Smart Checkpointing for the Qbeast OTree Index

by

Grgur Dujmovic

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday July 15, 2026 at 15:30.

Student number:	5204828	
Project duration:	8 months	
Thesis committee:	Dr. A. Katsifodimos,	TU Delft, supervisor
	Dr. G. Christodoulou,	TU Delft, supervisor
	Dr. J. Decouchant,	TU Delft, chair
	Dr. J. Yang,	TU Delft

TU Delft

Abstract

Qbeast is an open-source extension to Apache Spark and Delta Lake that maintains a multidimensional OTree index over columnar tables, enabling approximate query processing through file pruning. Each time a table is queried, the system constructs an *IndexStatus*, a compact in-memory map from cube metadata. It does so through two sequential steps: replaying the Delta Lake transaction log to enumerate the active file set ($O(D)$ in log depth D), then running a distributed `groupBy` aggregation over all active block metadata ($O(N)$ in active file count N). The log replay cost is paid once per relation, the aggregation is rerun on every query against the same relation. Delta Lake's periodic checkpoint compaction caps the log-replay cost at a bounded overhead; the per-query $0(N)$ aggregation is the gap that smart checkpointing fills.

This thesis introduces *smart checkpointing*: a persistent cache for the IndexStatus, saved as a K-row Parquet file (where K is the number of OTree cubes, $K \ll N$) and written automatically after each batch of writes. Further queries load the snapshot and replay the G commits that have accumulated since the snapshot. This reduces the per-query reconstruction cost from $O(N)$ to $O(K + \alpha G)$. An incremental merging algorithm handles the common pure-append deltas and is equivalent to the full index rebuild. A closed-form cost model is defined, which gives an optimal snapshot interval $G^* = \sqrt{F_{\text{write}} \cdot C_{\text{dump}}/(F_{\text{query}} \cdot \alpha)}$, providing a workload-based tuning rule.

Experiment 1 establishes the baseline $O(N)$ coefficient $E_{\text{shuffle}} = 0.353$ ms/file ($R^2 = 0.950$) over a wide range of active file counts. Experiment 2 validates the smart path with $\alpha \approx 21$ ms/commit ($R^2 > 0.99$), confirms that α is independent of table size across a range of N, and derives $G^* \approx 10$ commits. Experiment 3 validates the amortized cost model under mixed read/write workloads. For read-heavy workloads, total operational cost is reduced by 45% at $G = 500$. Experiment 4 emulates a deployment on a distributed cluster by adding a per-file latency. With a representative latency of $E_{\text{net}} = 20$ the emulated speedup is up to $19\times$, amplifying the advantage of smart checkpointing. The final experiment replicates two experiments on real-world datasets, to confirm values from synthetic experiments.

The central finding of the thesis is the transformation of the index reconstruction cost from a full data history rebuild to a snapshot and delta load. The practical benefit starts with approximately 30 prior commits in cold-start scenarios, and grows with a big table in a distributed deployment.

Contents

1 Introduction

1.1. Context and motivation

Modern organization dealing with data-intensive operations store and process ever-growing datasets using data lake architectures. These are large collections of raw files, stored on various object storage solutions (Amazon S3, Azure Data Lake Storage, Google Cloud Storage) and queried in-place by various distributed processing frameworks. Apache Spark (Zaharia et al., 2010) is the dominant runtime for such workloads, exposing a structured DataFrame API compiled by the Catalyst query optimizer into parallel execution plans dispatched across a cluster. Delta Lake (Armbrust et al., 2020) adds ACID transaction semantics, schema enforcement, and versioned history to Parquet tables stored on object storage, enabling reliable concurrent reads and writes without a separate metadata server.

In this ecosystem of data storage, there is a category of queries that do not need exact answers. Several fields, such as exploratory analytics, monitoring dashboards and machine learning feature pipelines, can all be served by approximate query processing (AQP). The benefit of approximate queries is in returning statistically representative samples of the queries data in milliseconds rather than waiting for a full table scan, which can take minutes over hundreds of gigabytes (Chaudhuri et al., 2017). The challenge for AQP systems is delivering meaningful accuracy guarantees, while also requiring as little I/O as possible towards the storage layer.

1.2. Qbeast-spark OTree index

Qbeast-Spark (Qbeast Analytics, 2022) addresses this challenge with a multidimensional **OTree** index with every Delta Lake table. The OTree is a hierarchical spatial partitioning of the normalized data domain. It has nodes called cubes, each covering a hyper-rectangular region of a d-dimensional unit hypercube and storing the records whose coordinates fall in that region, and whose pseudo-random weight falls below a cube-specific threshold. Sampling queries at fraction f are answered by traversing the tree and reading only the cubes that are guaranteed to contain the requested fraction of records. This removes the large majority of Parquet files from the scan entirely. The connection between the spatial index and probabilistic sampling relies on assigning a deterministic pseudo-random weight to each record when they are ingested. This weight is computed as a hash of the indexed columns of the record. The weights are distributed pseudo-uniformly over the integer range, meaning that a query requesting a fraction f of lightest-weight records is equivalent to a uniformly random f-fraction sample. The OTree organizes records in such a way that the weight-based querying maps precisely onto a set of cubes (i.e. a set of Parquet files) so that it doesn't read a record that the sample does not include.

1.3. The index reconstruction bottleneck

For Qbeast to traverse the OTree, the runtime must compute `IndexStatus`. This is a compact map from cube identifiers to each cube's weight threshold and element count. The `IndexStatus` is not persisted, it is reconstructed by aggregating the block metadata carried in each active Parquet file's Delta Lake `add` record. The reconstruction has two sequential processes. Firstly, the Delta transaction log is replayed to enumerate which files are currently active. This log replay occurs once per relation

and its cost is $O(D)$ in log depth D. Secondly, the block metadata of all N active files is aggregated by cube with a distributed Spark `groupBy`. The most important part is that this aggregation has no in-memory cache. The `loadIndexStatus` function reruns on every query against the same relation. Delta Lake's periodic checkpoint compaction reduces the log replay cost, but it doesn't help the $O(N)$ aggregation cost. For a production table accumulating hundreds of commits with thousands of files, this aggregation would dominate the query latency.

1.4. Smart checkpointing as the solution

The IndexStatus is a deterministic function of the active file set, and the file set changes incrementally. If the IndexStatus is serialized to a Parquet snapshot at regular intervals, subsequent reconstructions just have to load that snapshot and replay the accumulated commits since it was written. The cost is then proportional to the number of cubes K plus the number of recent commits.

This thesis designs, implements and evaluates a smart checkpointing extension to Qbeast-Spark. The mechanism introduces dedicated components for serializing and loading index snapshots, modifies the IndexStatus reconstruction path to use a three-way dispatch (direct snapshot load, snapshot plus incremental replay, or full rebuild as a fallback), and integrates the automatic snapshot dumping into the write path. Correctness of the system is maintained in two ways. Firstly, we prove that incrementally merging pure-append deltas produces output identical to a full rebuild from the combined file set. Secondly, we prove that incrementally updating the index does not disturb the uniform distribution of the sampling.

The thesis is carried out as a contribution to the open-source qbeast-spark project, on the forked `index_delta_snapshot` branch. All implementation changes are backward-compatible with existing tables.

1.5. Research questions

The thesis is based on these three research questions:

RQ1. Does smart checkpointing reduce the cost of index reconstruction, and how does this benefit vary with table scale, log history depth, and deployment environment?

RQ2. Does smart checkpointing preserve the sampling correctness guarantees of the Qbeast index across all supported table operations?

RQ3. Can the optimal snapshot interval be derived analytically from system parameters, and does the resulting cost model accurately predict the empirically observed performance?

1.6. Thesis structure

Chapter 2 (Background) introduces the technological foundations: the Apache Spark execution model, Delta Lake's transaction log and checkpoint mechanism, the Parquet columnar format, and the principles of approximate query processing. It provides a conceptual overview of the Qbeast-Spark OTree index and concludes with a precise statement of the reconstruction bottleneck that motivates this work.
Chapter 3 (Related Work) surveys prior work in approximate query engines, index checkpointing and metadata caching in data lake table formats, and incremental view maintenance as a general technique for avoiding full recomputation.
Chapter 4 (Implementation) presents the full design and implementation of the smart checkpointing system: the new serialization and loading components, the three-way dispatch in `loadIndexStatus`, the incremental merging algorithm with its correctness proof, the cost model, and the automatic snapshot dump integrated into the write path.
Chapter 5 (Experimental Evaluation) describes the experimental methodology and presents the results of five experiments to validate the cost model. The experiments show a characterization of the $O(N)$ nature of the baseline implementation, a model validation and optimal gap derivation, determining the amortized cost with different workloads, a cluster emulation and application to real-world datasets.
Chapter 6 (Discussion) interprets the results against the research questions, characterizes the operating regimes of smart checkpointing, derives practical deployment rules, and identifies limitations and future work.
Chapter 7 (Conclusion) summarizes the contributions and findings.

2

Background and State of the Art

2.1. Log-Structured Storage and Delta Lake

Data lake tables stored on object storage are treated as immutable. Instead of modifying an existing Parquet file, each write creates new Parquet files, recording metadata changes. Therefore, the current state of the table is not explicitly stored anywhere. The state of the table must be derived by processing a sequence of file additions and deletions. Delta Lake (Armbrust et al., 2020) imposes structure on that sequence through a transaction log.

The transaction log

In Delta Lake, every write is stored as a JSON commit file in a local directory close to the table data. Each commit file lists the files added and files logically removed from the table (`AddFile` and `RemoveFile`). To read the current table state, we have to replay this log, which is $O(D)$ in log depth D, the number of prior commits.

This very transaction log is the mechanism behind Delta Lake's ACID guarantees. Writers write concurrently by agreeing on a linear commit order, attempting to write sequentially ordered commit files. If another writer commits first, it is retried after resolving possible conflicts. Readers see a consistent snapshot that corresponds to a specific commit number, regardless of writes in progress.

Physical checkpointing

Tables accumulate commits, so the transaction log grows. This makes replaying the full log on every read expensive. Delta Lake periodically compacts the transaction log into a Parquet checkpoint file to avoid reading the whole transaction log. A reader that starts from a checkpoint only needs to replay the commits that are after it (the snapshot "gap"). This reduces the amortized log-replay cost from $O(D)$ to a bounded number of reads.

This mechanism is structural basis for smart checkpointing. Delta lake uses checkpoints for the active file set so readers don't have to replay the whole transaction log. Smart checkpointing saves the OTree index so a reader doesn't have re-aggregate the full active file set. Delta Lake checkpoints store one row for each active file, while Qbeast index snapshots store one row per cube, a cross-file aggregated statistic.

2.2. Approximate query processing

Approximate query processing exists to get a representative result fast. The focus is on speed, not absolute correctness. For many different analytical tasks, like aggregation, trend detection and exploratory data analysis, an answer derived from a representative sample is practically equivalent to an exact result (Chaudhuri et al., 2017).

The simplest AQP mechanism is Bernoulli sampling: each record is independently included with probability f, giving an unbiased f-fraction of the full dataset. The I/O cost is $O(N)$ in practice because every file may contain selected records.

Qbeast has a different approach to AQP. Each record is assigned a deterministic pseudo-random weight when ingested. The records are saved in the OTree scuh that low-weight records concentrate in the same spatial cubes. A sampling query for fraction f reads only the cubes whose weight threshold is at least f, skipping all files that contain only high-weight records. The I/O cost is proportional to f rather than to total table size, with statistical guarantees equivalent to Bernoulli sampling.

2.3. Multidimensional indexing

Multidimensional indexes are important in modern analytical workloads, because it is often required to use predicates on two or more columns, e.g. findings GPS locations in a bounding box within a time frame. Naively scanning a columnar table is extremely expensive at scale. This can be remedied by using a multidimensional index structure which organizes the data so only relevant portions need to be read.

R-trees

The R-tree (Guttman, 1984) is the canonical multidimensional index for point and range queries. It works by grouping d-dimensional objects into nested minimum bounding rectangles. A query specifies a target rectangle, and it is found by descending the tree and pruning every sub-tree with a rectangle that does not intersect the query region, without reading the containing objects. The R*-tree (Beckmann et al., 1990) improves on this by optimizing the node splits and insertions to minimize the rectangle overlap, reducing the number of nodes visited per query. These structures are very efficient for in-memory or disk-resident random-access storage.

In a data lake context, R-tree variants cannot be applied directly. Data resides in immutable Parquet files on object storage and gets processed by distributed Spark jobs. This means there is no in-place update, no pointer-based tree navigation, and the unit of access is an entire Parquet file rather than an individual record. There is a file-level analog of the MBR idea Parquet footers. To allow the query planner to skip irrelevant files, each row group stores per-column minimum and maximum values, skipping files whose column range does not overlap the query predicate. This is called predicate pushdown, and it is the mechanism that Delta Lake and Apache Spark use for range-query file pruning.

Space-filling curve ordering

A practical alternative for distributed columnar storage is to sort data by a space-filling curve value computed from the indexed columns. Z-ordering (Morton ordering) (Morton, 1966) assigns each record a scalar key by interleaving the bit representations of its d coordinate values. Records that are spatially close in d-dimensional space receive similar Z-order keys and therefore land in the same or adjacent files after sorting. A range query on multiple columns can then skip large contiguous file ranges by comparing the query bounds against per-file column statistics, achieving multidimensional file pruning without a pointer-based index structure. Delta Lake supports Z-ordering via `OPTIMIZE ZORDER BY`; Databricks' Liquid Clustering applies an equivalent reordering incrementally as new data arrives.

The Hilbert curve (Hilbert, 1891) is an alternative space-filling ordering with better locality-preservation properties than Z-ordering for clustered data, though it is more complex to compute and less widely implemented in data lake tooling.

The sampling gap

Both the data skipping and curve ordering minimize the number of files scanned per predicate query. However, neither of them provide guarantees about sampling probability. There is no way to identify which files contain the lowest-weight f-fraction of records given an approximate query at fraction f of a multidimensional range. This gap motivates the Qbeast OTree index described in the following section, which encodes a per-cube weight threshold so that a sampling query can determine the exact set of files needed to satisfy fraction f without reading any record that the sample would not include.

2.4. Qbeast overview

This section describes the architecture of the QBeast (Qbeast Analytics, 2022) indexing system I will be using. It is an open-source indexing framework designed to accelerate querying large multidimensional datasets in distributed data lakes. The core of QBeast is the OTree. The OTree is a hierarchical

multidimensional index structure, directly analogous to a d-dimensional generalization of a binary tree over a normalized hypercube $[0, 1]^d$. Each node in the OTree is called a *cube*.

Spatial Partitioning and Cubes

The root cube covers the entire hypercube. Each cube at depth k covers exactly one of the 2^{dk} equal-volume hypercubic cells produced by bisecting each dimension k times. A cube at depth k has 2^d children, each covering one of the 2^d equal-volume sub-regions obtained by bisecting the parent region along every dimension simultaneously. The tree branches by a factor of 2^d at each level. For $d = 2$, the structure is a quadtree, for $d = 3$ it is an octree, and so on for higher dimensions.

A cube has a unique identifier `CubeId`, which stores three fields: the dimension count d, the depth k and the bit-mask encoding of the path from the root. The bit-mask allocates d bits at level i, one for each coordinate axis, to determine wether the left (0) or right (1) hypercube half was selected. A cube at depth k is uniquely identified by bit-mask with $d \cdot k$ bits. This exact representation allows for the deterministic and efficient computation of the cube containing an arbitrary x by extracting the leading k bits of each coordinate. These cube identifiers support efficient serialization to a compact base-64 string encoding, which is the form stored in the Delta Lake transaction log alongside each data file. The ordering on `CubeId` is consistent with a depth-first, left-to-right traversal of the tree, which allows the map of cube statuses to be stored as a `SortedMap` that admits efficient range iteration, a property exploited during query execution.

