The Flash Compactor: Fast, Efficient, Scalable, and Resilient Data Lake Table Upserts and Deletes with Ray

Problem Statement

BACKGROUND

Compaction is a critical job executed against tables in Amazon's internal S3-based data lake to (1) normalize the table for queries by applying all enqueued row updates, inserts, and deletes, (2) support and optimize table consumption across a wide variety of compute platforms (e.g. Spark, Redshift, Hive, Postgress, etc.), and (3) delete rows required for GDPR compliance. Conceptually, a *compaction job run* drops duplicate rows from a table given one or more *primary key* columns (to detect duplicates) and one or more *sort key* columns (to decide which duplicate to keep).

CURRENT PAIN POINTS

For Amazon, the ideal compactor should not only produce correct results but also (1) affordably compact every table in our data lake, (2) accurately predict the time required to complete any *compaction job run*, and (3) gracefully scale to handle increasingly large datasets. Our existing EMR Spark compactor has served us well for several years, but is increasingly falling short of these ideals due to the following pain points:

Cost: The current data lake growth rate and cost per byte compacted is prohibitively high to support ongoing *compaction job runs* on all of our data lake tables.

Scalability: Our largest compaction clusters are limited to compacting ~50TiB of decompressed input data. Since input datasets must fit on-cluster, stability is impacted whenever a job run crashes due to unexpected exhaustion of available cluster memory or disk space. Operationally intensive data retention campaigns must also be run by table administrators if any table partition to compact grows too large.

SLA Guarantees: There are no firm SLAs for *compaction job run* completion times. Offering *compaction job run* completion time SLAs is complicated by both a wide range of variance in *compaction job run* completion times and by unanticipated job run failures.

GOALS

Given the above pain points, we set our sights on building a new compactor that (1) significantly reduces compaction cost, (2) supports firm job run completion time SLAs via highly predictable, horizontally scalable, low-latency, and reliable compaction, and (3) ensures that arbitrarily large input datasets can be compacted on any size cluster. In this paper, we discuss these goals and present the *flash compactor* - a new compactor implementation that leverages Ray, Arrow, and S3 to meet them.

Design

COMPACTOR I/O

Input

The *flash compactor* reads 1 or more *tabular storage files (*i.e. any tabular file format readable by PyArrow) from S3 annotated with their *source table ID* (e.g. ACME.CUSTOMER_ORDERS.V1)*, partition keys* (e.g. region="North America", order_day="1995-04-03"), *partition stream version* (e.g. V2), *partition stream lifecycle state* (e.g. active, unreleased, deprecated), *partition stream position (*e.g. dataset 7 of 50), *partition stream position file number* (e.g. file 2 of 10 for stream

position 7), and *partition stream position operation type* (e.g. upsert, delete) as input.

One or more *primary keys* and *sort keys* must also be defined and shared across all input files, with an input file's *partition stream position* serving as the default *sort key*. Any custom *sort key* columns specified must have a fixed byte width, and the total concatenated size of all *sort key* column values must not exceed 32 bytes (inclusive). All input files for a single *compaction job run* must share the same *source table ID*.

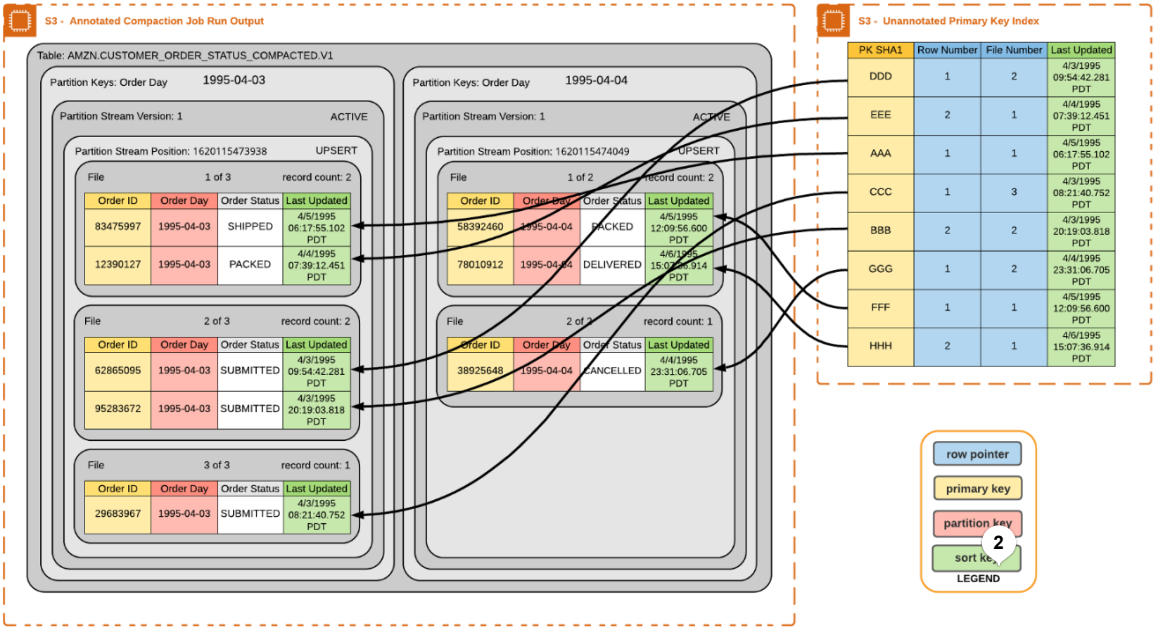
Output - Compacted Files
The compactor writes 1 or more *tabular storage files* annotated with their *compacted table ID, partition keys, compacted partition stream version,* active *partition stream lifecycle state, compacted partition stream position*, *compacted partition stream position file number*, and upsert *partition stream operation type* as output. These files contain the collective result of dropping duplicates from the input according to the input file *primary keys*, *sort keys*, and *operation types*. By default, each output file is written in the Parquet format (overridable to any tabular file format writeable by PyArrow) and contains a maximum of 4,000,000 rows (overridable to any positive integer).

Output - Primary Key Index
The compactor also writes 1 or more *primary key index* files annotated with their corresponding *compacted table ID, partition keys*, *compacted partition stream version*, *hash bucket*, and *primary key index version* as output. Each row in the *primary key index* contains at least 32 bytes (with no custom *sort keys*) and at most 64 bytes (with custom *sort keys*):

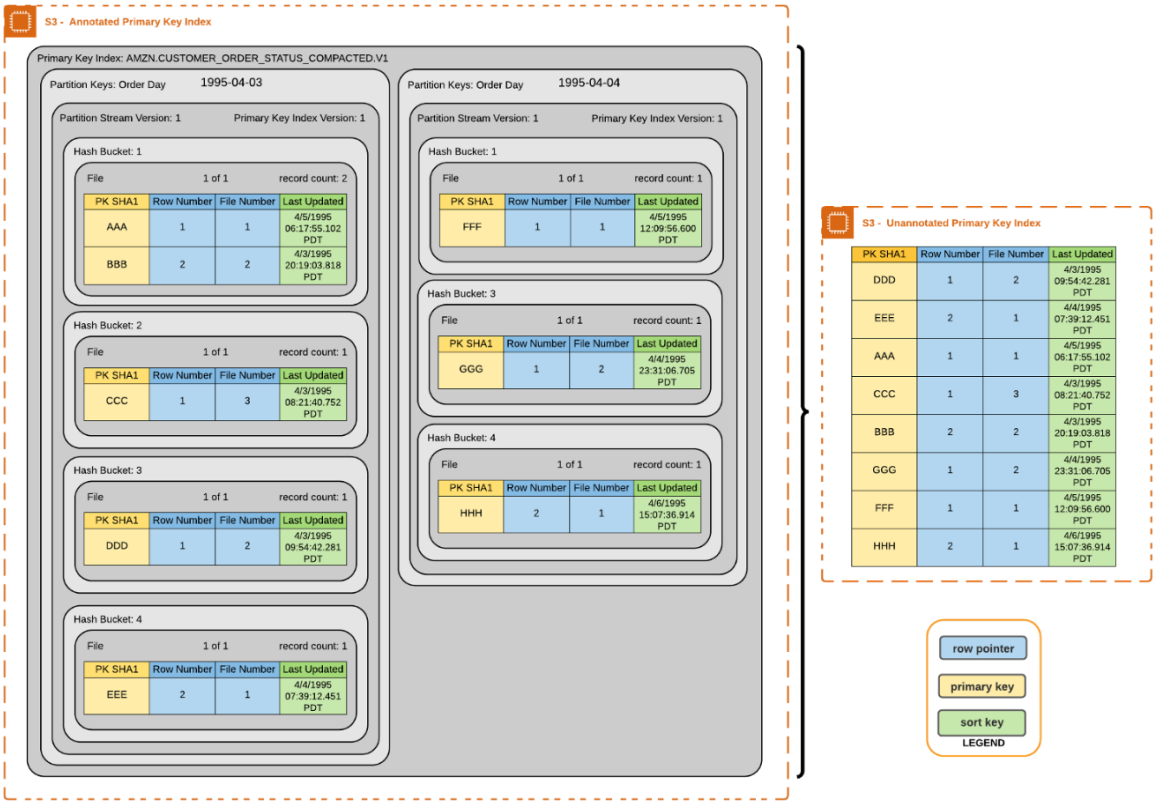
- 20 Bytes (required): SHA-1 digest of all ordered, concatenated, primary key bytes
- 12 Bytes (required): compacted primary key row pointer (for lazy materialization)
    - 4 Bytes: compacted *partition stream position file number*
    - 8 Bytes: compacted *file row index*
- 1 to 32 Bytes (optional): custom *sort keys*

Note that the *compacted partition stream position* is omitted since it is always the first *partition stream position* in the *compacted partition stream version*. Each *primary key index* file is associated with a parent *hash bucket* derived from its *SHA-1 primary key digest* and contains a maximum of 38,000,000 rows by default (overridable to any positive integer). The *primary key index* is used to discover duplicate rows, and serves as a compute cache to save successive *compaction job runs* the time and cost otherwise incurred by recomputing each row's *SHA-1 primary key digest* and shuffling it into an associated *hash bucket*.

