
  Read(BQ) → ParDo(Shingle) → ParDo(JAX MinHash) → Reshuffle →
  ParDo(Bigtable State Lookup) → Write(GCS).
  ```

- Stage 2 (Semantic phase): Identifies conceptual clones via 512-dimension BERT-Small embeddings.

  ```
  Read(GCS) → RunInference(BERT) → ParDo(JAX Intra-Batch Dedup) → Write(BQ
  Staging) → ParDo(Trigger BQ Vector Merge).
  ```

Phase 1: Data Locality vs. Shuffling

- Exploding documents into shingles via `FlatMap` and shuffling them across the network creates a massive data explosion (e.g., 20,000 documents yield 10.2 million shuffled elements).
- **Local Processing:** Compute shingles and execute the full JAX MinHash signature matrix multiplication locally on the worker.
- **Network Impact:** Only the final lightweight signatures (1,024 bytes) are transmitted, reducing shuffled volume by +90%.

Phase 1: Leveraging JAX-Beam Features

- Computing hundreds of MinHash permutations per document bottlenecks pure Python threads.
- Min-LSH is refactored into matrix multiplications using JAX.
- `DoFn.setup()` handles XLA compilation; `DoFn.process()` feeds tensors directly into the binary kernel.

```
def setup(self):
     import jax
     import jax.numpy as jnp

     # JIT compile the entire vmapped universal hash matrix operation once during startup
     self._jit_hash = jax.jit(jax_batch_hash, static_argnums=(4,))
```

Stage 1: Challenges and Considerations

- JAX recompiles kernels on shape changes. Inputs must be padded to discrete static buckets (e.g., 250, 500, 1000) to keep memory profiles flat.
- Top-level `import jax` can be removed to make images smaller for the Dataflow template launcher. Lazy imports within worker hooks isolate hardware dependencies.
- `beam.Reshuffle()` before signature generation balances workloads when BigQuery source reads return uneven splits.

Stage 1: Stateful LSH via Bigtable

- Near-duplicate detection requires global membership state
  - Local worker memory is insufficient
  - Shuffling all the data is incompatible with streaming pipelines
- Row keys use an MD5 salt prefix (`DOC#{salt}#{doc_id}`) to distribute load evenly across tablet servers and prevent hot-spotting.
- NumPy binary serialization (1,024-byte chunks) replaces JSON-like strings to minimize CPU overhead and read latencies.

```
# Raw binary arrays bypass JSON serialization locks
binary_sig = np.array(signature, dtype=np.int32).tobytes()
state_row.set_cell('cf1', b'sig', binary_sig)
```

Stage 2: Leveraging RunInference for embeddings

- BERT-Small executes in `float16` with a sequence length of 512 and quantization disabled preserving semantic resolution.
- The `RunInference` API acts as an automated micro-batcher, gathering streaming elements to maintain high GPU duty cycles without crashing workers.
- A variety of tuning knobs (such as dynamic batch boundaries and sequence constraints) are exposed as options, adapting the pipeline to different hardware VRAM profiles.

```
( p | 'Embeddings' >> get_embedding_transform(
        model_name='bert_small',
        params={
            'sequence_length': 512,
            'max_batch_size': stage2_options.max_batch_size,
            'quantization': None,
            'activation_dtype': 'float16'
        })
)
```

Stage 2: Delegated Compute vs. Pure Beam

- Joining billions of historical 512-dimension vectors against incoming streams will be problematic.
- **Option 1 - Pure Beam / SimHash:** Projects embeddings into binary keys and partitions candidates via CoGroupByKey shuffles on SimHash bands.
  - **Trade-off:** Executes pairwise cosine similarity on the worker GPU, but pulls heavy embeddings across the network causing massive shuffle volume and potentially OOMs.
- **Option 2 - Delegated Compute:** Use Beam as data ingestion, trigger and orchestrator.
  - Employs WaitOn to halt the graph until vectors are staged, then triggers a SQL `VECTOR_SEARCH` inside BigQuery.

```
# Beam using SimHash Shuffling
        grouped = {
            'in_flight': in_flight_bands,
            'historical': historical_bands,
        } | 'CoGroupByBandKey' >> beam.CoGroupByKey()
```

```
# Beam orchestrates the deduplication using specialized implementations
    staging_ranges = (
        pcoll.pipeline
        | 'Wait' >> beam.transforms.util.WaitOn(write_result.destination_load_jobid_pairs)
        | 'GenerateRanges' >> beam.ParDo(MaterializeAndGenerateRangesFn(...))
        | 'ExecuteSQLMerge' >> beam.ParDo(ExecuteChunkMergeFn(...))
    )
```

Stage 1: Performance vs. Related References

- **Reference Dataset:** 39M documents (peS2o dataset) processed on a 32-core AMD EPYC node.
  - **Baseline 1 - Traditional MinHash LSH:** ~308 docs/sec, 35.2 hours runtime, 202.4 GB index footprint. Representative of standard CPU frameworks like text-dedup and C4.
  - **Baseline 2 - LSHBloom Engine:** ~833 docs/sec, 13.0 hours runtime, 1.21 GB index footprint.
- **Our PoC:**
  - Stackoverflow dataset, ~41M entries, avg token length 200 (highly variable)
  - Sustained throughput of **~3,000 docs/sec**.
  - Incremental runs of 2, 4, 6 million documents (large duplication counts on each), taking between 20 and 35 minutes.
  - **Extrapolation:** 39M documents processed in ~3.2 hours.

Stage 2: Performance vs. Reference Baselines

- **Inference Baseline:** L4 GPU performance for 300M-parameter encoders typically averages 50–150 docs/sec. References: FineWeb and Ray Performance Guides.
- **Join Bottleneck:** Open-source pipelines using local FAISS indexes often crash or take days when comparing millions of vectors due to RAM limits.
- **Our PoC:**
  - **BERT-Small Throughput:** Achieves >1,000 docs/sec sustained. Dynamic batching saturates VRAM without triggering memory faults.
  - Setting up a BigQuery slots reservation, the pipeline completes full inference, staging, and procedural vector merges for 2M documents taking between 2 and 3 hours.
  - **E2E Throughput:** Sustains ~200 docs/sec across the complete inference and semantic join lifecycle.

Closeout: Some takeaways

- Yes, Beam is a (really) good option for LLM pre-training workloads.
- **PTransform Consolidation:** Encapsulating logic as shared Beam libraries simplifies by replacing potentially disparate batch and stream topologies, eliminating drifts at the execution layer.
- **Externalized Global State:** Decoupling the "seen" index into an external state (Bigtable=) simplifies the problem of finding global similar lexicographical content, granting streaming workers low-latency lookup access to previously resolved similarities.
- **Flexibility:** Beam transitions from moving data to orchestrating local/external/specialized compute quite easily—dispatching XLA matrix operations to local GPUs and delegating massive vector merges / joins to BigQuery while keeping clear guarantees.