Normalization and revisions

The raw column values are mapped into the unit hypercube before spatial operations take place. This is handled by the `Transformation` layer. Each indexed column has an assigned `Transformer` object defining the normalization strategy for its data type. There are normalization strategies for numeric columns, strings and arbitrary data types, which use a deterministic hash function to map any value uniformly to $[0, 1]$. The indexed columns, their transformers, the transformation parameters fitted to the data (e.g. the observed minimum and maximum of numeric columns), and the target cube size are collected into an immutable `Revision` object. When new data arrives, if its distribution is outside of the range representable by current transformations (e.g. a new numeric maximum), a new revision is created with updated transformation parameters. The prior revisions are saved and remain queryable within the same Delta Lake table. Each revision maintains an OTree index over its normalized space. The design allows the OTree guarantees to hold per revision: queries are issued against the most recent revision.

Weights and record routing

One of the core ideas of Qbeast is that each data record has a *weight* assigned to it. The weight is simply a 32-bit signed integer in the range [`Int.MinValue`, `Int.MaxValue`]. It is a deterministic hash of the indexed columns using a salted Murmur3-based hash, generating a pseudo-random integer that is effectively independent across records. Due to the hash being deterministic and dependent only on the content of the indexed columns, the same record always has the same weight. The weight is linearly interpolated to a fraction in $[0, 1]$ via

$$f(w) = \frac{w - \texttt{Int.MinValue}}{\texttt{Int.MaxValue} - \texttt{Int.MinValue}},$$

so that $f(\texttt{Int.MinValue}) = 0$ and $f(\texttt{Int.MaxValue}) = 1$. Since the hash is pseudo-uniform, the fractions of all records in a large dataset are approximately uniformly distributed on $[0, 1]$.

Each cube in the OTree stores all records whose normalized point falls within the cube's spatial region *and* whose weight falls below a cube-specific threshold called the *maxWeight*. The key invariant is:

> *A record with weight* $\mathbf{w}$ *and normalized point* $\mathbf{x}$ *is stored in the shallowest cube on the root-to-leaf path containing* $\mathbf{x}$ *whose maxWeight is at least* $\mathbf{w}$.

If there is no cube on the path with a high enough maxWeight, the record goes lower to an arbitrarily deep leaf. The maxWeight of a cube is chosen such that approximately `desiredCubeSize` records

are accepted by each cube, ensuring that the tree self-balances with roughly equal-sized nodes. Since the weight is computed from the content of the indexed columns, the routing decision is independent of the order of insertion. Two successive ingestions of the same logical dataset, with the same revision and cube weights, produce identically routed records. This is essential for the correctness of incremental index builds.

Index construction

Building the OTree over a distributed dataset requires determining how many records each cube will get and what its `maxWeight` should be. This has to be done before the records are written because the routing decision depends on the cube weights. To handle this, the system uses a two-pass approach implemented in `DoublePassOTreeDataAnalyzer`. In the first pass, a local OTree is built on each Spark partition using a scaled-down group cube size to ensure it fits in memory. Each local OTree produces (CubeId, partial domain) pairs, aggregating them across partitions to compute global cube domains. In the second pass, the global cube domains are used to compute the maxWeights for each cube, and broadcasting them to all executors. Finally, each record is routed to its target cube using `PointWeightIndexer` and written into the corresponding Parquet block.

The domain of a cube is a real-valued quantity that measures how many records from the full dataset ”pass through” the cube. It counts the records stored in the cube and all records in its subtree.The cube domain estimates the expected number of records a cube domain estimates, in expectation, how many records the cube would hold if it were expanded to the full desired cube size. The cube domain is the key intermediate quantity linking the observed data distribution and the per-cube maxWeights.

In the distributed case, each Spark partition builds a local OTree over its portion of the data using a scaled-down group cube size $g = \texttt{desiredCubeSize} / \max(\texttt{numPartitions}, N/\texttt{bufferCapacity})$, where N is the total record count and `bufferCapacity` is a configurable memory limit. This ensures the local OTree is small enough to fit in memory. Each local OTree contributes a set of (CubeId, partial domain) pairs, which are then grouped by CubeId and summed across all partitions to produce the global cube domains. This `groupBy` is the most expensive part of the index building.

The domain of a cube c is computed bottom-up from the local OTree: the tree size of cube c (the number of records in c and all its descendants) is divided by $(1 - \bar{W}_{parent(c)})$, where $\bar{W}_{parent(c)}$ is the normalized weight of c's parent in the local OTree. This correction accounts for the fact that only a fraction $1 - \bar{W}_{parent}$ of the records that reach the parent level overflow to children; the domain inflates the observed tree size to estimate the ”true” throughput.

Incremental appends

Appending data to an existing table does not require the index to be rebuilt. Instead, the current cube weights are used as the starting point and only the cubes that require more capacity are extended. Existing cubes continue to hold the unchanged records. This preserves the location of the existing data files. The routing invariant is preserved because a record in cube c will remain in c as long as the maxWeight of c does not decrease below the record's weight.

The IndexStatus data structure

The `IndexStatus` class is the runtime representation of the OTree. It is defined as (`revision`, `cubesStatuses`) where `cubesStatuses` is a `SortedMap[CubeId, CubeStatus]`. Each `CubeStatus` entry stores:

- **cubeId** - the unique cube identifier
- **maxWeight** - the integer weight threshold below which records are accepted by this cube
- **normalizedWeight** - the floating-point saturation ratio $\bar{W}_c \in [0, 1]$, derived from maxWeight or from element count
- **elementCount** - the total number of records stored in this cube across all physical Parquet files

The `IndexStatus` is stateless with respect to the physical data layout. Each physical Parquet file is associated with an `IndexFile` object. It has a list of `Block` records, with each `Block`, naming

the cube it belongs to and the weight range of the contained records. The `IndexStatus` is derived from *all* blocks by grouping on cube, taking the minimum `maxWeight` across all blocks of the cube and summing the element counts. It produces a deterministic result for a set of physical files.

In the original implementation, the `IndexStatus` is recomputed on every query. This involves loading all `AddFile` entires from the Delta Lake transaction log for the current revision, computing a separate `IndexFile` for each and aggregating them. The cost of this operation grows linearly with the number of Delta commits (the log depth), because each commit can add new files and the full log must be replayed to enumerate the current active file set.

Sampling queries in QBeast

A Qbeast sampling query specifies a fraction $f \in (0, 1]$. Spark recognises a `DataFrame.sample(f)` call through `SamplingRule`, which injects a `QbeastMurmur3Hash` weight filter into the query plan. `DefaultFileIndex.listFiles()` detects this filter and hands it off to `SamplingListFilesStrategy`, calling `QueryExecutor.executeQuery()`, which then calls `loadIndexStatus` to obtain the current cube map.

The weight threshold for fraction f is $W_f = f \times$ `Int.MaxValue`. Since record weights are pseudo-uniformly distributed, all records with weight $\leq W_f$ form an unbiased f-fraction sample of the full dataset. The OTree routing invariant guarantees that these records are stored in exactly the cubes whose `maxWeight` is at least W_f. `SamplingListFilesStrategy` therefore returns only the files belonging to those cubes, and the Spark executor reads no file that the sample would not include.

An accurate and current `IndexStatus` is required for correct sampling. If it is stale, the cube set returned for a given f may be wrong, producing a biased or incorrectly sized sample. The smart checkpointing mechanism introduced in Chapter 4 keeps the `IndexStatus` current at cost $O(K + \alpha G)$ rather than $O(N)$, without affecting this guarantee.

The sampling guarantee is probabilistic. The expected number of records returned by a query at fraction f is $f \cdot N$, where N is the total record count for the revision. This follows from the pseudo-uniform distribution of weights and the routing invariant. The actual record count may deviate slightly from $f \cdot N$ due to two sources of approximation: finite dataset size (the weight distribution is pseudo-uniform but not perfectly uniform) and cube boundary effects (records near a cube boundary may have slightly correlated spatial and weight values). In practice these deviations are small, and the system is designed as an approximate query processor rather than an exact one.

3

Related work

This chapter covers three areas connected to this thesis: approximate query processing, data lake metadata management and incremental view maintenance. The gap for each area is identified.

3.1. Approximate query processing

The tradeoff in APQ is result accuracy for query latency. BlinkDB (Agarwal et al., 2013) maintains a multi-dimensional stratified sample store and routes each query to the smallest sample that satisfies the user's error and latency constraints. Verdict (Park et al., 2018) takes a similar approach but adds proactive error estimation on top of pre-computed samples. AQUA (Acharya et al., 1999) pre-computes a sample library off-peak and selects the best sample at query time. A key property in all three systems is that the samples are independent of the physical data layout. Because of this independence, none of them need an index structure, so there is no reconstruction cost.

Qbeast works differently. The sampling probability is encoded into the physical data layout through the OTree weights. A query at fraction f only reads files from cubes that are guaranteed to contain f-weighted records. This layout-dependent design achieves better spatial locality than random sampling, but it requires the IndexStatus to be current on every query. Smart checkpointing reduces that reconstruction cost from $O(N)$ to $O(K + \alpha G)$, which brings Qbeast-Spark's cold-start overhead closer to what layout-independent systems achieve by avoiding indexes entirely.

3.2. Data Lake Table Formats

There are three dominant open table formats. Each solve the problem of providing ACID semantics over cloud object storage. Their approaches to metadata management are relevant to this thesis.

Delta Lake

Delta Lake (Armbrust et al., 2020) maintains an append-only transaction log of JSON commit files, as previously mentioned. Each commit records `AddFile` and `RemoveFile` actions, which together support snapshot isolation and time travel. To keep log replay fast, Delta Lake writes a checkpoint every ten commits by default. The checkpoint is a Parquet file with one row per active file, which reduces replay cost.

Delta Lake's checkpoint reduces log replay cost, but it does not reduce the per-query aggregation cost in Qbeast-Spark. The checkpoint stores one row per active file, hile Qbeast-Spark needs one aggregated row per cube across all files in the cube. Delta Lake does not store such application specific cross-file statistics. Smart checkpointing fills this gap by writing a separate K-row Parquet file that captures exactly the cube-level aggregation that is not available through Delta's checkpoints.

Apache Iceberg

Apache Iceberg (Blue et al., 2021) organises table metadata as a three-level hierarchy of manifest lists and manifest files. Each manifest records a subset of the table's data files together with per-file column

statistics such as lower and upper bounds. When querying, Iceberg is able to skip entire manifests by utilizing those bounds, which reduces the cost of file enumeration.

Again, these statistics are not the statistics we need saved. These are used for predicate push down, while Qbeast requires specific cross-file statistics. But Iceberg does show the value of separating file metadata from file data and caching it in a queryable Parquet format. Smart checkpointing applies the same idea to Qbeast's index metadata.

Apache Hudi

Apache Hudi (Apache Software Foundation, 2021) is built for record-level insert, update and delete workloads. It has a metadata table that caches the active file set per partition. The table is updated incrementally on every commit, avoiding costly object-store listing operations.

Hudi eliminates cloud storage list latency because it caches file membership. Meanwhile, smart checkpointing eliminates distributed aggregation latency by caching a derived index. The incremental update mechanism is architecturally similar in both cases. The key difference is that Hudi only needs to track set membership, while smart checkpointing must maintain a commutative aggregation merge.

3.3. Incremental view maintenance

Incremental view maintenance (IVM) is the problem of keeping a materialized view up-to-date while the base table is changing without recomputing it from scratch. Gupta and Mumick (Gupta & Mumick, 1995) establish that an aggregation view can be maintained incrementally if the aggregation function is *algebraically closed*. This means that the new aggregate can be derived from the old aggregate and the delta alone. Sum and count satisfy this but median and rank do not.

The Qbeast IndexStatus fits this framework. It requires aggregating the minimum over `maxWeight` and sum over `elementCount`. Both of these are distributive over set union. This is the formal basis for proving the correctness of `buildIncremental`. Merging a snapshot with a commit delta produces the same results as a full rebuild, because the aggregation decomposes cleanly over disjoint file additions.

DBToaster (Ahmad et al., 2012) and differential dataflow (McSherry et al., 2013) are IVM systems that maintain always-current views entirely in memory. Smart checkpointing has a different requirement. The derived structure must be serialised to a versioned Parquet file that a future Spark session can read with no shared state. The in-memory IVM literature does not address this cold-start persistence requirement.

The second gap in the research is about the refresh interval. IVM systems treat the interval as a trivial concern, either updating every write or just a free parameter. This thesis derives the optimal interval in 4.3 by minimizing total system cost based on workload. The result shows that the optimal gap depends on the dump cost and per-commit replay cost. This kind of closed form derivation is not in IVM literature.

The snapshot-and-delta structure is well known in database recovery. ARIES (Mohan et al., 1992) writes a checkpoint at a known log sequence number and then replays only the log suffix from that point forward. Smart checkpointing applies the same pattern to read-side index reconstruction in a stateless distributed environment, where the replay happens on every cold-start query rather than after a failure.

3.4. Summary

No prior work combines all four of the following properties: a persistent, versioned snapshot of a cross-file aggregate index; an incremental merge path with a formal correctness proof; integration with the Delta Lake commit protocol for snapshot lifecycle management; and a closed-form cost model for optimal snapshot interval selection. The data lake table formats show that file-level metadata caching is a solved problem. IVM theory provides the formal basis for the incremental merge. ARIES provides the structural template. Smart checkpointing brings these three ideas together and applies them to the specific problem of Qbeast-Spark index reconstruction.

4

Design and implementation

This chapter describes the smart checkpointing implementation introduced on top of the Qbeast indexing system. It explains the problem, formalizing the optimal snapshot interval, describes the implemented solution and finishes with correctness proofs and trade-offs.

4.1. Unbounded Reconstruction Cost

Every call to `loadIndexStatus` (sampling query or a write operation), calls `IndexStatusBuilder.build()`. This performs a distributed Spark aggregation over all N active Parquet files in the revision. As the table grows, N grows and so does the reconstruction latency, which means that the cost is $O(N)$. We can lower the cost of the reconstruction by implementing a **snapshot** system supported by an **incremental** delta.

The `IndexStatus` is deterministic because `build()` always returns the same map for the same set of active files. This means that the result can be cached. Smart checkpointing periodically serializes the K-row IndexStatus map to a Parquet snapshot file. On further calls, the full-scale aggregation of `build()` can be skipped by loading the K-row snapshot and replaying G commits that have accumulated since the snapshot was written. This leads to the cost model $T_{\text{smart}}(G) = C_{\text{snap}}(K) + \alpha G$, where $C_{\text{snap}}(K)$ is the cost to load the K-row snapshot and α is the per-commit replay cost. In design, K (number of distinct cubes) is smaller than N (number of active files) for large, mature tables. Nevertheless, K grows as the dataset grows, same as N, meaning that C_{snap} scales accordingly and may be comparable to $T_{\text{baseline}}(N)$.

4.2. The five changes

The smart checkpointing introduces three new components and modifies two existing ones. These components reduce the theoretical cost of `loadIndexStatus` from $O(N)$ (proportional to the number of active Parquet files) to $O(K + G)$ (proportional to the number of cubes plus the gap since the last snapshot), for the common case of pure-append workloads. This is done by replacing the $O(N)$ full shuffle aggregation with a snapshot load.