The following diagram illustrates the relationship between a compacted output table and its primary key index:



Viewing the fully annotated primary key index reveals its internal structure:



PRIMARY KEY INDEX REHASHING

As rows accumulate in compacted tables over time, more *hash buckets* may be required to successfully complete *compaction*

job runs. If this is the case, then the existing *primary key index version* must be rehashed into a new *primary key index version*. Rehashing is run on a dedicated Ray cluster on-demand. It is computationally lightweight and moderately I/O intensive, with better average end-to-end latency and cost efficiency than compaction itself.

Rehashing shuffles each *primary key index* row to a new *hash bucket* file by taking its existing *SHA-1 primary key digest* modulo the new *hash bucket* count. Python pseudocode: `hash_bucket_index = int.from_bytes(row_primary_key_sha1_digest_bytes) % HASH_BUCKET_COUNT`.

After rehashing completes successfully, all files associated with the old *primary key index version* can optionally be deleted to reduce ongoing S3 cost.

COMPACTION JOB RUNS

Compaction job runs are initiated by the *job run dispatcher* - a lightweight, autoscaling, high-availability Ray cluster that listens for new job run *instruction documents* arriving in an SQS queue. The *job run dispatcher* creates a job run *session launcher* for every *compaction job run* it receives. High-availability is achieved by launching at least 2 active *job run dispatcher* clusters via AWS Step Function workflow tasks using a default heartbeat timeout of 5 minutes. If any cluster fails to emit a heartbeat to its parent Step Function task within this window then the workflow will terminate its EC2 nodes and provision a new cluster. The workflow will be continued as new prior to reaching either its 25,000 event history limit or its 90th day of execution, whichever comes first.

The *instruction document* for a *compaction job run* contains explicit values for all parameters required by the compactor's minimal Python API signature, and optionally includes parameters to override all default values in the compactor's full Python API signature:

Minimal Python API signature:

```
def compact(
     source_table_id: str,
     primary_keys: Set[str])
```

Full Python API signature:

```
def compact(
     source_table_id: str,
     primary_keys: Set[str],
     sort_keys: List[Tuple[str, str]] = [(PARTITION_STREAM_POS_VIRTUAL_COL_NAME, "ascending")],
     desired_sla: Optional[int] = None,
     delta_catalog_client: DeltaCatalog.s3,
     destination_compacted_table_id: Optional[str] = None,
     rows_per_primary_key_index_file: int = 38_000_000,
     rows_per_compacted_file: int = 4_000_000,
     hash_bucket_count: Optional[int] = None,
     sns_event_topic_arn: Optional[str] = None,
     compacted_file_content_type = ContentType.PARQUET)
```

The latest implementation of the above APIs are currently available at https://github.com/ray-project/deltacat/tree/main/deltacat/compute/compactor. Here, *delta_catalog_client* is the client module used to read/write tables from/to durable storage, with a pure S3 implementation used by default. The source table partition streams targeted for compaction are implicitly resolved to the latest active *partition stream version* for each distinct set of *partition keys*. By default, the destination *compacted table ID* is identical to the *source table ID*. The input *primary key* set is sorted to ensure that their field bytes are always concatenated in deterministic order when computing *SHA-1 primary key digests.* If sort keys are provided, all must reference unique column names.

COMPACTION SESSIONS

A *compaction job run* given multiple input table *partition keys* from the same *source table ID* will create a dedicated *compaction session* for each *partition stream*. Each *compaction session* runs on an isolated Ray cluster concurrently with all other sessions. Each *compaction session* is launched with the *compaction job run's desired SLA*, outputs an *expected SLA* when it starts running, and continues to refine this *expected SLA* based on performance and resource availability. A *compaction job run's expected SLA* is equal to the max *expected SLA* for all of its *compaction sessions*.

A *session launcher* is responsible for launching the Ray cluster associated with a *compaction session*. The *session launcher* is a lightweight, autoscaling Ray cluster that determines the initial and maximum resources that the session's cluster can consume during execution. Initial and maximum resources are determined using the *compaction job run's desired SLA* and input dataset stats (e.g. bytes, rows, content type, etc.). Each *compaction session* scales its cluster resources on-demand via Ray's AWS Autoscaler, and is restricted to nodes that provide at least 8GB of memory per CPU.

The *session launcher* also determines the preferred number of *hash buckets* that a session should use in its *primary key index*, and launches Ray clusters to rehash existing *primary key indices* as required. Arbitrarily large input datasets can be compacted using any size cluster as long as they are split into a sufficient number of *hash buckets*. *Compaction session* latency increases as *hash bucket* counts increase and as cluster size decreases (see figures 3-1, 3-2, and 3-3). The initial number of *hash buckets* dedicated to each *partition stream's primary key index* is chosen such that its *compaction session* consumes approximately 1/2 of available Python heap memory during deduplication. If the number of *hash buckets* inherited from a prior *compaction session* is small enough that more than 90% of available Python heap memory will be consumed, then the *primary key index* is first rehashed to a *hash bucket* count large enough to ensure that it will consume approximately 1/2 of available Python heap memory.

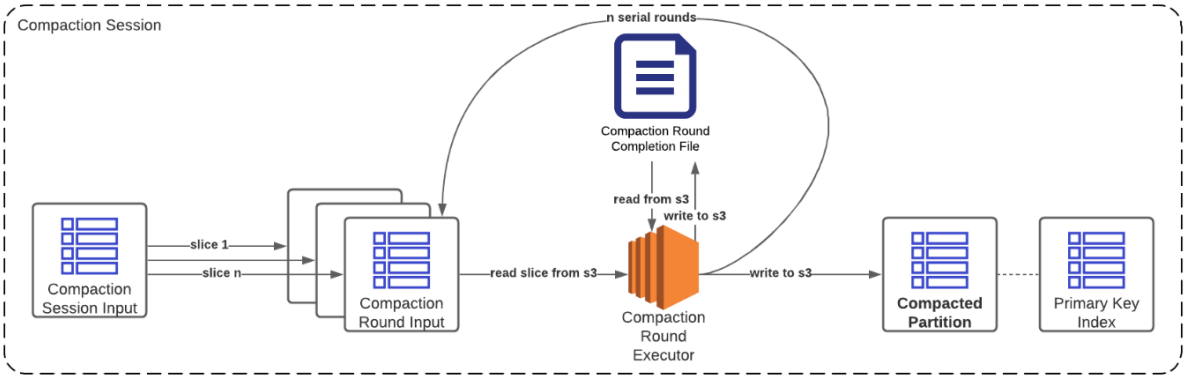
Since all rows consume a fixed amount of memory during compaction (via fixed *primary key index* row byte widths) and have nearly perfect uniform random distribution across cluster nodes (via primary-key-SHA1-digest-based shuffling), Ray cluster resource requirements and *hash bucket* counts required to both successfully complete a *compaction session* and meet the *compaction job run's desired SLA* can be accurately computed by the *session launcher* prior to cluster launch.

COMPACTION ROUNDS

To support compacting *partition streams* larger than available cluster memory/disk space, all input *delta files* (i.e. all files added to the *partition stream* since it was last compacted) for a single *compaction session* are split into N subsets and processed across N serial *compaction rounds* scheduled according to the following rules:

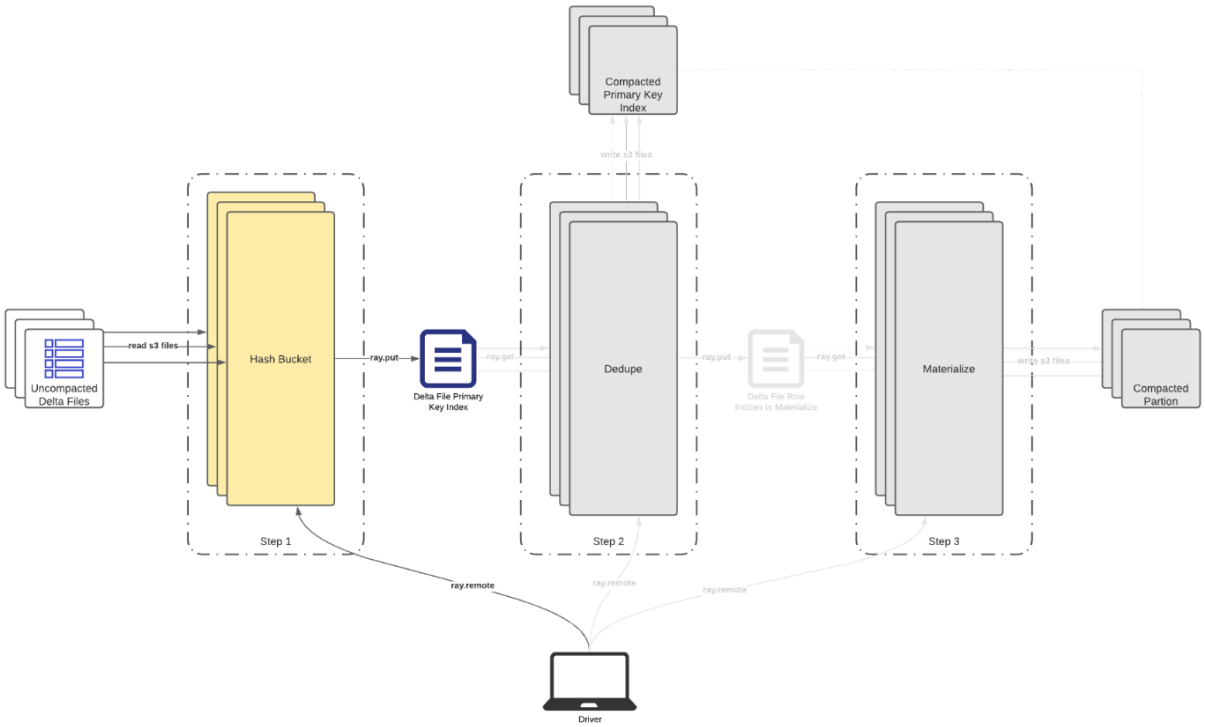
1. The input size of each round must fit within available cluster memory or, more specifically, all intermediate in-memory outputs produced during this round must fit within the cluster's *Plasma Object Store*.
2. All input files for round K of N must have a later *partition stream position* sort order than those of round K-1.
3. Each round must write a *primary key index* to S3 such that round K of N can efficiently dedupe its output with the output of rounds K-1, K-2, ..., 1.
4. After a *compaction round* completes, a *compaction round completion file* annotated with this *compaction session's source table ID*, *partition keys,* and *source partition stream version* must be written to S3 with the latest:
    a. Source Partition Stream Position Processed
    b. Hash Bucket Count
    c. Compacted Table ID
    d. Compacted Partition Stream Version
    e. Compacted Partition Stream Size (Bytes at Rest)
    f. Primary Key Index Version