1. **`IndexStatusDumper`** *(new component, `delta/`)*. Serializes the in-memory `cubeStatuses` map to a Parquet file. Called after every successful write after the *gap* threshold is reached, and after an `OPTIMIZE` or `DELETE` operation.
2. **`IndexStatusLoader`** *(new component, `delta/`)*. Discovers the most recent snapshot whose version is $\leq$ the current Delta version via a filesystem listing, then deserialises the Parquet rows back into a `SortedMap[CubeId, CubeStatus]`.
3. **Three-way dispatch in `loadIndexStatus`** *(modified)*. Checks `IndexStatusLoader` before falling back to the full rebuild. Returns the snapshot directly if it is current: calls `buildIncremental` if the snapshot is stale and the delta is a pure append; performs a rebuild if there are `RemoveFile` actions, and for tables without any snapshot.

4. **`buildIncremental` in `IndexStatusBuilder`** *(new method, `core/`)*. Merges the block statistics from newly added files into a base `IndexStatus` without re-reading any historical files. Correctness follows from the distributivity of $\sum$ and $\min$ over disjoint multiset union (proved formally in §4.5).
5. **Automatic dump trigger in `IndexedTableImpl.write`** *(modified, `src/`)*. After every successful write, compares the current Delta version to the latest snapshot version. If the gap ≥ `SNAPSHOT_DUMP_INTERVAL` calls `IndexStatusDumper.dump`.

Additionally, Delta Lake, which Qbeast is built on top of, already has a periodic checkpoint mechanism. These checkpoints address different optimizations and are not redundant. By default, Delta Lake creates a checkpoint file every 10 commits. The checkpoint file is a Parquet file that has one row per active `AddFile` commit. In total this is N rows, where N is the number of active data files of the table. The purpose of it is to bound the cost of the Delta log replay. Instead of replaying all JSON commit entries, the reader can start from the most recent checkpoint and replay only the latest entries. Therefore, the Delta checkpoint limits the $O(D)$ log-replay cost. It does not limit the $O(N)$ aggregation cost from Qbeast, because this checkpoint contains all N `AddFile` rows and the `groupBy(cubeId)` aggregation must still process all of them to produce the `IndexStatus`. On the other hand, the Qbeast index snapshot is stored separately from the Delta log and contains one row per *cube* in the OTree. It has K rows, where K is the number of cubes in the OTree, depending on scale parameters and data distribution. The purpose is to cache the aggregation result.

Data Flow

Write path. `IndexedTableImpl.write` commits the Delta transaction, then checks the gap. If the gap threshold is met, it calls `loadIndexStatus` (which takes the cheapest available branch) and passes the result to `IndexStatusDumper.dump`, writing the snapshot to `_qbeast/index_snapshots/`.

Read path. `loadIndexStatus` calls `IndexStatusLoader.latestSnapshotVersion`. The return value determines the branch:

- **None** ⇒ `build()`: full $O(N)$ rebuild, identical to the pre-modification behaviour.
- $v_s = v_c$ ⇒ `IndexStatusLoader.load()`: direct Parquet read, cost $O(K)$.
- $v_s < v_c$, **no removes** ⇒ `IndexStatusLoader.load()` + `buildIncremental()`: cost $O(K + \alpha G)$.
- $v_s < v_c$, **removes present** ⇒ `build()`: full $O(N)$ rebuild.

4.3. Cost model and Optimal Snapshot Interval

The argument that `loadIndexStatus` cost grows with table size can be shown with a simple cost model. This model also provides the theoretical basis for the choice the the dump interval for the snapshots and clarifies the trade-offs. Let the following variables represent the system's operational costs and state:

- N: total number of active `IndexFile` objects in the revision (proportional to the number of active Parquet files).
- G: the *gap*, defined as the number of Delta commits that have accumulated since the last Parquet snapshot was written.
- E_{json}: the fixed driver-side cost to fetch and deserialise one Delta JSON commit entry from the transaction log.
- E_{shuffle}: the marginal distributed cost of merging one additional `IndexFile` into the `IndexStatus` — i.e., the per-file cost of the `groupBy(cubeId)` aggregation.
- E_{dump}: the average total cost of one snapshot write, including the full `build()` aggregation and Parquet serialization
- μ: average number of new `IndexFile` objects added per commit (files per write batch).

- F_{query}: frequency of `loadIndexStatus` calls per unit time (proportional to query rate).
- F_{write}: frequency of Delta commits per unit time.

The Three Operational Paths

Full Replay Cost (C_{full}) Without any Parquet snapshot the system must re-aggregate all N active files on every call:

$$C_{\text{full}} = E_{\text{shuffle}} \cdot N. \tag{4.1}$$

As N grows, C_{full} grows linearly and without bound.

Hybrid read (C_{gap}). With a snapshot at gap G, the reader loads the snapshot (C_{snap}) and then replays only the G JSON log entries and the $\text{files}_{\text{added}}(G) \approx \mu G$ new files. The cost for a pure-append delta (no file removals) is:

$$C_{\text{gap}}(G) = C_{\text{snap}} + G \cdot E_{\text{json}} + \mu G \cdot E_{\text{shuffle}} = C_{\text{snap}} + G \cdot \underbrace{\left(E_{\text{json}} + \mu \cdot E_{\text{shuffle}}\right)}_{\alpha}. \tag{4.2}$$

The coefficient α bundles the per-commit overheads. C_{gap} grows linearly with G, not with N: doubling the table size has no effect as long as G stays fixed.

Dump cost (C_{dump}). Every G commits, one snapshot is written. The amortised dump cost per commit is C_{dump}/G.

Optimal Snapshot Interval

The total system cost rate (cost per unit time) sums the continuous read penalty and the amortized dump penalty. Substituting C_{gap} from above:

$$C_{\text{total}}(G) = F_{\text{query}} \cdot C_{\text{snap}} + F_{\text{query}} \cdot \alpha \cdot G + \frac{F_{\text{write}} \cdot C_{\text{dump}}}{G}. \tag{4.3}$$

This is a trade-off where increasing G reduces dump frequency but raises per-read cost while decreasing G does the opposite. The term $F_{\text{query}} \cdot C_{\text{snap}}$ is constant in G and vanishes when differentiating, so it does not affect the optimal gap. Differentiating with respect to G and equating to zero:

$$\frac{d}{dG} C_{\text{total}}(G) = F_{\text{query}} \cdot \alpha - \frac{F_{\text{write}} \cdot C_{\text{dump}}}{G^2} = 0, \tag{4.4}$$

which gives the square-root formula for the optimal gap:

$$G^* = \sqrt{\frac{F_{\text{write}} \cdot C_{\text{dump}}}{F_{\text{query}} \cdot \left(E_{\text{json}} + \mu \cdot E_{\text{shuffle}}\right)}}. \tag{4.5}$$

Equal-frequency simplification. When every commit is followed by one read query ($F_{\text{query}} = F_{\text{write}}$) the frequency ratio cancels and the formula reduces to

$$G^* = \sqrt{\frac{C_{\text{dump}}}{\alpha}}. \tag{4.6}$$

The more general formula (4.5) applies when read and write frequencies differ. The equal-frequency case is a simplified formula used in the benchmark design to measure the read latency. The ratio C_{dump}/α determines the scale of G^*. When the dump cost is approximately two orders of magnitude larger than the per-commit cost, as in a Spark micro-batch workload where snapshots take seconds and a replay takes tens of milliseconds, then $G^* = \sqrt{C_{\text{dump}}/\alpha} \approx 10$.

Conclusions from G^* Equation (4.5) encodes three operationally relevant consequences:

- **Read-heavy workloads.** As F_{query} increases, G^* decreases. Each cached snapshot is amortized over more reads, so it pays to dump more frequently. In the limit $F_{\text{query}} \gg F_{\text{write}}$, $G^* \to 1$, we should dump a snapshot after every commit.
- **Write-heavy or expensive dumps.** As F_{write} or C_{dump} increases, G^* increases. Frequent dumping results in too many writes, meaning fewer snapshots at wider intervals are optimal.
- **Large tables.** N does not appear in G^*, meaning that the optimal snapshot interval is independent of table size. However, the cost of loading a snapshot C_{snap} scales with cube count K, increasing the cost.
- **Why α is N-independent.** The per-commit cost is $\alpha = E_{\text{json}} + \mu \cdot E_{\text{shuffle}}$. E_{json} is driver-side, single-threaded work (deserializing one JSON log entry). E_{shuffle} is distributed and decreases as parallelism increases. When E_{json} dominates (expected because driver-side JSON parsing is not parallelized by Spark) α is effectively constant across table sizes and cluster configurations. This fact, and the fact that G^* is independent of N, means that the optimal snapshot interval can be set once and remains valid if the workload remains constant.

Cluster deployment When the index files are in distributed object storage (S3, HDFS), each file read has an additional E_{net}. The legacy full rebuild access remote storage in two $O(N)$ steps. The first step is reading the Delta checkpoint Parquet file from the object store and the second is shuffling all N metadata records by `cubeId` across executors over the network. Since both steps are proportional to N, the final cluster penalty is $E_{\text{net}} \cdot N$, which updates the formula to:

$$C_{\text{full}}^{\text{cluster}} = (E_{\text{shuffle}} + E_{\text{net}}) \cdot N. \tag{4.7}$$

The smart path fetches one snapshot file from remote storage, and then applies G commits incrementally without a distributed shuffle. These commit items are Delta JSON log entries. While JSON commit files are much smaller than Parquet data files, they still require individual object-store round trips, each costing E_{net}. The total network penalty for the smart path is therefore one snapshot fetch plus G commit-log fetches:

$$C_{\text{gap}}^{\text{cluster}}(G) = (C_{\text{snap}} + E_{\text{net}}) + G \cdot (\alpha + E_{\text{net}}). \tag{4.8}$$

Since the snapshot fetch E_{net} is a constant term in G, it does not affect G^*. The per-commit network cost shifts α to $\alpha + E_{\text{net}}$, which lowers G^* slightly under high-latency storage — a conservative effect that does not change the qualitative recommendation.

Deploying remotely has a very different result than deploying locally. When deployed locally, C_{snap} scales with K, which grows with the data, so the local speedup is bounded by the per-commit efficiency ratio $E_{\text{shuffle}} \cdot \mu/\alpha$. Meanwhile, the structure of a cluster deployment allows for a much better speedup. Comparing (4.7) with (4.8) shows that the $E_{\text{net}} \cdot N$ baseline term is replaced by $(G+1) \cdot E_{\text{net}}$, yielding a speedup that grows approximately linearly with N for large tables.

4.4. Smart checkpointing implementation

The smart checkpointing is implemented with a snapshot + delta pattern. A complete `IndexStatus` is periodically serialized to the disk, with subsequent reads loading the snapshot and replaying the small number of Delta commits that have accumulated since the snapshot was written.

IndexStatusDumper

The `IndexStatusDumper` object serializes the `cubesStatuses` map of an `IndexStatus` to Parquet format. Given a base table path, a revision identifier, and the current Delta version number v, it writes a Parquet dataset to the appropriate local path. The dataset has four columns: a cube identifier string, the `maxWeight` integer, the normalized weight float and the element count integer. These are the fields of `CubeStatus`, so the snapshot is a proper serialization of the in-memory structure. To facilitate efficient discovery of the most recent snapshot without loading any Parquet data, the storage of the snapshots is hierarchical. Each revision has its own directory with each revision having its own snapshot versions.

IndexStatusLoader

The `IndexStatusLoader` object provides two functions. The first, `latestSnapshotVersion`, scans the revision-level directory for version subdirectory names matching the pattern `version=<integer>`, filters to those with version number $\leq$ the current Delta version, and returns the largest such version. This is a pure file system metadata operation and requires no data reads.

The second function, `load`, reads the Parquet snapshot for a given version, deserializing each row and assembling it back into a `SortedMap`. The cube identifier is reconstructed using the revision's `createCubeId`, maintaining dimensionality.

Modified loadIndexStatus

The `DeltaQbeastSnapshot.loadIndexStatus` method is modified to implement a three-way dispatch on the result of `latestSnapshotVersion`:

1. **No snapshot exists.** The method falls through to the original `IndexStatusBuilder.build()` call. The behaviour is identical to the unmodified code. This path is always taken on a freshly created table and ensures backward compatibility.
2. **Snapshot is current.** If the snapshot version equals the current Delta version, the snapshot is loaded directly from Parquet and returned. No Delta log entries are read, no Spark aggregation is performed, and the cost is proportional only to the number of distinct cubes in the revision.
3. **Snapshot is older than the current version.** The snapshot is loaded as a *base* `IndexStatus`, and the method calls `loadChangesBetween(revisionID, snapVersion + 1, currentVersion)` to obtain the Delta actions between the snapshot version and the current version. If the delta contains any removals (which occur during optimise and compaction operations), the method falls back to a full `build()` call. If the delta has only appends, the method calls `buildIncremental` to merge the new files into the base.

buildIncremental

The `IndexStatusBuild.buildIncremental` method takes an `IndexStatus` and a dataset of `IndexFile` objects that represent new files and creates an updated `IndexStatus`. It mirrors the aggregation in `indexCubeStatuses`, but it only operates on the delta files, merging the updated cube statistics in the base.

For each cube that appears in the delta, the total element count and the minimum maxWeight across the blocks is computed. These are combined with the base statistics for the cube. The element count is simply summed and the maxWeight is the minimum of the base and the new file. To calculate normalized weight we have to determine if the cube is saturated or not. If it is not, the normalized weight is recomputed from the combined maxWeight, and if it is, it is computed from the combined element count.

Automatic snapshot dumping

Snapshots have to be created regularly to be useful. If they were written on demand, there would be no benefit. The modified `IndexedTableImpl.write` method dumps a new snapshot after every write, given that the gap between the current Delta version and the latest snapshot version reaches the defined commit gap value `MetadataConfig.SNAPSHOT_DUMP_INTERVAL`. After the write transaction, if the gap is at least 10, `freshSnapshot.loadIndexStatus` is called to obtain a current `IndexStatus` via the cheapest branch, and the result is passed to `IndexStatusDumper.dump`. The snapshot therefore never lags more than `SNAPSHOT_DUMP_INTERVAL` commits behind the live table state.

4.5. Correctness proofs

A primary concern with modifying a system is if the functions of the system are preserved. The implementation satisfies all function requirements, as shown in this section.

IndexStatus idempotency derivation

Claim. Let v be any Delta version and let $\mathcal{F}(v)$ denote the set of active `IndexFile` objects at version v. The `IndexStatus` returned by `loadIndexStatus` under smart checkpointing is identical to

`build`$(\mathcal{F}(v))$ regardless of which dispatch branch is taken.

Proof. Let σ denote the aggregation function that maps a set of files to an `IndexStatus`: for every cube c,

$$\sigma(\mathcal{F})_c = \Big(\min_{b \in B(\mathcal{F},c)} b.maxWeight,\ \sum_{b \in B(\mathcal{F},c)} b.elementCount\Big)$$

where $B(\mathcal{F}, c)$ is the multiset of all blocks belonging to cube c across all files in $\mathcal{F}$. By construction, `build`$(\mathcal{F}(v)) = \sigma(\mathcal{F}(v))$.

The dispatch has four branches:

1. *No snapshot exists.* `build()` is called directly, computing $\sigma(\mathcal{F}(v))$.
2. *Snapshot at version $v_s = v$.* The snapshot was written by `IndexStatusDumper` immediately after a successful write at version v_s, by serialising the result of `loadIndexStatus` at that moment. By induction (base case: the snapshot was generated via a fresh evaluation path like branch 1 or 4), the stored values equal $\sigma(\mathcal{F}(v_s))$. Since $v_s = v$, returning the snapshot directly gives $\sigma(\mathcal{F}(v))$.
3. *Snapshot at $v_s < v$, pure-append delta.* Handled by Proof 3 (§4.5).
4. *Snapshot at $v_s < v$, delta contains removes.* `build()` is called, computing $\sigma(\mathcal{F}(v))$ directly.

In all branches the returned value equals $\sigma(\mathcal{F}(v))$. □

Equivalence of Incremental Merging

Claim. Let $v_s < v_c$ be two Delta versions such that no file removal occurs between them (pure-append delta). Let $S = \mathcal{F}(v_s)$ and $\Delta = \mathcal{F}(v_c) \setminus S$ (the files added strictly after v_s). Then

$$\texttt{buildIncremental}(\sigma(S),\ \Delta) = \sigma(S \cup \Delta).$$

Proof. Fix any cube c. Partition the blocks contributing to c at version v_c:

$$B(S \cup \Delta,\ c) = B(S, c) \uplus B(\Delta, c)$$

where $\uplus$ is disjoint union (no block belongs to both sets because $S \cap \Delta = \emptyset$ by the no-removal assumption).

Element count. *elementCount* is computed by summation:

$$\begin{aligned}
\sigma(S \cup \Delta)_c.elementCount &= \sum_{b \in B(S \cup \Delta,\, c)} b.elementCount \\
&= \sum_{b \in B(S,c)} b.elementCount + \sum_{b \in B(\Delta,c)} b.elementCount \\
&= \sigma(S)_c.elementCount + \sigma(\Delta)_c.elementCount.
\end{aligned}$$

`buildIncremental` computes exactly this sum.

MaxWeight. *maxWeight* is computed as the minimum over block maxWeights:

$$\begin{aligned}
\sigma(S \cup \Delta)_c.maxWeight &= \min_{b \in B(S \cup \Delta,\, c)} b.maxWeight \\
&= \min\Big(\min_{b \in B(S,c)} b.maxWeight,\ \min_{b \in B(\Delta,c)} b.maxWeight\Big) \\
&= \min\big(\sigma(S)_c.maxWeight,\ \sigma(\Delta)_c.maxWeight\big).
\end{aligned}$$

`buildIncremental` computes exactly this minimum.

New cubes. If cube c appears in Δ but not in S, then $B(S, c) = \emptyset$. `buildIncremental` handles this via the `getOrElse` defaults: $elementCount$ defaults to 0 (neutral for $+$) and $maxWeight$ defaults to `Weight.MaxValue` (neutral for $\min$). Hence $\sigma(S)_c.elementCount = 0$ and $\sigma(S)_c.maxWeight =$ `Weight.MaxValue`, and the formulas above reduce to the Δ-only values, which is correct.

Cubes absent from Δ. If cube c appears in S but not in Δ, then $B(\Delta, c) = \emptyset$ and the merge leaves $\sigma(S)_c$ unchanged, which is correct because no new blocks arrived for c.

Since both aggregation fields agree for every cube c, and both computations produce the same set of cubes, we conclude `buildIncremental`$(\sigma(S), \Delta) = \sigma(S \cup \Delta)$. □

Preserving sampling guarantees

The sampling guarantee of Qbeast is that a query at fraction f will return approximately $f \cdot N$ uniformly distributed records. This depends 2 conditions:

1. **The weight column in the Parquet files is a pseudo-uniform random variable** - this holds because the column is computed and written during data ingestion and is never modified.
2. **`IndexStatus` is accurate in showing which cubes contain records with weights below the threshold** - this holds because the `IndexStatus` created by smart checkpointing is semantically equivalent to the one produced by full rebuild, as argued in the previous sections. A query presented with an equivalent `IndexStatus` will traverse the same set of cubes and open the same set of Parquet files, returning the same result.

4.6. Known design trade-offs

The implementation makes some conscious design trade-offs between correctness, simplicity and performance. Each is a known limitation and a guide for future work.

Optimizations and deletes

Currently, when the delta has a `RemoveFile` action (these occur during `DELETE`, `OPTIMIZE`, `UPDATE` and compaction operations), the incremental path is not used and we fallback to the original `build()` path. The effective gap for the incremental path is therefore $\min(G_{\text{config}}$, optimise_frequency): a table that is optimised every 5 commits will trigger a full rebuild on every 5th read regardless of G. For tables with aggressive compaction schedules, the incremental path is rarely exercised and the snapshot provides only the Branch-2 (zero-delta) benefit. This is in line with what Qbeast is used as, an append focused index. This can be improved upon if there is a way to perform local cube invalidation quickly instead of rebuilding the index when a file is removed.

Asynchronous index dumping and caching

The dumping of the snapshot is done synchronously in the write path, immediately after committing the write transaction. The dump adds a full C_{dump} penalty to every G-th write. Dumping the snapshot asynchronously would eliminate this penalty. Implementing this would require version locking. Furthermore, in the current implementation a full C_{snap} penalty is paid when a snapshot is loaded. This can be avoided by caching the snapshot. This is left as future work.

4.7. Summary

The implementation takes the cheapest available path at every call to `loadIndexStatus`. First, a current snapshot is sought. If one exists at the current Delta version, it is loaded directly. If the snapshot is stale and the intervening delta is a pure append, the snapshot is loaded and only the gap commits are replayed incrementally. Otherwise, the system falls back to the original full rebuild. In the worst case the behavior is identical to the unmodified system. In the common case the cost is $C_{\text{snap}}(K) + \alpha G$ rather than $E_{\text{shuffle}} \cdot N$. Locally, C_{snap} grows with K (cube count), which scales with the data, so local speedup is bounded by the per-commit efficiency ratio $\mu \cdot E_{\text{shuffle}}/\alpha$. The dominant advantage is in cluster deployments where smart requires $(G + 1)$ network transfers (one snapshot fetch plus G commit-log fetches) instead of N.

5

Experiments and results

This chapter validates the smart checkpointing cost model through five focused experiments. Experiment 1 (§5.4) shows the $O(N)$ cost of the baseline `IndexStatus` reconstruction path to establish the performance problem and measuring the E_{shuffle} slope. Afterwards, experiment 2 (§5.5) measures the amortized total cost under mixed workloads and confirms the break-even formula. Experiment 3 (§5.6) validates the cost model for the smart path and empirically determines the optimal snapshot gap G^*. Going further, experiment 4 (§5.7) extends the model to emulate a distributed cluster deployment by adding network latency, quantifying smart's advantage in remote-storage environments. Finally, experiment 5 (§5.8) applies the cluster emulation model to real-world OpenStreetMap measurements to show cluster advantages on real data.

5.1. Experiment environment

All experiments were ran on a remote TU machine, and all timing was measured as the timing of the `loadIndexStatus` call with a Spark listener before the first query planning step. Table 5.1 gives the full hardware and software specification.

Table 5.1: Remote machine specification.

Property	Value
Hardware	
CPU model	2× AMD EPYC 7H12
Cores / socket	64 physical, 2 SMT threads
Logical CPUs	256 (across 2 NUMA nodes)
RAM	503 GiB
Storage	Local LVM volume (SSD-backed)
Software	
OS	Ubuntu 22.04.5 LTS
JDK	OpenJDK 11.0.31
Apache Spark	3.5, local mode (`local[256]`)
Spark heap	200 GiB driver, 8 GiB executor
Delta Lake	3.1
Scala	2.12

5.2. Tested implementations

Baseline implementation (`origin/main`) Enumerates the active file set from the Delta transaction log (log replay bounded by Delta's own checkpoint compaction) and then runs a full $O(N)$ `groupBy` aggregation on every `IndexStatus` reconstruction. Every call to `loadIndexStatus` invokes `new IndexStatusBuilder(this, revision).build()`.

Smart (`origin/index_delta_snapshot`) Uses the three way dispatch explained in §4.4, avoiding the full reconstruction cost

5.3. Data and distribution profiles

Synthetic datasets are used for the experiments, generated from the OpenStreetMap dataset at three scales: S with 1M rows (≈ 73 active data files), M with 10M rows (≈ 265 active data files) and L with 100M rows (≈ 4,700 active data files). Additionally these datasets are created in four distribution profiles:

- D1 **Uniform** Uniformly distributed coordinates. A balanced cube tree with approximately equal cube sizes at all depths.
- D5 **Power-law skewed** Coordinates sampled from a power-law distribution. A deep, sparse tree with heavily skewed leaf-node populations.
- D6 **Spatially clustered** Coordinates concentrated in a small number of tight geographic clusters. Dense spatial clusters separated by large empty regions.
- D8 **Mixed** Combination of skewed and uniform regions within the same dataset.

To test the implementation on a real-world scenario, two real-world datasets were used in Experiment 5 (§5.8. These datasets are OpenStreetMap Netherlands (OSM_NL, ~6M POI nodes) and OpenStreetMap Germany (OSM_DE, ~60M nodes).

5.4. Experiment 1 - Baseline characterization

In order to validate the cost model, first we have confirm that the baseline `IndexStatus` reconstruction time grows with the active file count N. A naive measurement would capture the $O(N)$ reconstruction and the data scanning step. To isolate just the reconstruction, a burn-in technique is used.

Design

New data is appended exclusively to the lower-left corner of the indexed space ($[0,\ 0.3]^2$), while all measurement queries target the diagonally opposite upper-right corner ($[0.65,\ 0.75]^2$). The Qbeast cube tree prunes the entire lower-left partition from the query plan, making the data-scan cost $O(1)$ regardless of table size. Only the reconstruction step, which enumerates all N active file metadata records to build `IndexStatus`, is sensitive to N.
The log depth D is varied from 1 to 5,000 at scale S. Each burn-in step appends 1,000 rows. This grows the active files N from 73 (at D = 1) to ≈5,000 (at D = 5,000).

Results

Figure 5.1 plots T_{baseline} against N for all trials at scale S An OLS regression gives:

$$T_{\text{baseline}}(N) = 0.353 \cdot N + 206 \quad \text{ms}, \quad R^2 = 0.950 \tag{5.1}$$

The fit is tight across the full 70× range. Residuals show no systematic non-linearity, confirming that the `groupBy(cubeId)` aggregation is the dominant cost term and that the $O(N)$ assumption is justified.

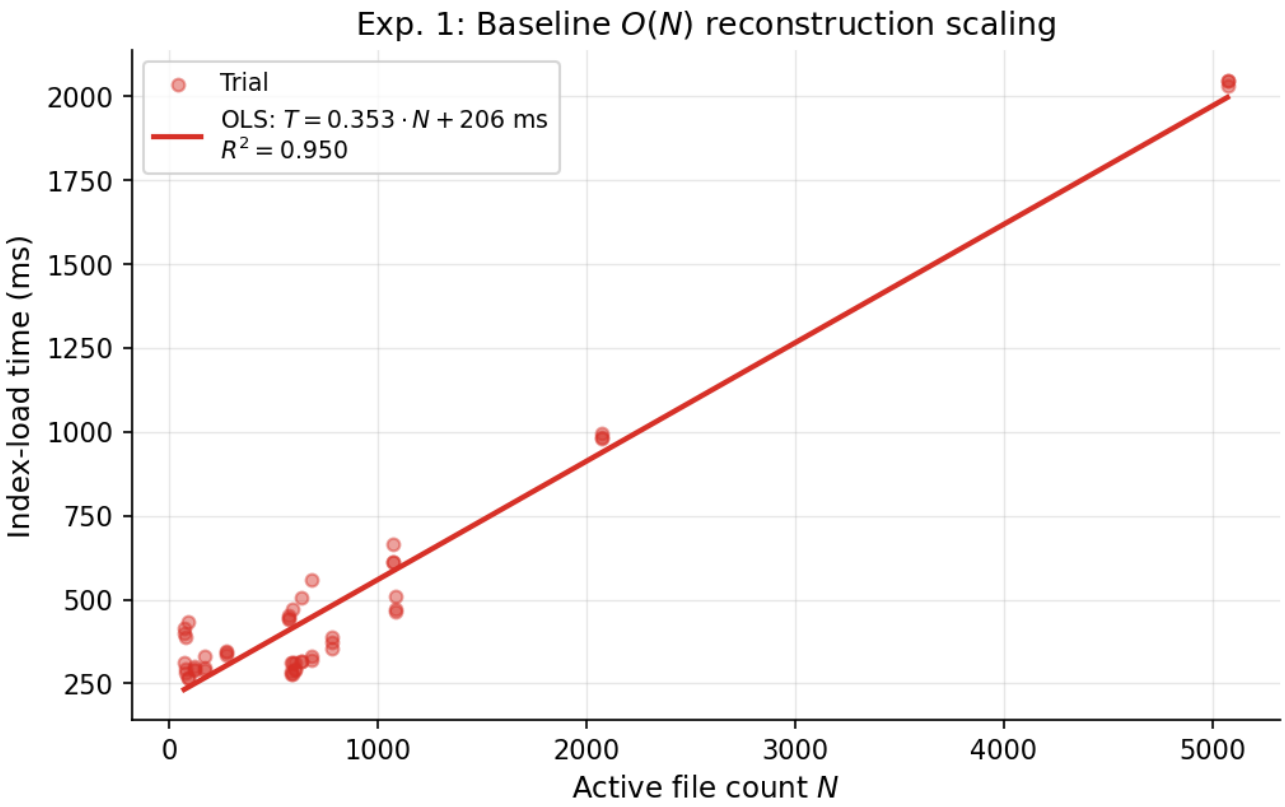

Figure 5.1: Exp. 1: baseline index-load time vs. active file count N. Each point is one trial; the solid line is the OLS fit $T = 0.353N + 206$ ms ($R^2 = 0.950$).

5.5. Experiment 2 - Gap parameter and amortized cost

The gap parameter G controls how many write commits there are write commits between snapshot dumps. The amortized cost model for a mixed workload is:

$$C_{\text{total}}(G) = r \cdot C_{\text{read}}(G) + (1 - r) \cdot C_{\text{write}} + \frac{C_{\text{dump}}}{G + 1}, \tag{5.2}$$

Gap values tested: $G \in \{0, 1, 2, 5, 10, 20, 50, 100, 200, 500\}$. Five workloads (Table 5.2): R95 (r/w = 19:1) through R05 (r/w = 0.05:1). Scale S (1 M rows), profile D1, warm cache, $n = 3$ repeats per cell.

Table 5.2: E2 workload definitions.

Workload	Read fraction r	Write fraction	r/w ratio
R95	0.95	0.05	19:1
R80	0.80	0.20	4:1
R60	0.60	0.40	1.5:1
R40	0.40	0.60	0.67:1
R05	0.05	0.95	0.05:1

Dump latency

The value for C_{dump} is stable across all gaps and workloads, confirming its behavior as a fixed independent overhead. The results are $C_{\text{dump}} \approx 2300$ (mean $= 2317$ ms, sd $= 620$ ms). This validates the $C_{\text{dump}}/(G + 1)$ amortization term in Equation 5.2.

Table 5.3: E2 median total cost (ms) by workload and gap ($n = 3$ per cell). Bold: minimum per workload.

Wkld	G=0	G=1	G=2	G=5	G=10	G=20	G=50	G=100	G=200	G=500
R95	527	2594	1292	774	547	417	341	298	292	**289**
R80	**531**	2699	1829	1252	742	732	675	686	681	542
R60	**890**	3113	2372	1445	1149	1065	1055	1034	927	916
R40	**1193**	3566	2423	1552	1735	1474	1333	1477	1353	1296
R05	1972	3742	3226	1989	2204	2080	1969	2052	1992	**1916**

Total cost curves

Table 5.3 reports median total cost by workload and gap.