This flow chart illustrates the relationship between a *compaction session* and its iterative *compaction rounds*:



Each *compaction round* starts by reading the *compaction round completion file* for the previous round, then processes its input *delta files* through three *bulk synchronous parallel* steps:

1. **Hash Bucket**: The *hash bucket step* takes all input *delta files* for the round as input and writes an in-memory *delta primary key index* to Ray's *Plasma Object Store* as output. Each parallel *hash bucket task* is scheduled according to the following rules:
    a. The maximum number of concurrent *hash bucket tasks* is equal to the number of cluster CPUs. In other words, each *hash bucket task* consumes 1 CPU.
    b. Each cluster node is assigned a roughly equal number of *hash bucket tasks* via a *load-balancing scheduler*.
    c. Each *hash bucket task* is assigned an approximately equal-size subset of this round's input *delta files*, where each subset is limited to fit in a single *hash bucket task's* available Python heap memory (i.e. the Python heap memory available per CPU).
    d. The total number of *hash bucket tasks* launched is equal to the number of input *delta file* subsets.



Each parallel *hash bucket task* runs on a single thread in an isolated process. Each task runs the following steps:

a. Read *primary key* and *sort key* columns from each input *delta file* into a set of in-memory PyArrow Tables, **S[T]**. Python pseudocode:

b.
```
input_file_to_table: Dict[str, pyarrow.Table] = read_input_files(
  file_paths,
  primary_keys,
  sort_keys)
```

c. Wrap all PyArrow Tables in **S[T]** in a list of *delta file envelopes*, **L[E[T]]**, denoting the source *delta file's partition stream position*, *partition stream position file number*, and *operation type*. Python pseudocode:

d.
```
delta_file_envelopes: List[Dict[str, Any]] = [{
  "partition_stream_position": get_input_file_stream_position(input_file),
  "file_number": get_input_file_number(input_file),
  "operation_type": get_input_file_operation_type(input_file),
  "table": table
} for input_file, table in input_file_to_table.items()]
```

e. Append a new column to each PyArrow Table in **S[T]** containing the SHA-1 digest of each row's *primary key* values. Python pseudocode:

f.
```
for table in input_file_to_table.values():
  table.append_column(
    pyarrow.field(name=pk_sha1_digest_column_name, type=pyarrow.binary(20)),
    sha1_digest_generator(all_pk_fields))
```

g. Drop *primary key* columns from each PyArrow Table in **S[T]**. Python pseudocode:

h.
```
for table in input_file_to_table.values():
  table.drop(primary_key_columns)
```

i. Group *source file row indices* by *hash bucket* for each PyArrow Table in **S[T]** to form **D[I]**. Python pseudocode:

j.
```
for table in input_file_to_table.values():
  hash_bucket_to_row_indices: Dict[int, List[int]] = defaultdict(list)
  row_index = 0
  for pk_sha1_digest in table[pk_sha1_digest_column_name]:
    hash_bucket_index = int.from_bytes(pk_sha1_digest, "big") % num_buckets
    hash_bucket_to_row_indices[hash_bucket_index].append(row_index)
    row_index += 1
```

k. Use **D[I]** to group rows by *hash bucket* for each PyArrow Table in **S[T]** to produce a new PyArrow Table dictionary, **D[T]**. At the same time, append a new 8-byte *source row index column* for each table in **D[T]** that maps each *primary key SHA-1 digest* to its *source file row index*. Python pseudocode:

l.
```
  hash_bucket_to_table = Dict[int, pyarrow.Table]
  for hash_bucket_index, row_indices in hash_bucket_to_row_indices.items():
    hb_table = table.take(row_indices)
    hash_bucket_to_table[hash_bucket_index] = hb_table.append_column(
      pyarrow.field(name=src_row_index_column_name, type=pyarrow.int64()),
      row_indices)
```

m. Rewrite *delta file envelopes,* **L[E[T]],** grouped by *hash bucket* for each PyArrow Table in **D[T]** to yield **D[L[E[T]]**. This represents the final in-memory *delta primary key index* that will be output by the *hash bucket task*. Python pseudocode (where `group_by_pk_hash_bucket` runs steps c-f above):

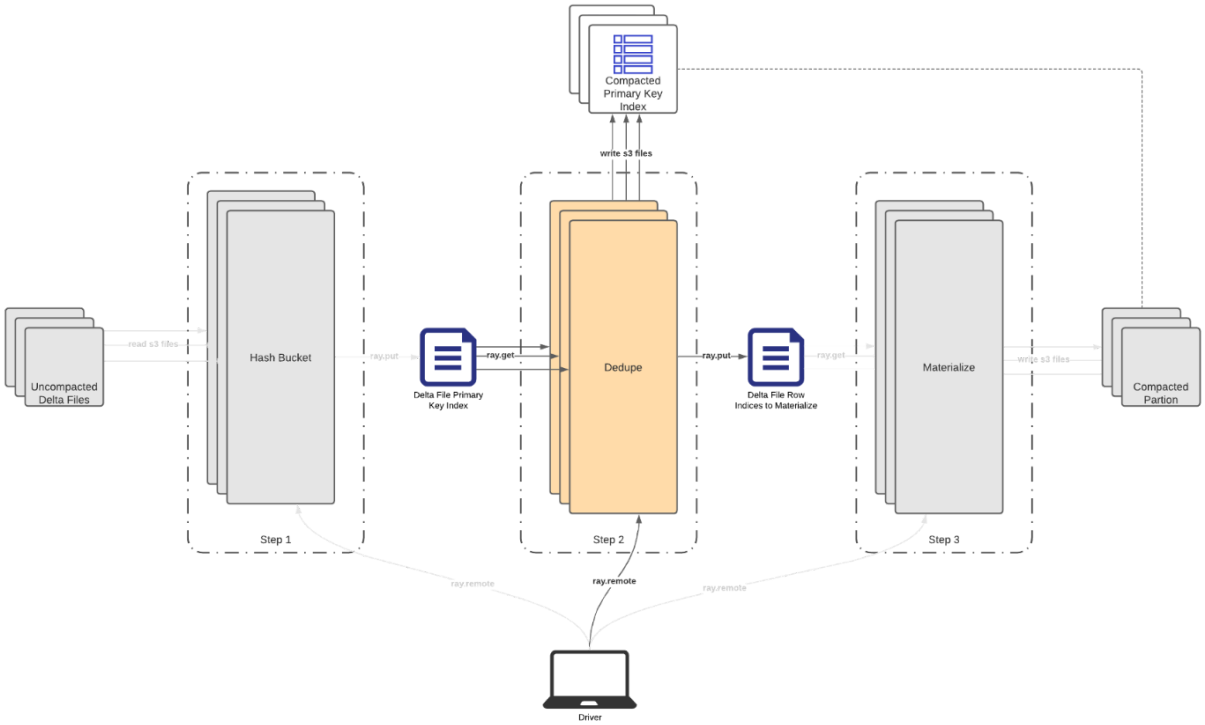
n.
```
hash_bucket_to_delta_file_envelopes = defaultdict(list)
for envelope in delta_file_envelopes:
```

```
    hash_bucket_to_table = group_by_pk_hash_bucket(envelope["table"])
    for hash_bucket_index, table in hash_bucket_to_table.items():
      hash_bucket_to_delta_file_envelopes[hash_bucket_index].append({
        "partition_stream_position": envelope["partition_stream_position"],
        "file_number": envelope["file_number"],
        "operation_type": envelope["operation_type"],
        "table": table
      })
```

o. Write the *delta primary key index* for each *hash bucket* in **D[L[E[T]]** to the local node's *Plasma Object Store*, grouping *Plasma Object References* by *hash bucket*. Python pseudocode:

p.
```
hash_bucket_to_obj_ref: Dict[int, ObjectRef] = {
  hash_bucket_index:ray.put(delta_file_envelopes) \
    for (hash_bucket_index, delta_file_envelopes) \
    in hash_bucket_to_delta_file_envelopes.items()
}
```

2. **Dedupe**: The *dedupe step* takes the in-memory *delta primary key index* from the *hash bucket step* as input, and any existing *primary key index* for this *hash bucket* from S3 as input. It writes in-memory *source file row indices* to materialize to Ray's *Plasma Object Store* as output (grouped by source file), and writes deduped *primary key index* files as output. Each parallel *dedupe task* is scheduled according to the following rules:
    a. The maximum number of concurrent *dedupe tasks* is equal to the number of cluster CPUs. In other words, each *dedupe task* consumes 1 CPU.
    b. Each cluster node is assigned a roughly equal number of *dedupe tasks* via a *load-balancing scheduler*.
    c. Each *dedupe task* is given the in-memory *delta primary key index* for a single *hash bucket* as input, where the *delta primary key index* for each *hash bucket* is expected to consume a roughly equivalent amount of memory and fit within a single dedupe task's available memory (i.e. the available memory per CPU).
    d. Each *dedupe task* is assigned a monotonically increasing *dedupe task index* along the interval [0, *total_number_of_dedupe_tasks*) where *total_number_of_dedupe_tasks* is equal to the number of *hash buckets* output from the *hash bucket step*. This index determines the order of materialization of its *source file row indices* relative to every other *dedupe task*.
    e. Each *dedupe task* is given a handle to the same *RowCountsPendingMaterialize* Ray actor instance. Source file row counts pending materialization are stored in this shared actor instance grouped by *dedupe task index*. These row counts are then used to build a predictive *primary key index* whose row pointers reference rows in the compacted table pending materialization. This provides substantial performance and efficiency benefits vs. recomputing the same *primary key index* post-materialization and is a unique feature of our design made possible by Ray.