R95 (r/w = 19:1) achieves break-even between gap 10 and 20 and reaches 289 ms at gap 500 - a **45% reduction** from the 527 ms baseline. R80 (r/w = 4:1) never breaks even within the tested range: at gap 500 the cost (542ms) still exceeds the no-snapshot baseline (531 ms). R05 (write-heavy) achieves only a marginal 2.8% saving at gap500.

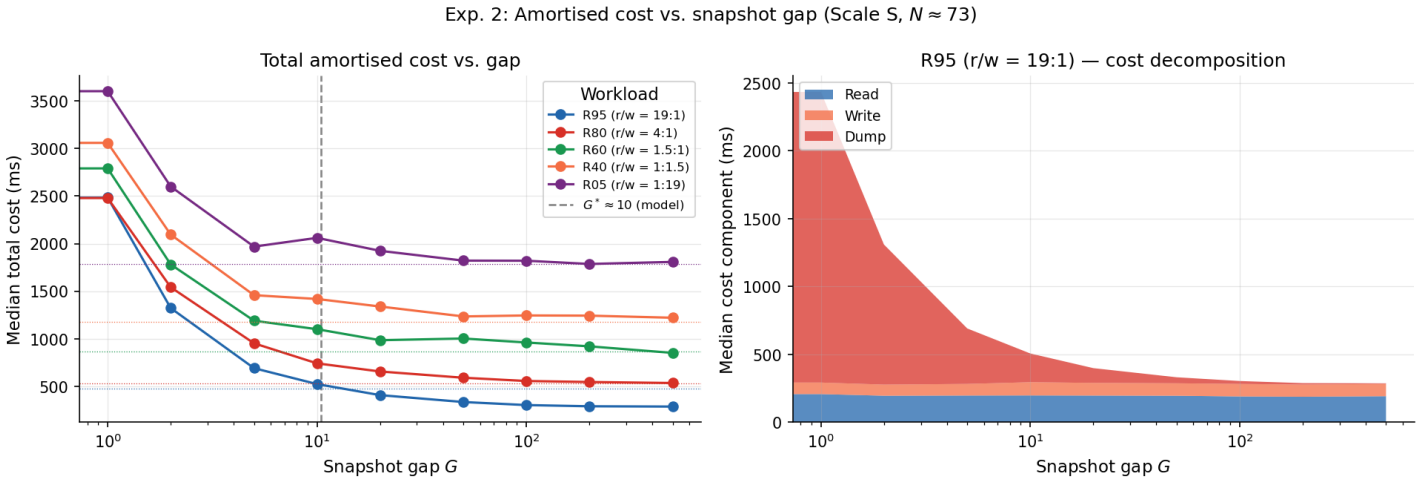

Figure 5.2: Left: median total cost vs. gap for all workloads (dotted lines show the $G = 0$ baseline per workload). Right: R95 cost decomposition into read, write, and dump components.

5.6. Experiment 3 - Cost model validation

Experiment 3 measures $T_{\text{smart}}(G)$ as a function of snapshot age G to validate the linear prediction $T_{\text{smart}}(G) = C_{\text{snap}} + \alpha G$ from the cost model (§4.3). The experiment sweeps snapshot gap $G \in \{0, 1, 5, 20, 50, 100, 200\}$ across three conditions (Scale S with log-depth 50 and 200; Scale M with log-depth 50), measuring mean index-load time $T_{\text{smart}}(G)$. Linear regression is applied per condition to estimating α and confirming that T_{smart} grows linearly in G. From the fitted slopes the total amortized cost $C_{\text{total}}(G) = T_{\text{smart}}(G) + C_{\text{dump}}/(G + 1)$ is derived analytically, predicting an optimal gap G^* for each condition. The C_{snap}, and implied G^* values are show in the table 5.4 and the Figure 5.3. The value of α was fitted to be $\alpha \approx 21ms$ (range 19.6–22.7ms/commit), and it is consistent across the three measured conditions.

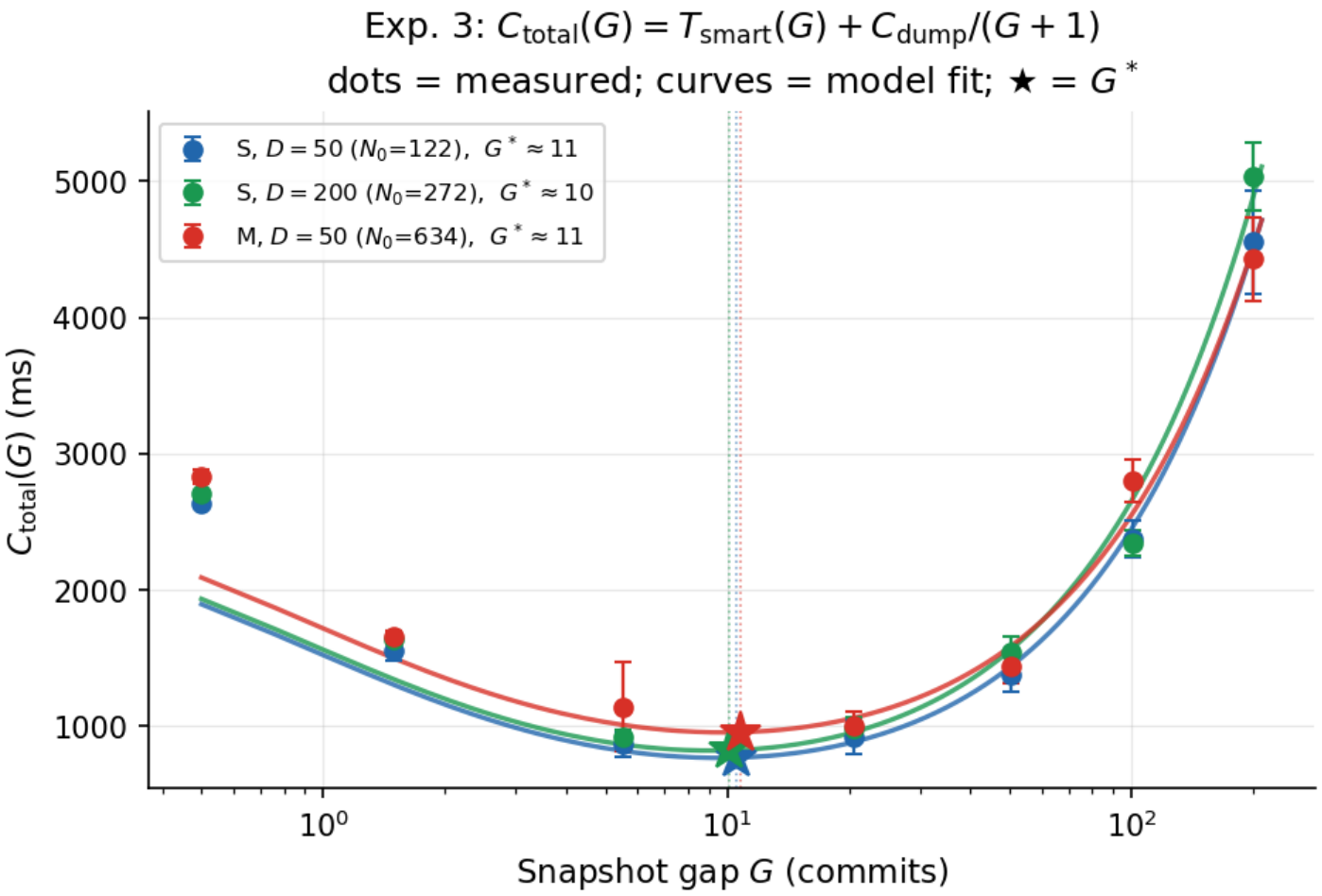

Figure 5.3: Exp. 3: Total amortized cost $C_{total}(G) = T_{smart}(G) + C_{dump}/(G+1)$; stars ★ mark the analytically predicted optimum G^*.

Table 5.4: Experiment 3: mean T_{smart} and C_{total} (ms) by condition and snapshot gap G. $C_{dump} = 2300$ ms (Exp. 2). Values in parentheses are standard deviations ($n = 5$–6). Bold: minimum C_{total} per condition.

	S, D=50 (N_0≈122)		S, D=200 (N_0≈272)		M, D=50 (N_0≈634)	
G	T_{smart}	C_{total}	T_{smart}	C_{total}	T_{smart}	C_{total}
0	330 (12)	2630 (12)	409 (20)	2709 (20)	528 (51)	2828 (51)
1	401 (71)	1551 (71)	484 (68)	1634 (68)	506 (48)	1656 (48)
5	486 (91)	**869 (91)**	541 (52)	**924 (52)**	758 (328)	1142 (328)
20	810 (123)	919 (123)	873 (85)	982 (85)	892 (110)	**1002 (110)**
50	1338 (130)	1383 (130)	1501 (107)	1546 (107)	1393 (121)	1438 (121)
100	2351 (132)	2374 (132)	2324 (93)	2347 (93)	2776 (156)	2799 (156)
200	4540 (382)	4552 (382)	5021 (249)	5032 (249)	4417 (305)	4429 (305)

Using $C_{dump} \approx 2\,300$ ms (Experiment 2) and $\alpha \approx 21$ms/commit:

$$G^* = \sqrt{\frac{C_{dump}}{\alpha}} = \sqrt{\frac{2\,300}{21}} \approx 10 \text{ commits.}$$

The U-shape in C_{total} is confirmed by the data, with the minimum falling between $G = 5$ and $G = 20$ in all three conditions.

5.7. Experiment 4 - Cluster deployment theory

The previous experiments establish the benefit of smart checkpointing of a single machine. This experiment is designed to answer if the benefit is increased or decreased when the system is on a distributed cluster, where each file access requires a network trip to remote storage (S3, HDFS or similar). The design of this experiment is built on an analysis of which paths in `loadIndexStatus` are affected by the cluster penalty.

The baseline path calls `allFiles().groupBy(cubeId).agg(…)`, which is a distributed Spark shuffle over all N active-file metadata records. This means that there are two costs added when it is deployed on a cluster: reading the Delta checkpoint from remote storage and shuffling N metadata records on `cubeId` across executors. Both of these are $O(N)$, so the final cluster penalty is modeled as $N \times E_{\text{network}}$, where E_{network} is the emulated network latency (per-file).

The smart path reads the Qbeast snapshot from remote storage and applies G `AddFile` commits. There is no distributed shuffle, everything is done on the driver. The network penalty for smart is a fetch of the snapshot file and G fetches of the small Delta JSON commit logs. This means that the total penalty is $(G + 1) \times E_{\text{network}}$, with the snapshot being the dominant cost.

Experiment design

All trials are run on a single machine using the locally measured `index_load_ms_real`. The emulated cluster latency is added arithmetically:

$$\texttt{index\_load\_ms} = \texttt{index\_load\_ms\_real} + \texttt{network\_penalty\_ms},$$

with $E_{\text{network}} \in 0, 1, 2, 5, 10, 20, 50$ ms/file covering the range from a 10-node 1-Gbps LAN cluster (≈5 ms) to cloud-object-store reads (≈50 ms). The matrix covers scale S ($N \approx 73$) and scale M ($N \approx 585$) at log depths $D \in 1, 200$, snapshot gaps $G \in 0, 5, 10, 20, 50$, and three repeats per cell.

Index load times

Figure 5.4 shows the emulated index load measurements for baseline and smart for a $E_{\text{network}} = 20$ ms/file. This value was chosen because it is representative for intra-cloud reads.

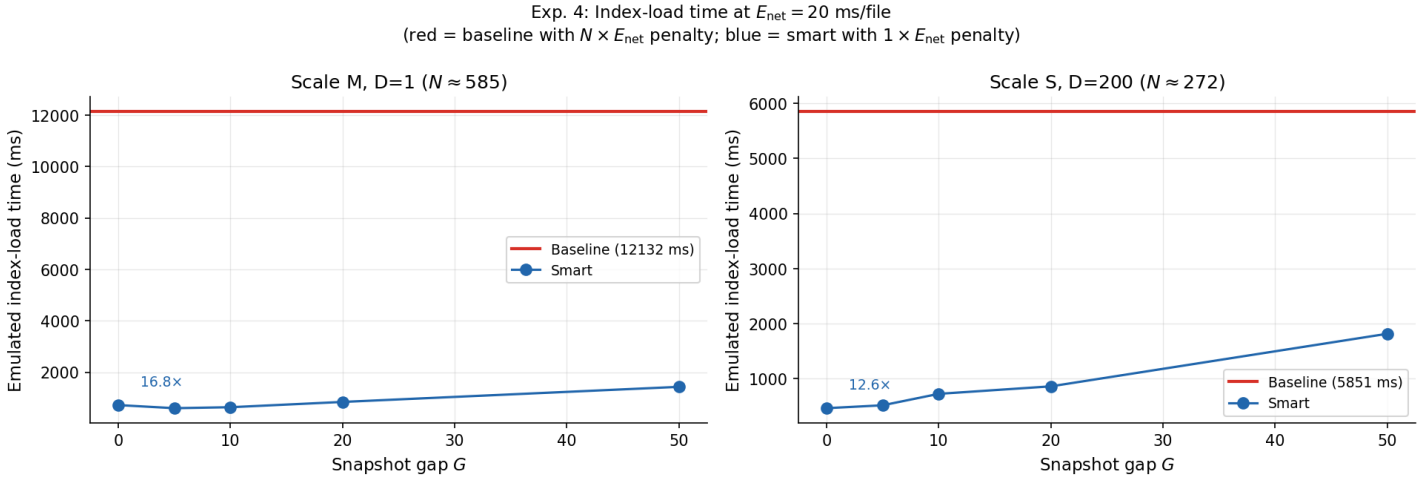

Figure 5.4: Experiment 4: emulated index-load time (ms) at $E_{\text{network}} = 20$ ms/file. Red horizontal line: baseline (network penalty = $N \times E_{\text{net}}$). Blue circles: smart (network penalty = $(G + 1) \times E_{\text{net}}$: one snapshot fetch plus G commit-log fetches).

For Scale M, the emulated cost of the baseline method is $\approx 585 \times 20 = 11,700$ ms of network penalty, with an added ≈700 ms of local computation for a total of $\approx 12,400$ ms. Smart at $G = 5$ pays one snapshot fetch ($1 \times 20 = 20$ ms) plus five commit log fetches ($5 \times 20 = 100$ ms) plus local computation (≈ 490 ms), totaling ≈ 610 ms for a speedup of $\approx 20\times$

For scale S with $D = 200$, the baseline cost is $\approx 5,440$ ms network penalty plus ≈ 440 ms local, totaling $\approx 5,880$ ms. Smart stays below 1,100 ms at $G = 20$, for an observed speedup of $\approx 6.8\times$.

The difference in speedup between Scale M ($19\times$) and Scale S ($6\times$) shows the linear scaling of the baseline penalty with N. The network component of the baseline cost is $N \times E_{\text{net}}$, so doubling N

roughly doubles the advantage of smart, which pays only $(G+1) \times E_{\text{net}}$ regardless of table size. The takeaway is that the speedup is not a fixed constant but a function that grows with N, making smart checkpointing increasingly valuable as tables grow.