Each parallel *dedupe task* runs on a single thread in an isolated process. Each task runs the following steps:

a. Read any existing *primary key index* for this *hash bucket* from S3 and project the implicit *compacted partition stream position* onto every row of this table. Save the result as the first item in an ordered list of PyArrow Tables, **L[T]**. Python pseudocode:

b.
```
primary_key_index_tables: List[pyarrow.Table]
if latest_compacted_table_id:
  primary_key_index_table = read_durable_primary_key_index(
    latest_compacted_table_id,
    partition_keys,
    latest_compacted_partition_stream_version,
    hash_bucket,
    latest_primary_key_index_version)
  partition_stream_position = get_first_partition_stream_position(
    latest_compacted_table_id,
    partition_keys,
    latest_compacted_partition_stream_version)
  primary_key_index_table.append_column(
    pyarrow.field(name="partition_stream_position", type=pyarrow.int64()),
    repeat(partition_stream_position, primary_key_index_table.num_rows)
  primary_key_index_tables.append(primary_key_index_table)
```

c. Sort all *delta file envelopes* from the input in-memory *delta primary key index* for this *hash bucket,* **L[E[T]]**, by (1) *partition stream position* and (2) *partition stream position file number*. This ensures that dedupe results remain deterministic for the same input dataset across multiple job runs (e.g. whenever two rows have identical *sort keys*), and presents the correct final deduplication order when using only the default *partition stream position* sort key. Python pseudocode:

d.
```
sorted_delta_file_envelopes = sorted(
  delta_file_envelopes,
```

```
    key=lambda dfe: (dfe["partition_stream_position"], dfe["file_number"]),
    reverse=False) # ascending
```

e. Project required *delta file envelope* metadata (*partition stream position*, *partition stream position file number*, and *operation type*) onto new columns for each ordered table in **L[E[T]]** and append the result to **L[T]**. Python pseudocode:

f.
```
for envelope in sorted_delta_file_envelopes:
  table = envelope["table"]
  table.append_column(
    pyarrow.field(name="partition_stream_position", type=pyarrow.int64()),
    repeat(envelope["partition_stream_position"]), table.num_rows)
  table.append_column(
    pyarrow.field(name="file_number", type=pyarrow.int32()),
    repeat(envelope["file_number"]), table.num_rows))
  table.append_column(
    pyarrow.field(name="operation_type", type=pyarrow._bool()),
    repeat(envelope["operation_type"]), table.num_rows))
  primary_key_index_tables.append(table)
```

g. Perform a zero-copy concatenation of all PyArrow tables in **L[T]** to form a new logical PyArrow Table, **C[T]**, holding the *source file primary key index union* for this hash bucket. Python pseudocode:

h.
```
primary_key_index_union = pyarrow.concat_tables(primary_key_index_tables)
```

i. Drop duplicates from the *source file primary key index union* PyArrow Table, **C[T]**, according to the *primary key SHA-1 digest*, *sort keys*, and delta file *operation type*. Python pseudocode:

j.
```
primary_key_index_union = drop_duplicates(
  primary_key_index_union,
  "operation_type",
  pk_sha1_digest_column_name,
  sort_key_column_names)
```

k. Drop the *operation type* column from the *source file primary key index union* PyArrow Table, **C[T]**. Python pseudocode:

l.
```
primary_key_index_union.drop("operation_type")
```

m. Group *source file row indices* to materialize by *source file ID* to form **D[I]**. Python pseudocode:

n.
```
src_file_id_to_row_indices:
  Dict[tuple[int, int, int], List[int]] = defaultdict(list)
for row_index in range(primary_key_index_union.num_rows):
  source_file_id = (
    primary_key_index_union["partition_stream_position"][row_index],
    primary_key_index_union["file_number"][row_index])
  source_file_row_index = \
    primary_key_index_union[src_row_index_column_name][row_index]
  src_file_id_to_row_indices[source_file_id].append(source_file_row_index)
```

o. Group *source file row indices* to materialize in **D[I]** by their target *materialize task index* and *source file ID* to form **D[D[I]]**. Also capture *row counts pending materialization* grouped by *materialize task index* and *source file ID* as **D[D[C]]**. Python pseudocode:

p.
```
materialize_task_idx_to_src_file_row_count:
  Dict[int, Dict[tuple[int, int, int], int] = defaultdict(dict)
materialize_task_idx_to_src_file_row_indices:
  Dict[int, Dict[tuple[int, int, int], List[int]] = defaultdict(dict)
for src_file_id, src_file_row_indices in src_file_id_to_row_indices.items():
```

```
  mat_task_idx = file_id_to_materialize_task_index(src_file_id)
  mat_task_idx_to_src_file_rows[mat_task_idx][src_file_id] = \
    np.array(src_file_row_indices)
  mat_task_idx_to_src_file_row_count[mat_task_idx][src_file_id] = \
    len(src_file_row_indices)
```

q. Write in-memory *source file row indices* and the current *dedupe task index* grouped by *materialize task index* and *source file ID*, **D[D[I]]**, to the local node's *Plasma Object Store*, grouping *Plasma Object References* by *materialize task index*. Python pseudocode:

r.
```
materialize_task_idx_to_indexed_obj_ref = {}
for mat_task_idx, src_file_row_indices in \
  materialize_task_idx_to_src_file_row_indices.items():
    materialize_task_idx_to_indexed_obj_ref[mat_task_idx] = {
        "dedupe_task_index": dedupe_task_index,
        "obj_ref": ray.put(src_file_row_indices)
    }
```

s. Add this dedupe task's *row counts pending materialization*, **D[D[C]]** to the *RowCountsPendingMaterialize* actor. Python pseudocode:

t.
```
materialize_task_row_counter.add_row_counts(
    dedupe_task_index,
    materialize_task_idx_to_src_file_row_count)
```

u. Wait for all *dedupe tasks* to add their *row counts pending materialization*, **D[D[C]]**, to the *RowCountsPendingMaterialize* actor, then use them to convert the *source file primary key index union* row pointers in **C[T]** to compacted destination file row pointers pending materialization (i.e. convert source/delta file *partition stream position file numbers* and *source file row indices* to their destination/compacted file equivalents). This converted index forms the final *compacted file primary key index* PyArrow Table, **P**, to be written to S3. Python pseudocode:

v.
```
compacted_primary_key_index_table = source_to_dest_file_primary_key_index(
  primary_key_index_union,
  materialize_task_row_counter.get_record_counts(),
  max_rows_per_primary_key_index_file,
  max_rows_per_compacted_file,
  dedupe_task_index)
```

w. Write the *compacted file primary key index*, **P**, to one or more *primary key index* output files in S3 and delete the old *primary key index*. Python pseudocode:

x.
```
write_durable_primary_key_index(
  compacted_primary_key_index_table,
  destination_compacted_table_id,
  partition_keys,
  latest_compacted_partition_stream_version + 1,
  hash_bucket,
  pk_index_version=1)
delete_durable_primary_key_index(
  latest_compacted_table_id,
  partition_keys,
  latest_compacted_partition_stream_version,
  hash_bucket,
  latest_primary_key_index_version)
```

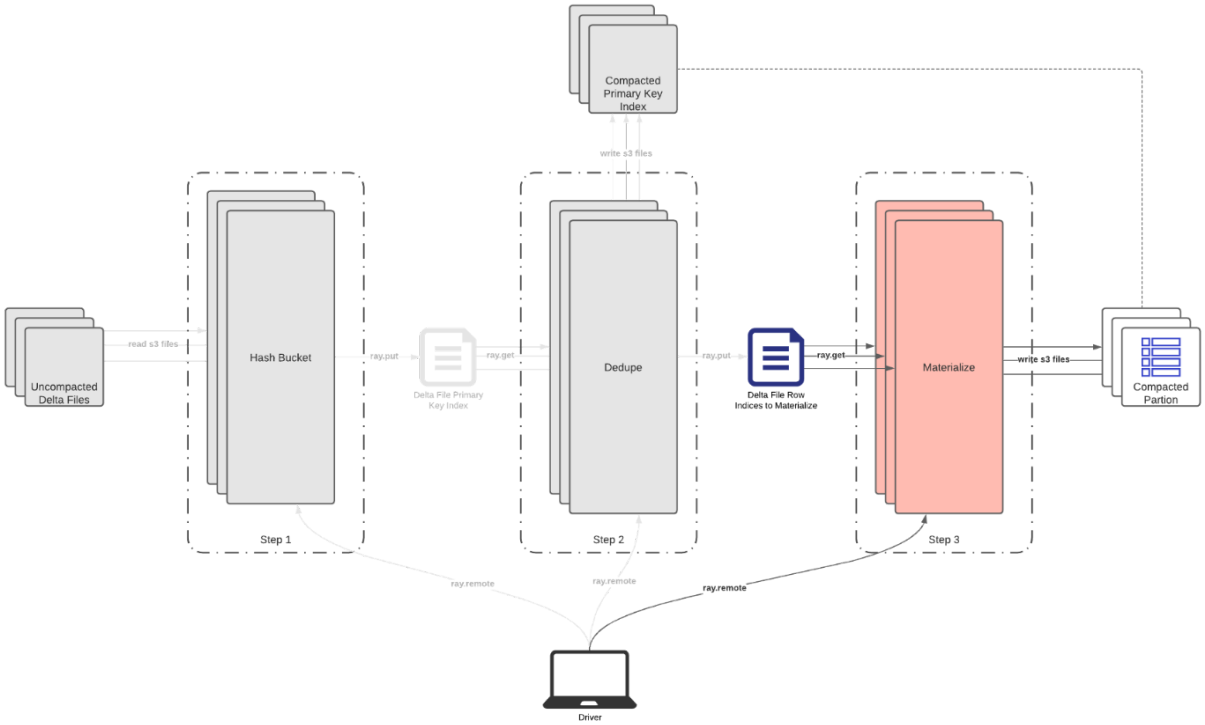
3. **Materialize**: The *materialize* step takes all in-memory *source file row indices* to materialize from the *dedupe step* as input, and writes compacted tabular storage files annotated with their *compacted table ID, partition keys, compacted partition stream version, compacted partition stream position*, *compacted partition stream position file number*, and upsert *partition stream operation type* as output. Each parallel *materialize task* is scheduled according to the following rules:

    a. The maximum number of concurrent *materialize tasks* is equal to the number of cluster CPUs. In other words, each *materialize task* consumes 1 CPU.

    b. Each *materialize task* is assigned a monotonically increasing *materialize task index* along the interval [0, *total_number_of_materialize_tasks*), where *total_number_of_materialize_tasks* is equal to the number of source file groups created by the *dedupe step*.

    c. Each *materialize task* is given all *source file row indices* for 1 or more source files assigned to its *materialize task index* as input. The number of source files assigned to a single *materialize task* is adjusted to balance load across concurrent *materialize tasks*, ensuring that overall *materialize step* latency does not increase due to straggler tasks taking on too large a workload.

    d. Each cluster node is assigned a roughly equal number of *materialize tasks* via a *load-balancing scheduler*.



Each parallel materialize *task* runs on a single thread in an isolated process. Each task runs the following steps:

    a. Group the input *source file row indices* to materialize, **L[D[L[I]]]** by *source file ID* to form **D[L[I]]**. Python pseudocode:

    b.
```
src_file_id_to_rows = defaultdict(list)
for dedupe_task_idx, src_file_rows in dedupe_task_idx_to_src_file_rows.items
  for src_file_id, row_indices in src_file_rows.items():
    src_file_id_to_rows[src_file_id].append(
      (row_indices, repeat(dedupe_task_idx, len(row_indices)))
    )
```

c. Materialize all *source file row indices* for each *source file ID* in **D[L[I]]** sorted by dedupe task index to form the final list of compacted PyArrow tables, **L[C]**. Python pseudocode:

d.
```
compacted_tables = []
for src_file_id in sorted(src_file_id_to_rows.keys()):
  table = read_table(src_file_id)
  row_filter_mask = list(repeat(False, table.num_rows))
  for row_index in chain_rows(src_file_id_to_rows[src_file_id]):
    row_filter_mask[row_index] = True
  compacted_table = table.filter(pa.array(row_filter_mask))
  dedupe_task_indices = chain_dedupe_tasks(src_file_id_to_rows[src_file_id])
  compacted_tables.append(sorted_table(compacted_table, dedupe_task_indices))
```

e. Perform a zero-copy concatenation of all tables in **L[C]** and write the result to S3, sorting all files output by parallel materialize tasks to the compacted table's single *partition stream position* by materialize task index. Python pseudocode:

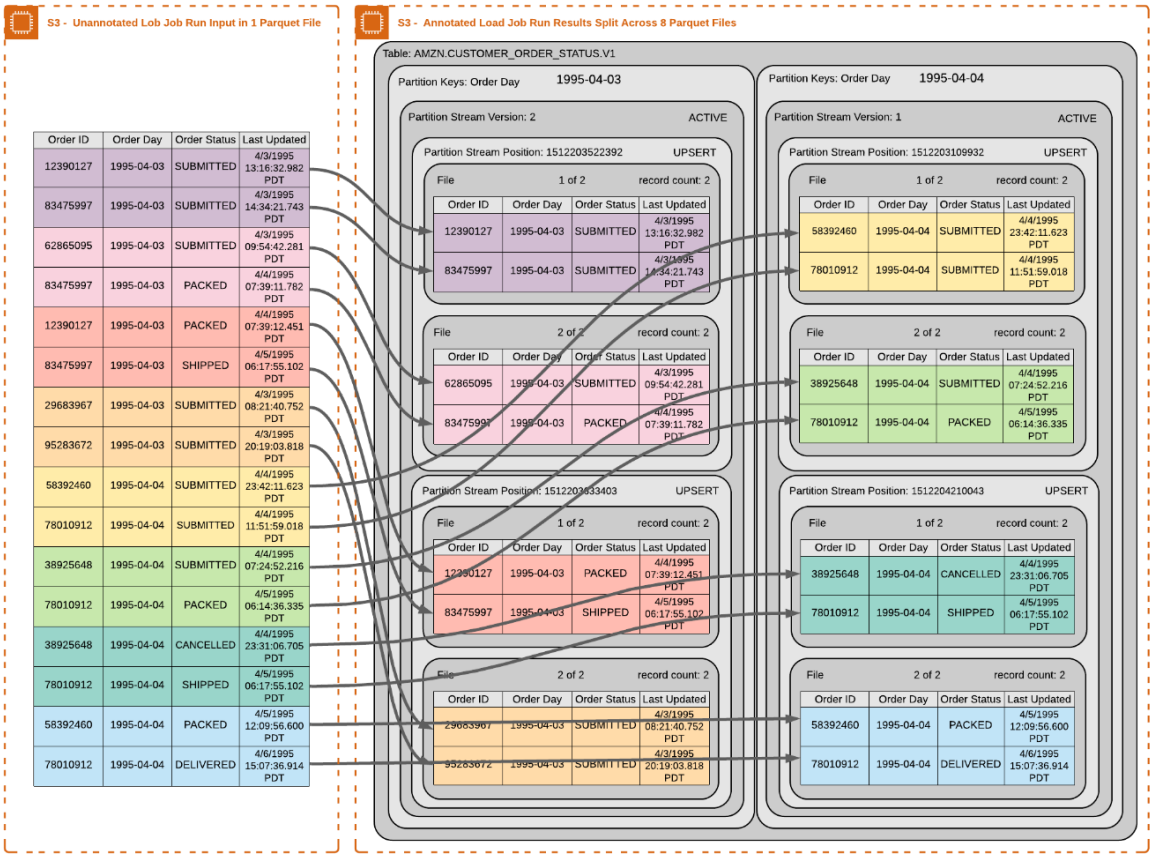
f.
```
write_durable_compacted_files(
  pa.concat_tables(compacted_tables),
  max_rows_per_compacted_file,
  compacted_file_content_type,
  destination_compacted_table_id,
  partition_keys,
  materialize_task_index,
  latest_compacted_partition_stream_version + 1)
```

COMPACTION ILLUSTRATED

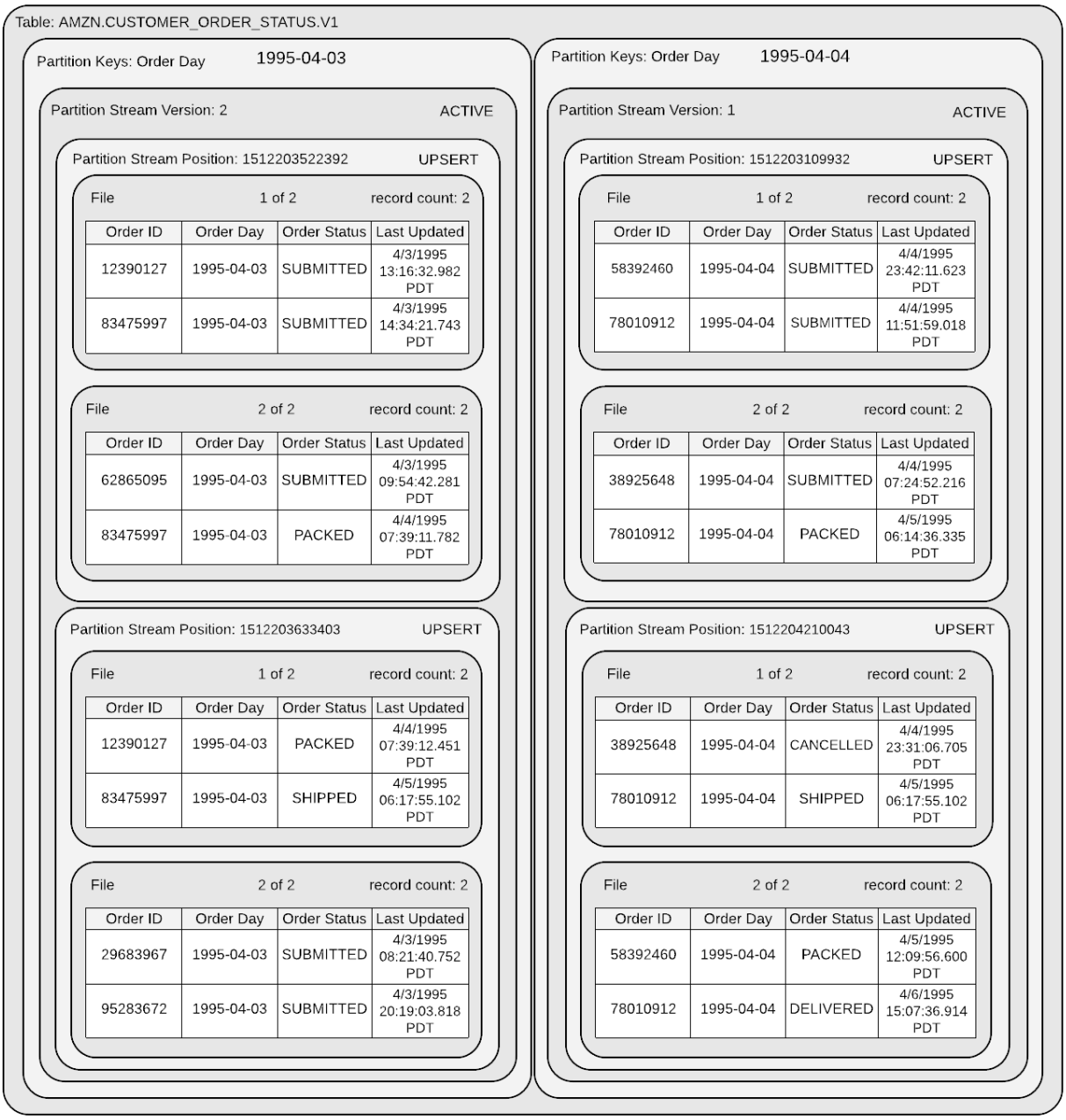
Assume the table AMZN.CUSTOMER_ORDER_STATUS tracks the current status of customer orders by unique order ID:

Order ID	Order Day	Order Status	Last Updated
12390127	4/3/1995	SUBMITTED	4/3/1995 13:16:32.982 PDT
83475997	4/3/1995	SUBMITTED	4/3/1995 14:34:21.743 PDT
62865095	4/3/1995	SUBMITTED	4/3/1995 09:54:42.281 PDT
83475997	4/3/1995	PACKED	4/4/1995 07:39:11.782 PDT
12390127	4/3/1995	PACKED	4/4/1995 07:39:12.451 PDT
83475997	4/3/1995	SHIPPED	4/5/1995 06:17:55.102 PDT
29683967	4/3/1995	SUBMITTED	4/3/1995 08:21:40.752 PDT
95283672	4/3/1995	SUBMITTED	4/3/1995 20:19:03.818 PDT
58392460	4/4/1995	SUBMITTED	4/4/1995 23:42:11.623 PDT
78010912	4/4/1995	SUBMITTED	4/4/1995 11:51:59.018 PDT
38925648	4/4/1995	SUBMITTED	4/4/1995 07:24:52.216 PDT
78010912	4/4/1995	PACKED	4/5/1995 08:33:36.335 PDT
38925648	4/4/1995	CANCELLED	4/4/1995 23:31:06.705 PDT
78010912	4/4/1995	SHIPPED	4/5/1995 06:17:55.102 PDT
58392460	4/4/1995	PACKED	4/5/1995 12:09:56.600 PDT
78010912	4/4/1995	DELIVERED	4/6/1995 15:07:36.914 PDT

Recall that all rows of this table are contained in files annotated with their *source table ID, partition keys*, *partition stream version*, *partition stream lifecycle state*, *partition stream position*, *partition stream position file number*, and *partition stream position operation type*. Let's assume that a load job run for the above table splits it across 8 files annotated as follows:



If we omit the row movement noise from the above diagram, then the current annotated table we're left with in S3 is:



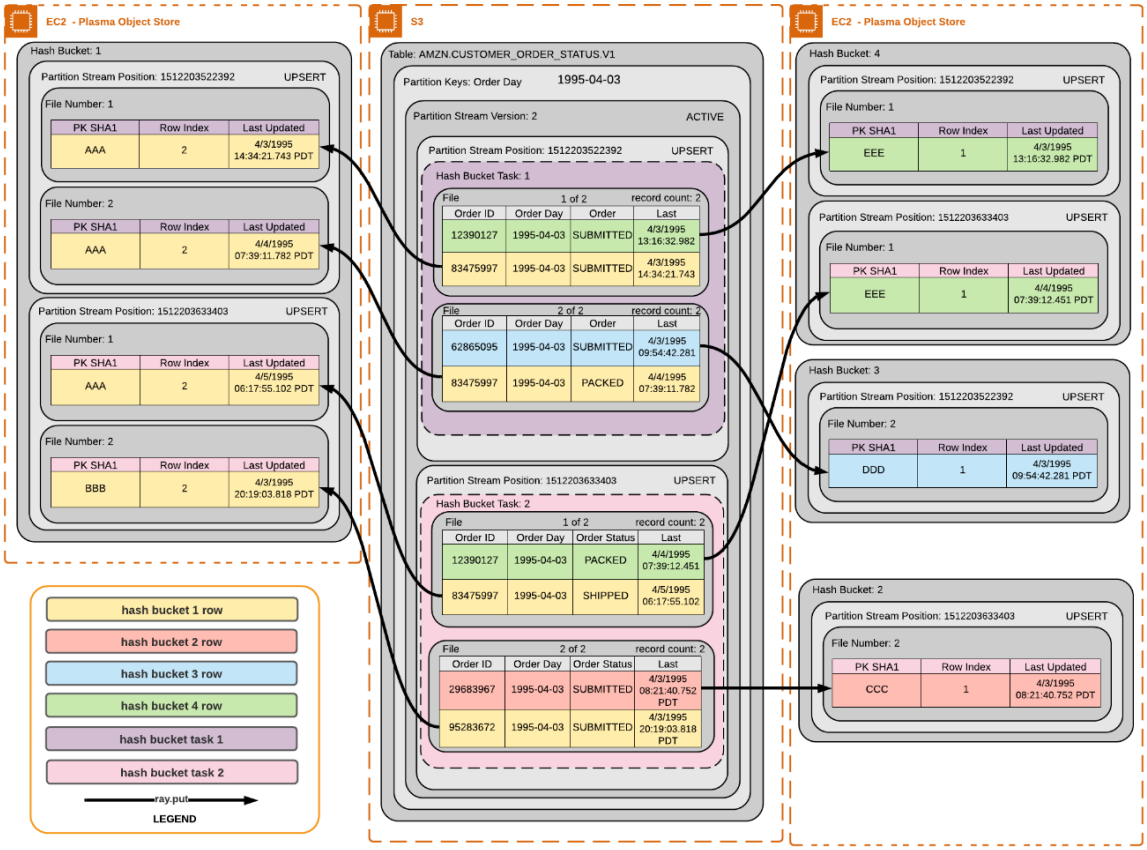
The completion of the above load job run triggers a job run instruction document to be sent to our SQS queue. The *job run dispatcher* receives this instruction document and uses it to initiate a *compaction job run* against this table via:

```
compact(
    source_table_id="AMZN.CUSTOMER_ORDER_STATUS.V1",
    primary_keys={"Order ID"},
    sort_keys=[("Last Updated", "ascending")],
    desired_sla=None,
    delta_catalog_client=DeltaCatalog.s3,
    destination_compacted_table_id="AMZN.CUSTOMER_ORDER_STATUS_COMPACTED.V1",
    rows_per_primary_key_index_file=2,
    rows_per_compacted_file=2,
    hash_bucket_count=4,
```

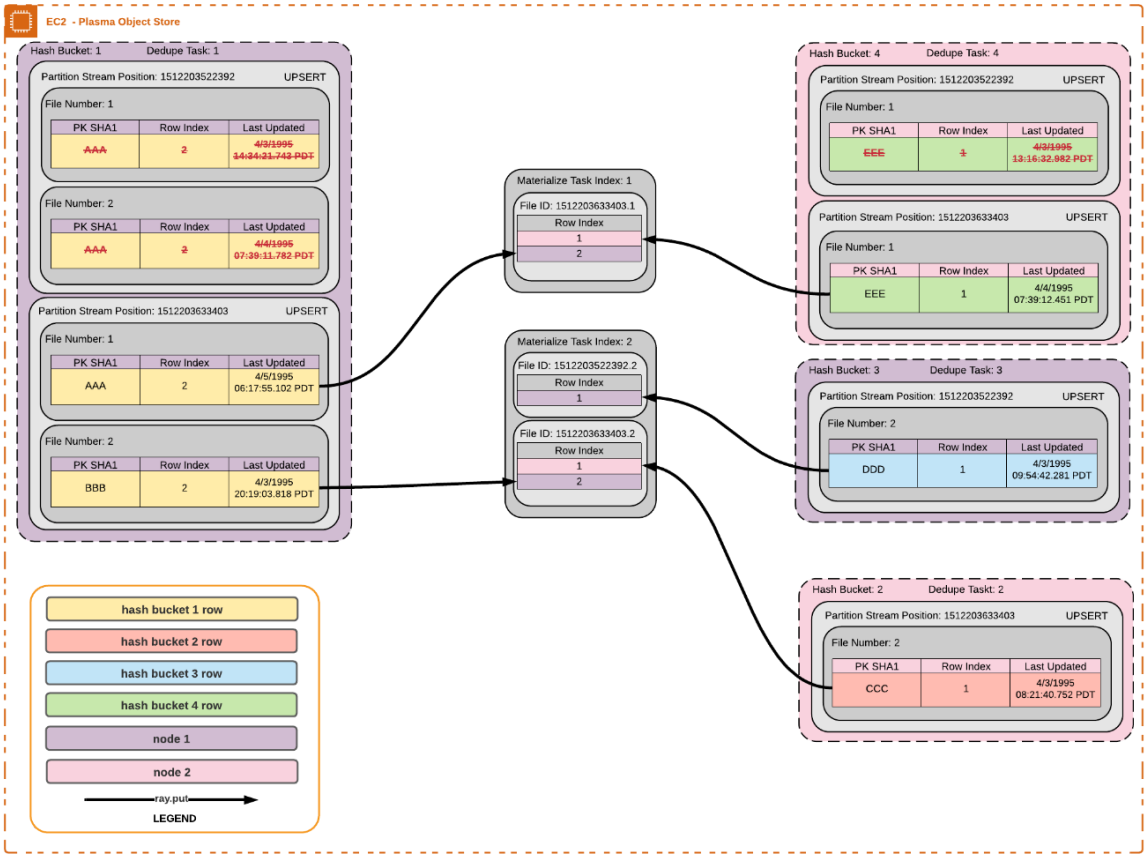
```
sns_event_topic_arn=arn:aws:sns:us-east-1:123456789012:CompactionJobRunEvents,
compacted_file_content_type=ContentType.PARQUET)
```

This, in turn, launches two parallel *compaction sessions* - one for the latest active *partition stream version* of the 1995-04-03 order day partition, and another for the latest active *partition stream version* of the 1995-04-04 partition.