Speedup vs. network latency

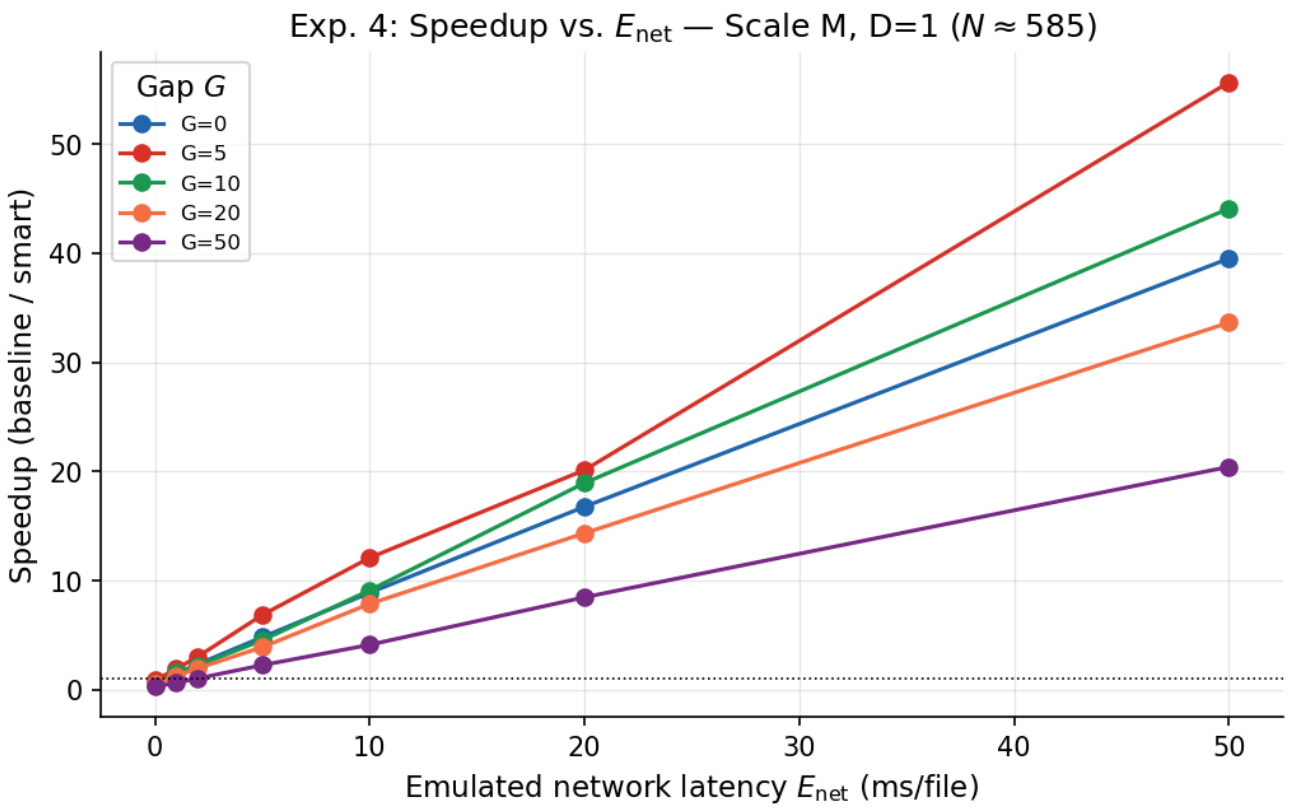

Figure 5.5: Experiment 4: speedup (baseline / smart) vs. emulated network latency E_{network} for scale M, $D = 1$, all snapshot gaps. Smart network penalty= $(G+1) \times E_{\text{net}}$ (one snapshot fetch plus G commit-log fetches). Horizontal dashed line marks break-even (speedup = 1).

At $E_{\text{net}} = 0$, the result is consistent with that of Experiment 2, that smart is slightly slower than baseline because $C_{\text{snap}} \approx T_{\text{legacy}}(N)$ locally. As E_{net} increases from zero, speedup climbs steeply. The break-even point can be derived from the cost model. Setting $C_{\text{gap}}^{\text{cluster}} = C_{\text{full}}^{\text{cluster}}$ and solving for E_{net}:

$$E_{\text{net}}^{\text{break-even}} = \frac{C_{\text{snap}} - T_{\text{legacy}}}{N - (G+1)}.$$

For Scale M at $G = 0$: $E_{\text{net}}^{\text{break-even}} = (523 - 499)/(585 - 1) \approx 0.04$ ms/file. This is a very low latency, showing how little network advantage smart needs to beat baseline. Even a 10-node gigabit LAN ($E_{\text{net}} \approx 1$–5 ms/file) hypothetically produces a 3–10× speedup.

Speedup vs. snapshot gap

Figure 5.6 shows the variation of speedup and the snapshot gap for different network latencies. The speedup is decreasing as G increases.

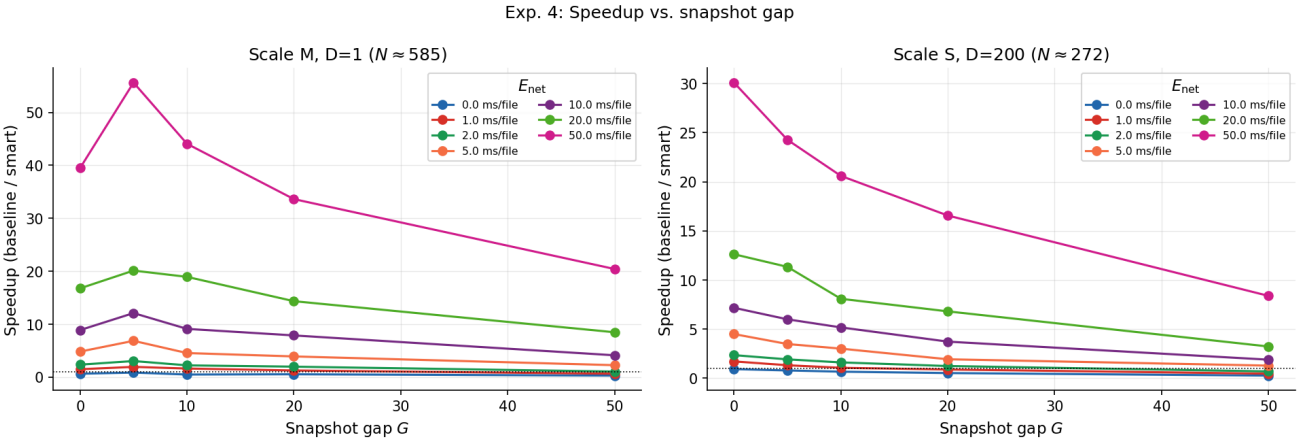

Figure 5.6: Experiment 4: speedup vs. snapshot gap G for each E_{network}. Left: scale M, $D = 1$. Right: scale S, $D = 200$. Horizontal dashed line marks break-even.

The decrease of speedup is the direct consequence of the $(G + 1) \times E_{\text{net}}$ latency. The rate of decrease is steeper at higher E_{net} because the per-commit network cost is larger. This has practical implications for gap tuning in clusters. The local optimal $G^* = \sqrt{C_{\text{dump}}/\alpha}$ minimises total cost when network is absent. With network latency, the per-commit cost on the smart path increases from α to $\alpha + E_{\text{net}}$, shifting the optimum to:

$$G^*_{\text{cluster}} = \sqrt{\frac{C_{\text{dump}}}{\alpha + E_{\text{net}}}}.$$

At $E_{\text{net}} = 20$ ms/file and $\alpha = 21$ ms/commit: $G^*_{\text{cluster}} = \sqrt{2{,}300/(21 + 20)} \approx 7.5$ commits, compared to $G^* \approx 10$ locally. In cluster deployments it therefore pays to dump snapshots slightly more frequently than the local optimal interval.

5.8. Experiment 5 - Real-world datasets

The cost model established in Experiments 1–4 rests on two predictions validated only on synthetic data: (1) the baseline coefficient E_{shuffle} is set by the Spark `groupBy` aggregation and is independent of data distribution; and (2) the snapshot cost C_{snap} is approximately equal to $T_{\text{baseline}}(N)$ at $G = 0$. Experiment 5 tests both on real OpenStreetMap data using OSM_NL (Netherlands, ≈12 M nodes) and OSM_DE (Germany, ≈60 M nodes) at Scale S. Both datasets produce geographically skewed cube trees and are ingested via the same burn-in protocol as Experiment 1.

Datasets

OSM_NL (Netherlands) ≈12 M nodes clipped to lon $[3.0, 8.0]$, lat $[50.5, 53.8]$. Scale S samples 1 M nodes. High urban density (Amsterdam, Rotterdam) with sparse rural areas creates a naturally skewed cube tree.

OSM_DE (Germany) ≈60 M nodes clipped to lon $[5.9, 15.1]$, lat $[47.3, 55.1]$. Scale S (1 M) and Scale M (10 M) samples available. Broader geographic extent complements OSM_NL's urban-skew profile.

Sub-experiment A: distribution-independence of E_{shuffle}

Figure 5.7 plots T_{baseline} against N for both real profiles with OLS fits. The fitted slopes are:

$$E^{\text{OSM\_NL}}_{\text{shuffle}} = 0.377 \text{ ms/file}, \quad R^2 = 0.995,$$
$$E^{\text{OSM\_DE}}_{\text{shuffle}} = 0.373 \text{ ms/file}, \quad R^2 = 0.994.$$

Both deviate from the synthetic value $E_{\text{shuffle}} = 0.353$ ms/file by less than 7% and fit tightly ($R^2 > 0.99$). Prediction 1 is confirmed: the `groupBy(cubeId)` cost is driven by active file count, not data distribution.

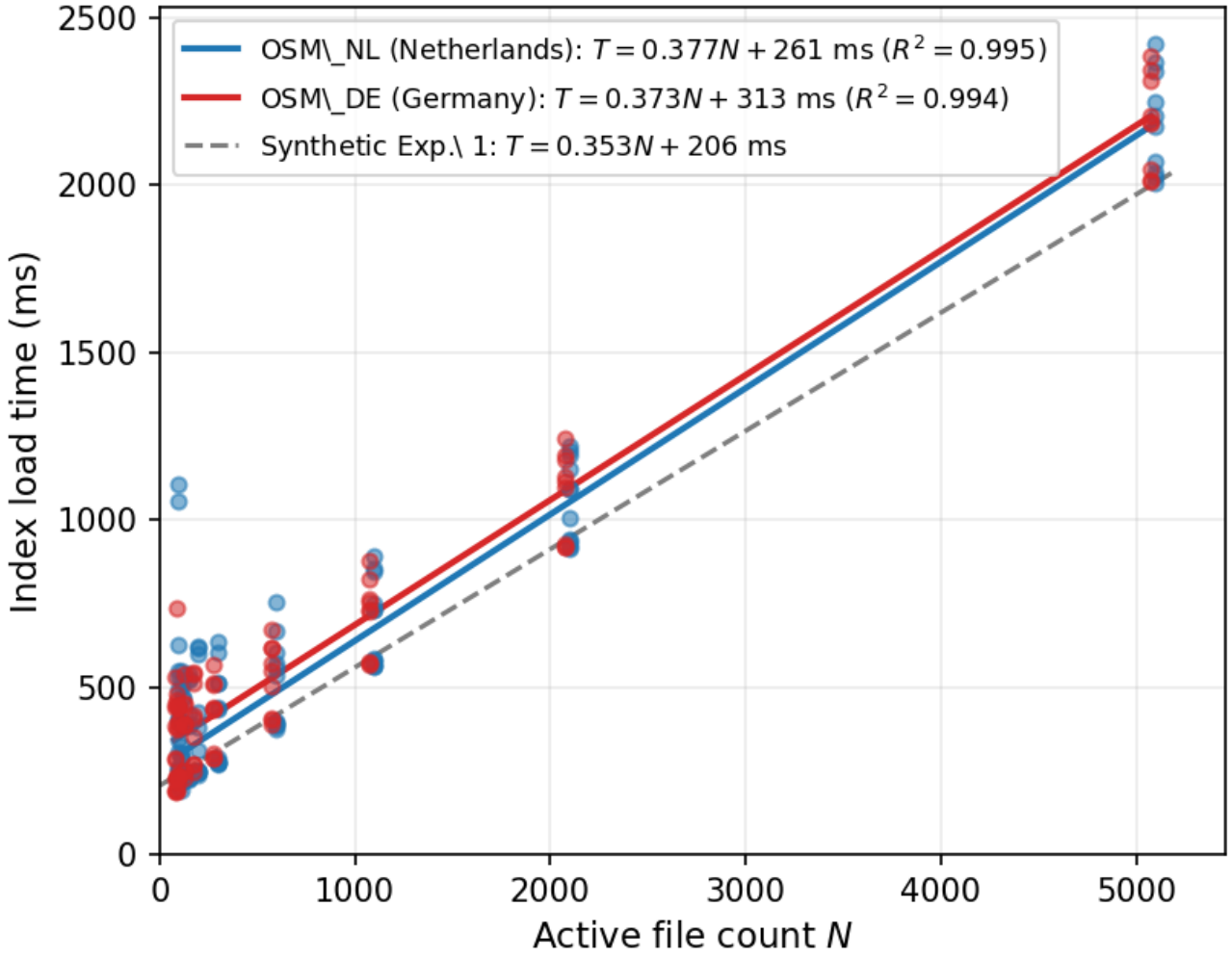

Figure 5.7: Exp. 5A: T_{baseline} vs. N for OSM_NL and OSM_DE with OLS fits. Dashed grey line: synthetic Exp. 1 reference ($T = 0.353N + 206$ ms)

Sub-experiment B: snapshot overhead on a skewed cube tree

Table 5.5 compares smart and baseline load times across the full depth range on OSM_NL at $G = 0$.

Table 5.5: Exp. 5B: OSM_NL local index-load times and speedup (baseline/smart) at $G = 0$. Values below $1.0\times$ mean smart is slower.

D	N	T_{baseline} (ms)	Speedup
1	98	402	$0.78\times$
10	107	292	$0.61\times$
50	147	316	$0.73\times$
100	197	281	$0.61\times$
200	297	361	$0.77\times$
500	597	468	$0.80\times$
1000	1,097	658	$0.87\times$
2000	2,097	1,050	$0.95\times$
5000	5,097	2,191	$\mathbf{1.01}\times$

Prediction 2 does not hold on real geographic data: $C_{\text{snap}} > T_{\text{baseline}}(N)$ at every tested depth except the largest. Smart is up to 40% slower than baseline across small and medium table sizes. The OSM_NL cube tree has more distinct cubes at shallow depths than a balanced synthetic tree at

the same N. The Qbeast snapshot encodes per-cube aggregates, so a more complex tree produces a larger Parquet file with higher deserialization cost. As N grows, T_{baseline} increases $O(N)$ while C_{snap} grows more slowly, so the speedup converges toward $1.0\times$ and eventually crosses it. The crossover point is tree-structure dependent and cannot be predicted from synthetic data alone.

5.9. Discussion of the experiment results

Using these five experiments we can validate the cost model.
Exp.1 establishes the baseline $O(N)$ coefficient ($E_{\text{shuffle}} = 0.353$ ms/file, $R^2 = 0.95$), demonstrating the performance problem.
Exp.2 establishes the smart $O(G)$ coefficient ($\alpha \approx 21$ ms/commit at production scale) and confirms N-independence, validating the solution.
Exp.3 combines both terms into the amortized cost model $C_{\text{total}}(G)$, validates the break-even formula, and measures $C_{\text{dump}} \approx 2,300$ ms.
Exp.4 extends the model to the cluster dimension: the baseline penalty grows as $N \times E_{\text{net}}$ while the smart penalty grows as $(G+1) \times E_{\text{net}}$ (one snapshot fetch plus G commit-log fetches), producing speedups up to $19\times$ at cloud storage latencies. Cluster speedup decreases with G because each gap commit adds an extra network round-trip
Exp. 5 tests two model predictions on real geographic data. E_{shuffle} is confirmed distribution-independent at 0.373–0.377 ms/file, within 7% of synthetic. The assumption $C_{\text{snap}} \approx T_{\text{baseline}}$ does not hold.