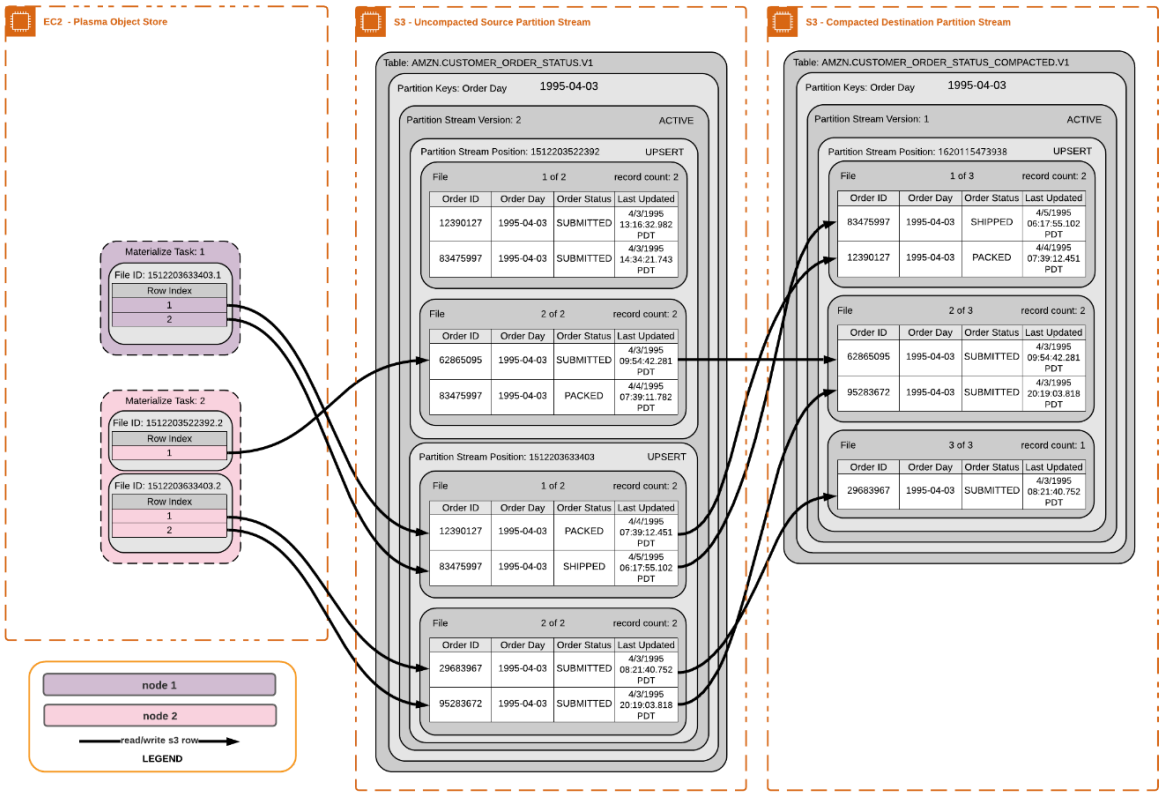
Let's also assume that we execute each *compaction session* on a tiny 2-node cluster where each node provides 2 CPUs and enough memory to ingest 2 rows per CPU. In the following figures, tasks launched or objects stored on node 1 will be colored purple while those owned by node 2 will be pink. We can then visualize the *hash bucket step* results for the first *compaction round* of the 1995-04-03 partition's *compaction session* as:



Next, the *dedupe step* takes the output from the *hash bucket step* and launches one *dedupe task* per hash bucket to produce deduplicated *source file row indices* to materialize grouped by *source delta file*:



Finally, the *materialize step* uses the *source file row indices* from the *dedupe step* to materialize the compacted output files:



Note that the rows in the compacted output *partition stream* are first sorted by (1) *materialize task index*, (2) *source file ID*, and (3) resolver *dedupe task index* to produce a dataset consistent with the predictive *primary key index* output by the *dedupe step*.

Results

SUMMARY

To measure the efficacy of our compactor, we ran it against production Parquet data lake tables whose *delta file* input sizes varied from 20TB to 1PB per partition stream. Comparisons to our current Spark compactor on EMR are based on the existing production implementation, whose Scala pseudocode is:

```
import unionedDs.sqlContext.implicits._
import org.apache.spark.sql.expressions.Window.partitionBy
import org.apache.spark.sql.functions.{col, row_number}

val window = partitionBy(primaryKeyColumns.map(k => unionedDs(k)): _*)
   .orderBy(unionedDs(PARTITION_STREAM_POSITION_COLUMN_NAME).desc)
unionedDs
   .withColumn(COMPACTION_ROW_NUMBER_COLUMN_NAME, row_number.over(window))
   .where(col(COMPACTION_ROW_NUMBER_COLUMN_NAME) === 1)
   .drop(COMPACTION_ROW_NUMBER_COLUMN_NAME)
```

To determine the adherence of our design and implementation to our goals, we measured its *performance, scalability, efficiency, resilience,* and *SLA adherence* across 402 compaction test cases ingesting production Parquet datasets:

1. **Performance - Maximum Throughput Rate**: **1300TiB/hr** (1.42PB/hr, 23.8 TB/min) on a 250 node r5n-8xlarge cluster

(8000 vCPUs) compacting 117TiB of decompressed Parquet input data.

   a. **Spark on EMR Comparison:** ~13X our max throughput rate on Spark.

   b. **Notes:** For context, the current "general-purpose" (Daytona) 100TB sort record at sortbenchmark.org is 44.8 TB/min on a 10240 CPU cluster.

2. **Scalability - Largest Partition Stream Compacted in 1 Session: 1.05PiB** at 353TiB/hour (18,115 rows/s-core) on a 110 node r5n-8xlarge cluster (3520 vCPUs).

   a. **Spark on EMR Comparison:** ~12X larger than what we can compact on an equivalently sized Spark cluster.

   b. **Notes:** This illustrates the ability of our compaction algorithm to pull successive rounds of data to compact into working memory from S3 (e.g. the requested input data to compact does not need to all fit into cluster node memory or disk space). We are thus able to accommodate arbitrarily large input datasets at a cost of gradually lower throughput per core (with the only real question being how long a customer is willing to wait for the job run to complete, and how much money they're willing to spend).

3. **Efficiency - Best Cluster Utilization Rate: $0.24/TB** or 61,477 rows/s-core (91 MiB/s-core) on an 11 node r5n-8xlarge (352 vCPUs) EC2 On-Demand cluster compacting ~20TiB of decompressed input Parquet data.

   a. **Spark on EMR Comparison:** ~91% cost-reduction vs. Spark on EMR at our average efficiency of $0.46/TB.

   b. **Notes:** For context, the current cloud sort cost record at sortbenchmark.org is $1.44/TB.

4. **Resilience - Success Rate: 99.3%** or 133/134 *compaction job runs* successfully completed on average with *compaction session* processing ~100TiB of input *delta files*.

   a. **Spark on EMR Comparison:** ~3% improvement over our trailing 1-year success rate of 96.36% with Spark.

   b. **Notes:** Historically, the most common cause of job run failure was https://github.com/ray-project/ray/issues/14886.

5. **SLA Adherence: 99.3%** or, in other words, *compaction job run* adherence to its initial *expected SLA* is limited by job run success rate (i.e. job runs typically only fail to meet their *expected SLA* due to an unexpected crash or deadlock).

   a. **Spark on EMR Comparison:** No SLA guarantees for our Spark compactor.

   b. **Notes:** On average, 100 *compaction job runs* on the same size cluster, using the same number of hash buckets with the same input dataset size show up to 9.2% variance in their job run latencies. Thus, 99.3% adherence of *compaction job runs* to their initial *expected SLA* was achieved by simply adding a 10% job run latency buffer to the initial *expected SLA* calculated by each *compaction session*.

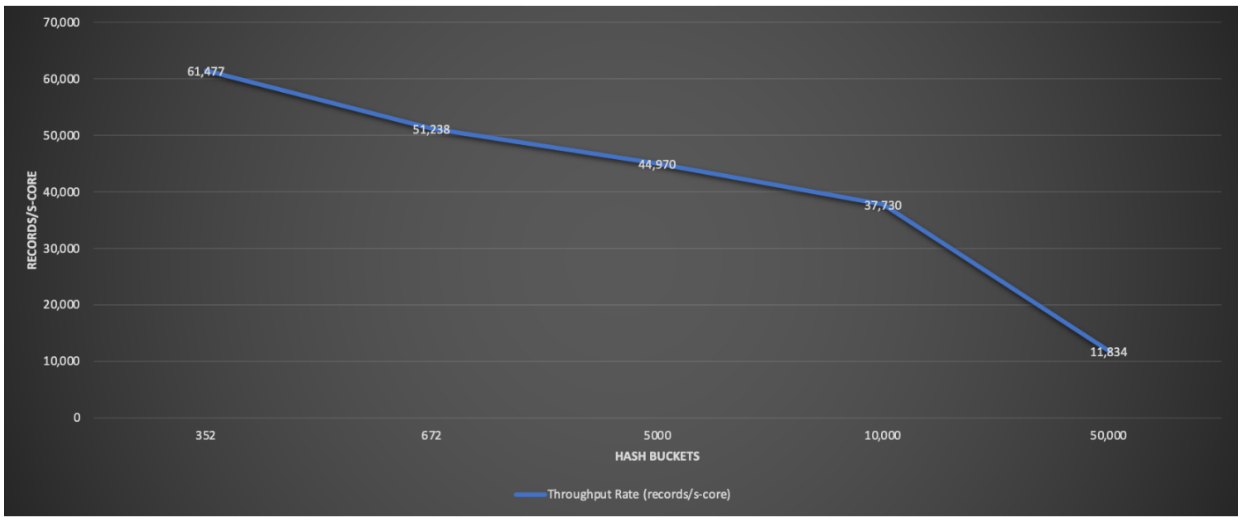
DATASET SIZE SCALABILITY

Figure 3-1: Recommended Hash Bucket Counts for Common Dataset Sizes

Input Records to Compact per Job Run (input deltas + prior compacted)	Approximate Input Bytes to Compact per Job Run	Recommended Hash Buckets	Recommended Memory Per CPU
<= 46.875 Billion	16 - 80 TiB	313	8 GiB/CPU
<= 93.75 Billion	32 - 160 TiB	625	8 GiB/CPU
<= 187.5 Billion	64 - 320 TiB	1250	8 GiB/CPU
<= 375 Billion	128 - 640 TiB	2500	8 GiB/CPU
<= 750 Billion	0.25 - 1.25 PiB	5000	8 GiB/CPU
<= 1.5 Trillion	0.5 - 2.5 PiB	10000	8 GiB/CPU
<= 3 Trillion	1 - 5 PiB	20000	8 GiB/CPU
<= 6 Trillion	2 - 10 PiB	40000	8 GiB/CPU

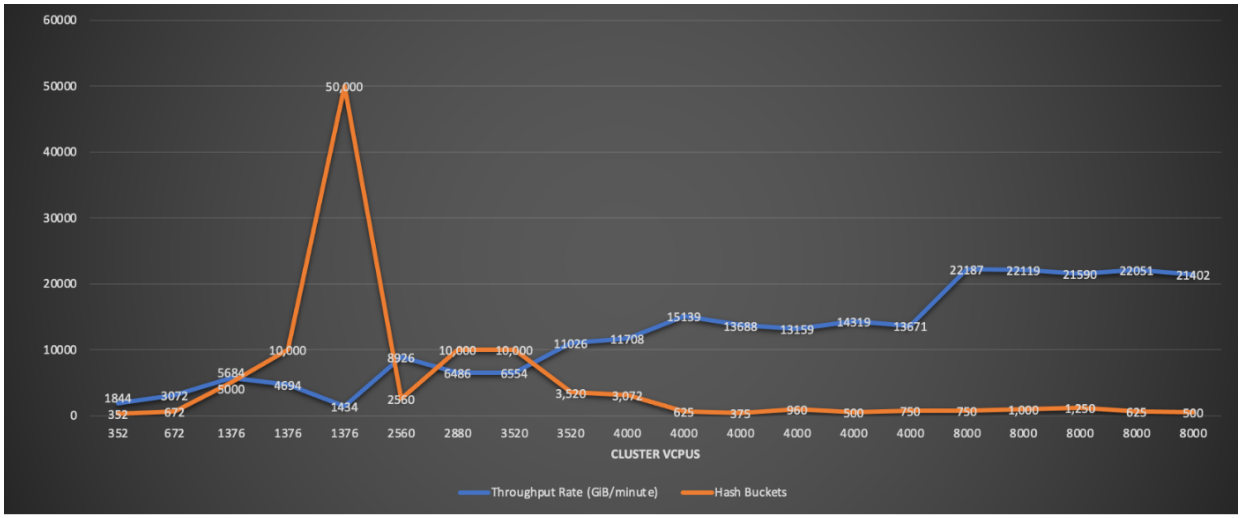
INPUT SIZE SCALING: EFFICIENCY

Figure 3-2: Compaction Records/s-core vs. Hash Bucket Count



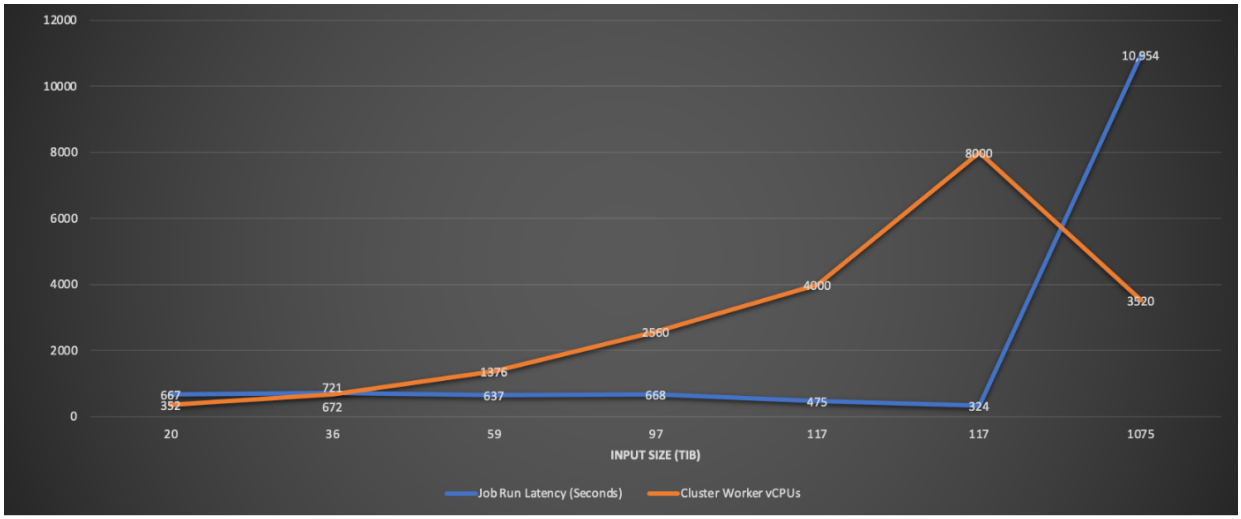
HORIZONTAL SCALING: PERFORMANCE

Figure 3-3: Compaction Throughput Rate vs. Hash Bucket Count and Cluster vCPUs



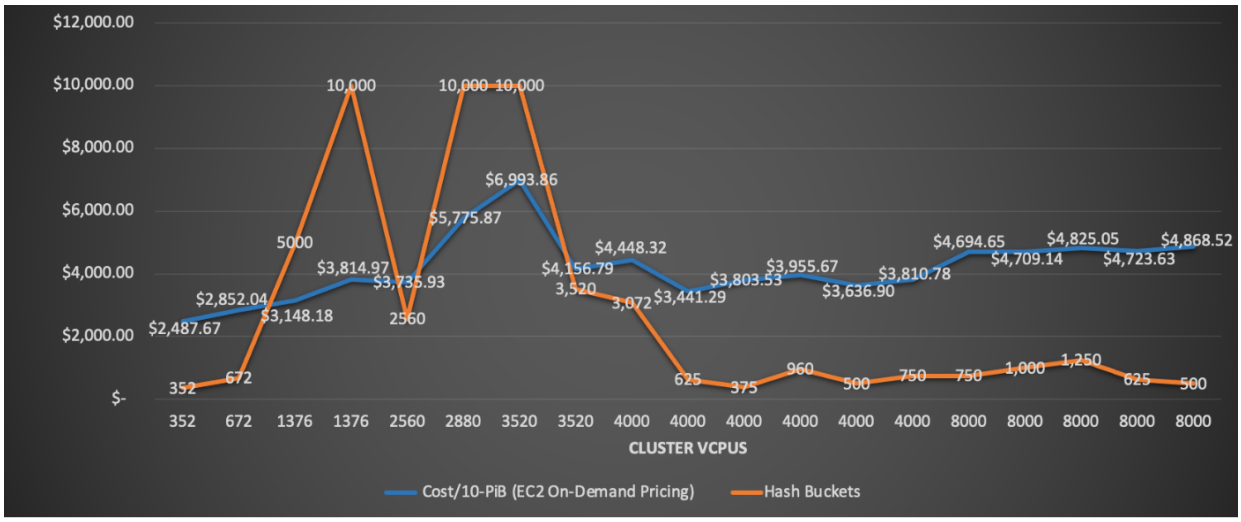
HORIZONTAL SCALING: LATENCY

Figure 3-4: Job Run Latency vs. Cluster Worker vCPUs and Input Size



INPUT SIZE AND HORIZONTAL SCALING: COST EFFICIENCY

Figure 3-5: Cost vs. Hash Buckets and Cluster vCPUs



Appendix

1. FAQ

- Why is the Ray Compactor so much more efficient than the Spark Compactor?

    a. **Purpose-Built:** The ability to write a custom, optimal algorithm end-to-end with Ray vs. accepting a generic map/reduce execution plan for Spark. While the code to do this operation in Spark is very succinct, the underlying implementation details that fulfill a generic “orderBy” and “drop” (duplicates) are far from an optimal implementation of deduping with sort keys. A few examples:

        i. **Record Pointers and Lazy Materialization**: Our Ray implementation sends minimally-annotated 32-byte record pointers between cluster nodes whenever possible and only materializes records when absolutely required. The Spark compactor is always passing fully materialized record data between nodes. For our production tables, this means Spark is typically sending ~1KiB between nodes for every ~32 bytes we send between nodes.

ii. **Replace O(nlogn) Sort --> Dedupe w/ O(n) Dedupe w/ Sort Key Comparator:** The Spark compactor inefficiently projects redundant sort values (like the *partition stream position* of all records in an S3 file) across every record of the dataset and runs an O(nlogn) sort by these columns to find the correct duplicate records to keep. Where possible, we pull repeated sort values up to larger record groups sharing the same sort value, and find the correct duplicate records to keep in O(n) (by just comparing the *sort keys* for an existing duplicate to see if a conflicting record is kept or discarded).

iii. **Column-Oriented**: When materialization of record pointers is required, our Ray implementation leans heavily on the column-oriented layout of Parquet and Arrow to only materialize the columns required for any operation and to maximize memory locality when reading successive fields from these columns.

iv. **Compute Caching**: We save a primary key index in S3 for any table that we compact. This improves the performance of subsequent job runs against the same table by leveraging cached SHA-1 digest computation and shuffle results from prior job runs.

v. **Minimize Type Casting and Data Copies:** We deal with the data in terms of bytes whenever possible, and only cast to Python, Arrow, or NumPy data types or copy data when absolutely required. The Spark compactor casts all data eagerly into its DataFrame types (although a more optimal Spark implementation could theoretically be written that does not do this).

b. **Ray Scheduler Performance:** Near zero-overhead from Ray's task scheduling framework. Task scheduling and gathering of results from completed tasks consistently takes under 2% of the total end-to-end execution time on Ray.

c. **Ray Zero-Copy Object Store**: Task inputs and outputs in Ray utilize zero-copy memory reads/writes wherever possible (e.g. when task 1 on Node A writes output that will also be used as input to task 2 on Node A).

d. **Delegate Heavy-Lifting to C++:** Lightweight Python Ray and Arrow layers that delegate all heavy-lifting (e.g. per-row operations and memory management) to highly optimized C++ layers vs. the Spark backend which relies heavily on the JVM.