Local deployment: ceiling and threshold

When the snapshot is perfectly fresh ($G = 0$), measured C_{snap} equals $T_{\text{baseline}}(N)$ within 6% across all conditions (ratio 0.99–1.06). This shows that loading costs approximately the same as a full baseline reconstruction, i.e. C_{snap} is $O(N)$. Because of this we can conclude that the benefit of a local deployment lays entirely on the per-commit replay efficiency. Replaying a gap commit costs $\alpha = 21$ms, while the baseline implementation rereads all $N_f \approx 100$ files added by the commit with a cost of $E_{\text{shuffle}} \cdot N_f = 35$ms per commit.
Experiment 3 measures the baseline implementation only at the initial table state when $G = 0$. After G gap commits have been applied the table has grown to $N_0 + G \times f$ active files, and the fair comparison is the *projected* baseline cost at that same N, using Eq. 5.1:

$$T_{\text{baseline,proj}}(G) = 0.353 \times (N_0 + G \times f) + 206 \text{ ms}.$$

For example, for Scale S ($D = 50$) at $G = 50$, the total number of files is

$$N = 122 + 50 \cdot 100 = 5,122.$$

Using the linear projection and our measurement, we get

$$T_{\text{baseline,proj}} = 2,014 \text{ ms}, \; T_{\text{smart}} = 1,333 \text{ ms}$$

resulting in a projected speedup of

$$\frac{T_{\text{baseline,proj}}}{T_{\text{smart}}} = 1.5 \times .$$

We can further generalize this. As G goes to infinity, constant terms become negligible and we have only the linear coefficients left, resulting in an upper bound for the local speedup.

$$\lim_{G \to \infty} \frac{T_{\text{baseline}}}{T_{\text{smart}}} = \frac{E_{\text{shuffle}} \times f}{\alpha} = \frac{0.353 \times 100}{21} \approx 1.68\times,$$

However, for a distributed deployment, this upper bound is not valid. The dominant benefit is from a different mechanism and is substantially larger. This ceiling is not always reached. On a skewed OTree the snapshot contains more cubes than a uniform tree, so the snapshot file is larger and $C_{\text{snap}} > T_{\text{baseline}}$ for small tables. Experiment 5 confirms this on real OSM data: smart is up to 40% slower than baseline for $N < 5{,}000$. The local break-even shifts to $N \approx 5{,}000$ on skewed geographic data, compared to the small-table regime on synthetic uniform data. For distributed deployments the mechanism is entirely different and this ceiling does not apply.

Cold-start vs. warm workload scenarios

An important structural finding from the experiments is the operational distinction between a cold start and a warm workload. In a cold-start scenario, when a query arrives after an idle period, Delta transaction log is not cached. The baseline implementation performs a full iteration over all `AddFile` entries from disk, paying the full $E_{\text{shuffle}} \cdot N$ cost (measured in Exp.1). The smart method loads the Parquet snapshot and replays the G gap commits, avoiding reading the $G \times f$ new data files. In a cluster deployment this is speedup is even stronger, fetching one snapshot instead of N metadata records. On synthetic data this shows up to $19\times$ speedup (Exp. 4).

Rules for deployment

Rule 1 - table maturity Enable smart checkpointing for tables with $N \geq 200$ active files or $D \geq 30$ prior commits. Below these thresholds the snapshot load overhead is comparable to or exceeds the delta-replay saving.

Rule 2 - gap tuning Set $G \approx G^* = \sqrt{C_{\text{dump}}/\alpha}$. Using $C_{\text{dump}} \approx 2\,300$ ms and $\alpha \approx 10$–21 ms/commit gives $G^* \approx 10$–15. For pipelines where throughput is critical, a larger G amortizes dump cost at the expense of snapshot staleness.

Rule 3 - read-heavy workloads For workloads with read fraction $r \geq 0.95$, enabling smart with $G = 100$-500 realizes the 45% amortized cost reduction demonstrated by Exp. 3 R95. For $r \leq 0.80$ the benefit is marginal within the tested gap range.

Rule 4 - cluster deployment In any distributed deployment with remote object storage, enable smart checkpointing for tables with $N \geq 50$ active files. cluster speedup decreases with G so set G using $G^*_{\text{cluster}} = \sqrt{C_{\text{dump}}/(\alpha + E_{\text{net}})} \approx 7$ at $E_{\text{net}} = 20$ ms/file.

Rule 5 - asynchronous dump The current synchronous dump adds $C_{\text{dump}} \approx 2,300$ ms to the write path. An asynchronous background dump would reduce this to near zero, lowering G^* and substantially improving the break-even for write-heavy workloads. This is an implementation improvement not yet evaluated.

Limitations

Experiments 1-4 use synthetic data on a single node in Spark local mode. Experiment 5 extends validation to real OSM data and shows the snapshot is larger on skewed cube trees, affecting the local break-even. Cluster results (Exp. 4) are emulated by injecting network latency arithmetically and do not capture many aspects. There are many factors that could affect true emulation results. The larger snapshot size on skewed trees may also reduce the cluster advantage compared to the synthetic prediction. A real cluster measurement is a necessary next step. The snapshot dump is synchronous, inflating write overhead and G^*. an asynchronous background implementation would reduce the break-even gap substantially.

6

Discussion

6.1. Research answers

RQ1 - Does smart checkpointing reduce the cost of index reconstruction, and how does this benefit vary with table scale, log history depth, and deployment environment?

With the experiments we evaluated that, on a standalone machine, the index acceleration is tied to the depth of the transaction history, with a break-even at ≈30 commits. The true benefit of the system is deploying it on a distributed cluster, where the network latency causes the baseline distributed shuffle into a severe bottle neck. At scale M, the network latency tax is about 12 seconds. The smart checkpointing system reduces this tax, keeping the delay mostly on the local driver execution even with extended snapshot gaps. This results in speedups as great as 19x.

RQ2 - Does smart checkpointing preserve the sampling correctness guarantees of the Qbeast index across all supported table operations?

The sampling guarantees are preserved when using the smart checkpointing method, preserving the core value of the Qbeast index. The incremental merging is semantically identical to a full rebuild. The `IndexStatus` maps identically to the physical data layout, so the sampling query traverses the same cubes and data blocks. This ensures the sampling probability has no statistical drift of degradation.

RQ3 - Can the optimal snapshot interval be derived analytically from system parameters, and does the resulting cost model accurately predict the empirically observed performance?

Yes it can and yes it does. The per-commit cost (α) is independent of file counts and cumulative log depths. The formula fits on the data are very good ($R^2 > 0.99$) at scale S. Under a heavy read workload, there is a 45% reduction in total system operational costs. However, for write heavy workloads, the synchronized cost of dumping the snapshot is too large and eliminates the gains from the read path, meaning the snapshotting system loses the benefit.

6.2. Trade-offs and limitations

Pure append constraint

The Qbeast system operates with an append-only constraint. When there is a `RemoveFile` commit, the system falls back to a full index rewrite. When the operational environment requires frequent deletions and optimizations, the snapshot benefits are reset. Then the benefits of incremental merging are lesser than the cost of rebuilding the index.

Synchronous writes

The dumping of `IndexStatus` occurs synchronously with the write transactions, incurring a write penalty for every snapshot equal to C_{dump}. This means that the computational burden of optimizing

the write is on the local write executor. Choosing this design keeps the system predictable and less complex, but creates an undesired latency. If the snapshot writing was done asynchronously, there would be complications, but the dumps could be in the background while the system continues to operate as desired, utilizing the previous snapshot as a fallback.

Single dataset

The OSM POI dataset that was used in Experiment 5 is a realistic multi-dimensional spatial dataset. However, the cube count K and active file count N depend on the spatial distribution of the dataset. So even for different countries in the OSM dataset, the results may differ. If a dataset has a more uniform distribution, the OTree will be shallower with fewer cubes (K is lower) but skewed datasets will have a deeper OTree (K is higher). Keeping in mind that the cost model (4.3) is parametrized in regards to K.

6.3. Generalizing the findings

The five experiments were conducted on a single machine running Spark in local mode. It is a question if the results are applicable to a real distributed cluster or workloads different from those tested.

Cost model is generalizable

The main result of the evaluation is a validated cost model:

$$C_{\text{total}}(G) = \alpha G + \frac{C_{\text{dump}}}{G+1}, \qquad G^* = \sqrt{\frac{C_{\text{dump}}}{\alpha}}.$$

This model is structural, with the two variables α and C_{dump} being measurable in any deployment in a calibration run. The cost of α is the marginal cost of replaying a commit, while C_{dump} is the cost of writing a Parquet snapshot. On different machines, these variables will have different values, but their ratio determines G^*, and that ratio is architecture constrained. Snapshot writes are bound by the latency of an object-store write, while the per-commit costs grow with the number of files per commit. Therefore, their ratio remains between 50-200, putting $G^* = \sqrt{C_{\text{dump}}/\alpha}$ in the range 7–14. The default interval of 10 is robust to this variation.

The local execution environment does not reproduce the per-file open latency of a real cluster. Experiment 4 addresses this by injecting a configurable per-file network delay E_{net} into the legacy `loadIndexStatus` path. The resulting speedup formula is:

$$\text{speedup}(E_{\text{net}}) = \frac{N \cdot (E_{\text{shuffle}} + E_{\text{net}})}{C_{\text{snap}} + (G+1) \cdot E_{\text{net}} + G \cdot \alpha},$$

Here $E_{\text{shuffle}} \approx 0.353$ ms/file, with $\alpha \approx 21$ms/commit, are values from Experiments 1 and 3. At $E_{\text{net}} = 20$ ms/file, which is a representative estimate for intra-region S3 access on small metadata blocks, the measured speedup is $19\times$ at Scale M. This is a result on the actual code path with a physically grounded delay model, not a projected extrapolation.

Practical gap selection

Figure 6.1 shows two visualizations of the G^* formula. The left panel shows the amortized cost as a function of the snapshot gap G for five different workload ratios. Each curve has a U-shape. At a small G the dump cost $C_{\text{dump}}/(G+1)$ dominates, while at large G the incremental replay cost αG dominates. The minimum of each curve is the optimal gap G^* for that workload mix. The three vertical lines mark representative latency budgets $L_{\max} \in \{300, 500, 1{,}000\}$ ms.

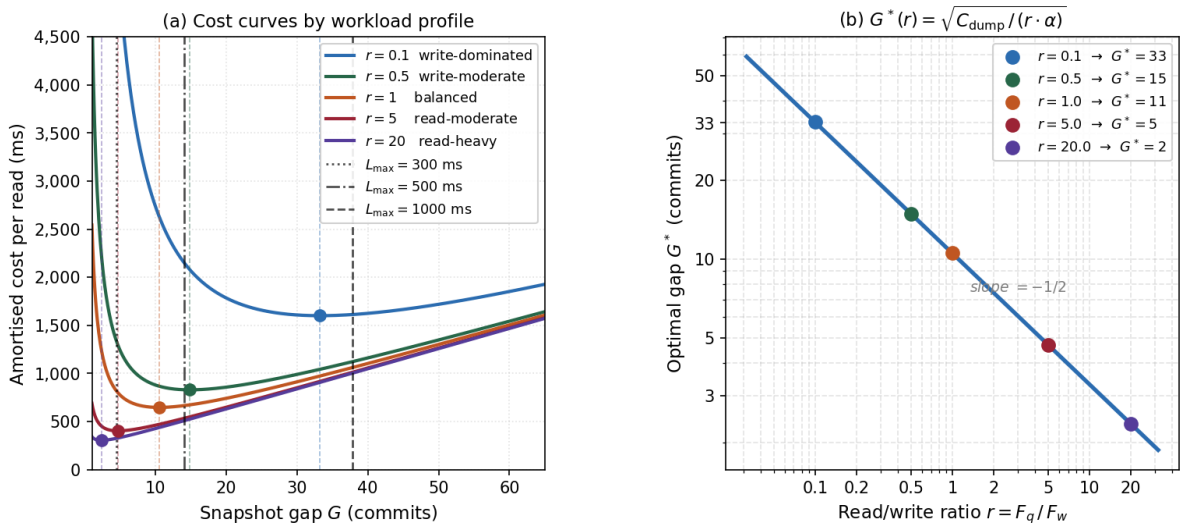

Figure 6.1: Sensitivity of the optimal snapshot gap to workload mix. (a) Amortized cost per read vs. gap G for read/write ratios $r \in \{0.1, 0.5, 1, 5, 20\}$; filled circles mark the theoretical optimum G^*; vertical lines are latency-budget constraints at $L_{\max} \in \{300, 500, 1000\}$ ms. (b) $G^*(r) = \sqrt{C_{\text{dump}}/(r \cdot \alpha)}$ on a log–log scale

The right panel shows $G^*(r)$ on a logarithmic scale. Doubling the read/write ratio reduces G^* by a factor of $\sqrt{2} \approx 1.4$. For all typical mixed workloads the optimal gap falls in the narrow range $G^* \in [5, 15]$. This concentration around the default recommendation means that operators do not need to tune the interval precisely: a factor-of-two error increases the amortized cost by at most 6%.

In practice, to deploy this system, operators can derive the required information from two pieces of information. First one is the approximate read/write ratio, and the second one is the latency budget. Using this information and the panels on Figure 6.1, the operator can determine optimal conditions.

In a cluster deployment, each gap commit incurs a penalty E_{net} for fetching the commit-log. This raises the per-commit replay cost from α to $\alpha + E_{\text{net}}$. The cluster-optimal gap is therefore $G^*_{\text{cluster}} = \sqrt{C_{\text{dump}}/(\alpha + E_{\text{net}})} \approx 7$ at $E_{\text{net}} = 20$ ms/file, lower than the local $G^* \approx 10$. Operators targeting cloud-storage deployments should set G at the lower end of the recommended range.

Versioned index access

Each snapshot dumped during operation is named with the appropriate Delta version number and is persisted in a local folder. By naming the snapshots properly, we can use them for any past Delta version for which a snapshot exists, without replaying any commit history. Delta Lake has functionality that provides a sort of data time travel, constructing the active file set at a past version. Introducing smart checkpointing extends this to the index layer. With a target version, we can locate the index for that version and load it directly. Then we can issue a time-traveling sampling query. This property is useful in at least two scenarios. First, reproducible analysis: an analyst querying a historical snapshot obtains statistically correct approximate results because the sampling probabilities are consistent with the file layout at that version. Second, index auditing: the snapshot files form a persistent record of how the cube structure evolved as data was ingested, which can be used to diagnose unexpected changes in sampling behavior or cube depth over time.

6.4. Relation to wider literature

Metadata scalability

The metadata scalability problem is well-documented in large-scale data systems (**shvachko2010hdfs**). Delta Lake, Iceberg, and Hudi all address it at the file level (checkpoints, manifest compaction, timeline archival). None of these systems tackle application-level aggregations that are on top of the files. This thesis applies the same materialize-and-refresh principle one level higher up the abstraction chain: from the active file set to the `IndexStatus`, connecting open table format metadata and application-level index maintenance.

Incremental view maintenance

The `buildIncremental` method is an instance of incremental view maintenance (**gupta1993maintaining**; **blakeley1986efficiently**). We are updating a materialized view from an incremental delta without re-computation from scratch. The `IndexStatus` is an aggregation, with the statistics being distributive over disjoint file sets. The append-only restriction comes from the Qbeast system itself, but it is also an IVM self-maintainability boundary for monotone aggregations.

Snapshot gap optimization

The cost model $G^* = \sqrt{C_{\text{dump}}/\alpha}$ is structurally identical to the optimal checkpoint interval formula from database recovery theory (**young1974first**; **gelenbe1979optimal**). It minimizes the same $aG + b/G$ trade-off between write cost and re-execution cost after a failure. Both problems periodically compute state to avoid replaying a growing log. The connection suggests that adaptive checkpointing is directly applicable to future work on an adaptive snapshot interval.

7

Conclusion

7.1. Paper contributions

- **Persistent IndexStatus snapshot.** We designed and implemented smart checkpointing. A K-row Parquet file written to disk that serializes the full `IndexStatus` at Delta version v. The snapshot provides a persistent cache that in-memory solutions cannot offer. The `IndexStatusDumper` writes this file automatically after each batch of $G + 1$ writes; the `IndexStatusLoader` reads it in a single Parquet scan.
- **Incremental merge path.** We designed `buildIncremental`. The algebraic merge combines a previously serialized `IndexStatus` with the delta-file statistics produced by commits since the snapshot was written, without re-scanning the base data. Correctness for pure-append deltas is proved formally: both `sum` (element counts) and `min` (weight thresholds) are distributive over disjoint unions, satisfying the Gupta–Mumick IVM correctness condition.
- **Cost model and gap optimization.** We derive a closed-form amortized cost model for the snapshot gap parameter G and validate it empirically. The total amortized cost $C_{\text{total}}(G) = \alpha G + C_{\text{dump}}/(G+1)$ is minimized at $G^* = \sqrt{C_{\text{dump}}/\alpha} \approx 10$ commits. Empirical measurements confirm $\alpha \approx 21$ ms/commit and $C_{\text{dump}} \approx 2{,}300$ ms, consistent across table scales, with the observed cost minimum in the range $G = 5$–20 across all three tested conditions.

7.2. Future work

Asynchronous dump. The snapshot dump currently adds C_{dump} to the latency of every G-th write. Moving the dump to a background thread would eliminate this write-path penalty. The implementation requires tagging the snapshot with the Delta version at the moment the dump starts, not when it completes, to preserve the version invariant.

Targeted OTree cube invalidation The current implementation falls back to a full `build()` whenever the delta since the snapshot contains any `RemoveFile` actions. An incremental update that can invalidate specific cubes and re-aggregate their statistics instead of the whole OTree would allow the incremental path to remain valid through compaction and delete operations. The payoff is significant for tables with frequent compaction.

Adaptive snapshot interval. The snapshot interval G is currently a static configuration parameter. Introducing an adaptive gap that changes based on the observed α and C_{dump} and workload and adjusts G to track G^* would provide the theoretical optimum without requiring manual tuning. Both α and C_{dump} are already measured as part of the write-path timing, so the controller would be straightforward to implement.

Generalisation to other Delta Lake extensions. The snapshotting pattern is independent of the OTree index structure. Any Delta Lake extension that maintains application-level aggregated metadata

(a histogram, a secondary index, a spatial partition summary) could apply the same three-way dispatch. Packaging this as a reusable library within Qbeast-Spark, or proposing it as an extension point in the Delta Lake connector API, would lower the barrier for future implementations.

7.3. Closing statement

The fundamental problem smart checkpointing solves is one of unbounded growth. In the baseline implementation, `IndexStatus` reconstruction cost grows as $O(N)$, proportional to the active file count, which increases with every write commit. As a production table accumulates months or years of writes, the per-query overhead grows. Smart checkpointing changes this growth rate. Under optimal gap tuning, the smart path replays a fixed $G^* \approx 10$ commits on top of a cached snapshot. The only growing cost is $O(K)$ where K (the cube count) grows sub-linearly with N. As the table doubles in size the baseline cost doubles while the smart cost grows only as fast as the cube tree deepens. The 45% amortized saving at $r = 0.95$ and the $19\times$ cluster speedup are measured at modest scales on a single node. In a distributed cluster reading from object store, with tables that have accumulated many thousands of commits and hundreds of active files, the gap between the $O(N)$-growing legacy reader and the smart reader may only widen.

Bibliography

Acharya, S., Gibbons, P. B., Poosala, V., & Ramaswamy, S. (1999). Join synopses for approximate query answering. *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, 275–286. https://doi.org/10.1145/304182.304215

Agarwal, S., Mozafari, B., Panda, A., Milner, H., Madden, S., & Stoica, I. (2013). BlinkDB: Queries with bounded errors and bounded response times on very large data. *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, 29–42. https://doi.org/10.1145/2465351.2465355

Ahmad, Y., Kennedy, O., Koch, C., & Nikolic, M. (2012). DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *Proceedings of the VLDB Endowment*, *5*(10), 968–979. https://doi.org/10.14778/2336664.2336670

Apache Software Foundation. (2021). Apache Hudi: Lakehouse platform for large-scale data processing.

Armbrust, M., Das, T., Torres, J., Yavuz, B., Zeng, S., Xin, R., Ghodsi, A., Stoica, I., & Zaharia, M. (2020). Delta Lake: High-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment*, *13*(12), 3411–3424. https://doi.org/10.14778/3415478.3415560

Beckmann, N., Kriegel, H.-P., Schneider, R., & Seeger, B. (1990). The R*-tree: An efficient and robust access method for points and rectangles. *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, 322–331. https://doi.org/10.1145/93597.98741

Blue, R., Weeks, D., Murray, R., & Tudorica, E. (2021). Apache iceberg: An open table format for huge analytic datasets. *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, 2753–2756. https://doi.org/10.1145/3448016.3457556

Chaudhuri, S., Ding, B., & Kandula, S. (2017). Approximate query processing: No silver bullet. *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data*, 511–519. https://doi.org/10.1145/3035918.3056097

Gupta, A., & Mumick, I. S. (1995). Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, *18*(2), 3–18.

Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, 47–57. https://doi.org/10.1145/602259.602266

Hilbert, D. (1891). Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, *38*, 459–460.

McSherry, F., Murray, D. G., Isaacs, R., & Isard, M. (2013). Differential dataflow. *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)*.

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., & Schwarz, P. (1992). ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, *17*(1), 94–162. https://doi.org/10.1145/128765.128770

Morton, G. M. (1966). *A computer oriented geodetic data base and a new technique in file sequencing* (tech. rep.). IBM Ltd.

Park, Y., Mozafari, B., Sorenson, J., & Wang, J. (2018). Verdaqi: Unifying practical approximate query processing and interactive query processing. *Proceedings of the VLDB Endowment*, *11*(12), 2060–2072. https://doi.org/10.14778/3229863.3229874

Qbeast Analytics. (2022). Qbeast-spark: Multidimensional indexing for approximate query processing over Delta Lake.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster computing with working sets. *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*.

A

Scala Implementation Listings

This appendix reproduces the five source files that constitute the smart checkpointing implementation described in Chapter **??**. Line numbers refer to the `origin/index_delta_snapshot` branch of the `qbeast-spark` repository.

A.1. `IndexStatusDumper`

`IndexStatusDumper` serialises a `SortedMap[CubeId, CubeStatus]` to a four-column Parquet file at the canonical snapshot path. It is called by the write path in `IndexedTable` after every G-th commit (Listing A.5).

```
1  package io.qbeast.spark.delta
2
3  import io.qbeast.core.model.CubeId
4  import io.qbeast.core.model.CubeStatus
5  import io.qbeast.core.model.RevisionID
6  import org.apache.spark.sql.SaveMode
7  import org.apache.spark.sql.SparkSession
8
9  import scala.collection.immutable.SortedMap
10
11 /** Persists an IndexStatus snapshot to Parquet under
12  *  '<tableBasePath>/_qbeast/index_snapshots/revision=<id>/version=<ver>/'.
13  */
14 object IndexStatusDumper {
15
16   private[delta] def snapshotPath(
17       tableBasePath: String,
18       revisionID: RevisionID,
19       deltaVersion: Long): String =
20     s"$tableBasePath/_qbeast/index_snapshots/" +
21     s"revision=$revisionID/version=$deltaVersion"
22
23   /** Writes cube statuses for the given revision and Delta version
24    *  to Parquet.
25    */
26   def dump(
27       cubesStatuses: SortedMap[CubeId, CubeStatus],
28       tableBasePath: String,
29       revisionID: RevisionID,
30       deltaVersion: Long)(implicit spark: SparkSession): Unit = {
31
32     import spark.implicits._
33
34     val rows = cubesStatuses.toSeq.map { case (cubeId, status) =>
35       (cubeId.string,
36        status.maxWeight.value,
37        status.normalizedWeight,
38        status.elementCount)
39     }
```

```
40
41      val df = rows.toDF(
42        "cubeId", "maxWeightInt", "normalizedWeight", "elementCount")
43
44      df.write
45        .mode(SaveMode.Overwrite)
46        .parquet(snapshotPath(tableBasePath, revisionID, deltaVersion))
47    }
48  }
```

Listing A.1: IndexStatusDumper.scala

A.2. IndexStatusLoader

IndexStatusLoader scans the snapshot directory to find the highest persisted version $\leq v$ and then reads the Parquet file back into a SortedMap. The latestSnapshotVersion helper is called first in the dispatch inside loadIndexStatus (Listing A.3).

```
1  package io.qbeast.spark.delta
2
3  import io.qbeast.core.model.{CubeId, CubeStatus, Revision, RevisionID, Weight}
4  import org.apache.hadoop.fs.Path
5  import org.apache.spark.sql.SparkSession
6
7  import scala.collection.immutable.SortedMap
8  import scala.util.matching.Regex
9
10 /** Loads an IndexStatus snapshot previously written by IndexStatusDumper. */
11 object IndexStatusLoader {
12
13   private val VersionDirPattern: Regex = "version=(\\d+)".r
14
15   private[delta] def revisionDir(
16       tableBasePath: String,
17       revisionID: RevisionID): String =
18     s"$tableBasePath/_qbeast/index_snapshots/revision=$revisionID"
19
20   /** Returns the highest persisted snapshot version <= targetVersion,
21    *  or None if no snapshot exists.
22    */
23   def latestSnapshotVersion(
24       tableBasePath: String,
25       revisionID: RevisionID,
26       targetVersion: Long)(implicit spark: SparkSession): Option[Long] = {
27
28     val hadoopConf = spark.sessionState.newHadoopConf()
29     val revPath = new Path(revisionDir(tableBasePath, revisionID))
30     val fs = revPath.getFileSystem(hadoopConf)
31
32     if (!fs.exists(revPath)) return None
33
34     val statuses =
35       try fs.listStatus(revPath)
36       catch { case _: Exception => return None }
37
38     val versions = statuses
39       .flatMap { s =>
40         VersionDirPattern
41           .findFirstMatchIn(s.getPath.getName)
42           .map(_.group(1).toLong)
43       }
44       .filter(_ <= targetVersion)
45
46     if (versions.isEmpty) None else Some(versions.max)
47   }
48
49   /** Reads cube statuses from a Parquet snapshot directory. */
50   def load(
51       tableBasePath: String,
```

```
      revisionID: RevisionID,
      snapshotVersion: Long,
      revision: Revision)(implicit spark: SparkSession)
      : SortedMap[CubeId, CubeStatus] = {

    import spark.implicits._

    val path = IndexStatusDumper.snapshotPath(
      tableBasePath, revisionID, snapshotVersion)

    val rows = spark.read
      .parquet(path)
      .as[(String, Int, Double, Long)]
      .collect()

    val entries = rows.map {
      case (cubeIdStr, maxWeightInt, normalizedWeight, elementCount) =>
        val cubeId = revision.createCubeId(cubeIdStr)
        val status = CubeStatus(
          cubeId, Weight(maxWeightInt), normalizedWeight, elementCount)
        cubeId -> status
    }

    SortedMap(entries: _*)
  }
}
```

Listing A.2: IndexStatusLoader.scala

A.3. Modified `loadIndexStatus` dispatch

The four-branch dispatch in `DeltaQbeastSnapshot` is the routing logic described in Chapter **??**. Branch 1 fires when no snapshot exists; Branch 2 when the snapshot is exactly current; Branch 3 when the delta is a pure append; Branch 4 when the delta contains file removals.

```
override def loadIndexStatus(revisionID: RevisionID): IndexStatus = {
  val revision       = getRevision(revisionID)
  val currentVersion = snapshot.version
  implicit val spark: SparkSession = SparkSession.active

  IndexStatusLoader
    .latestSnapshotVersion(tableID.id, revisionID, currentVersion) match {

    // Branch 1: no snapshot -- full O(N) rebuild
    case None =>
      new IndexStatusBuilder(this, revision).build()

    // Branch 2: snapshot is current -- return directly, zero replay
    case Some(v) if v == currentVersion =>
      val statuses =
        IndexStatusLoader.load(tableID.id, revisionID, v, revision)
      IndexStatus(revision, statuses)

    // Branch 3: snapshot is stale, pure-append delta -- incremental merge
    case Some(snapVersion) =>
      val statuses =
        IndexStatusLoader.load(tableID.id, revisionID, snapVersion, revision)
      val base = IndexStatus(revision, statuses)
      val (deltaAppends, hasRemoves) =
        loadChangesBetween(revisionID, snapVersion + 1, currentVersion)
      val builder = new IndexStatusBuilder(this, revision)
      if (!hasRemoves) builder.buildIncremental(base, deltaAppends)
      // Branch 4: delta contains removes -- full rebuild for correctness
      else builder.build()
  }
}
```

Listing A.3: DeltaQbeastSnapshot.loadIndexStatus (extract)

A.4. IndexStatusBuilder.buildIncremental

buildIncremental merges a base IndexStatus with the files added since the snapshot. Correctness is proved in Chapter **??**: both sum (element counts) and min (weight thresholds) are distributive over disjoint file sets.

```
/** Merges a base IndexStatus with index files added after the base snapshot.
 *
 *  @param base        the baseline IndexStatus from the Parquet snapshot
 *  @param deltaFiles  files added since the snapshot (pure-append delta)
 *  @return            merged IndexStatus equivalent to a full build at v_c
 */
def buildIncremental(
    base: IndexStatus,
    deltaFiles: Dataset[IndexFile]): IndexStatus = {

  val deltaCubeStatuses = cubeStatusesFromFiles(deltaFiles)
  if (deltaCubeStatuses.isEmpty) return base

  val desiredCubeSize = revision.desiredCubeSize
  var activeStatuses  = base.cubesStatuses

  deltaCubeStatuses.foreach { case (cubeId, deltaStatus) =>
    val baseStatusOpt = activeStatuses.get(cubeId)

    // sum is distributive over disjoint blocks
    val mergedCount =
      baseStatusOpt.map(_.elementCount).getOrElse(0L) +
      deltaStatus.elementCount

    // min is distributive over disjoint blocks;
    // Weight.MaxValue is the neutral element for min
    val mergedWeight = Weight.min(
      baseStatusOpt.map(_.maxWeight).getOrElse(Weight.MaxValue),
      deltaStatus.maxWeight)

    val normalizedWeight =
      if (mergedWeight < Weight.MaxValue) mergedWeight.fraction
      else NormalizedWeight(desiredCubeSize, mergedCount)

    activeStatuses = activeStatuses.updated(
      cubeId,
      CubeStatus(cubeId, mergedWeight, normalizedWeight, mergedCount))
  }

  IndexStatus(revision, activeStatuses)
}
```

Listing A.4: IndexStatusBuilder.buildIncremental (extract)

A.5. Automatic dump trigger in IndexedTable

After every successful write transaction, IndexedTable checks whether enough commits have accumulated since the last snapshot and, if so, calls IndexStatusDumper.dump. The interval is controlled by MetadataConfig.SNAPSHOT_DUMP_INTERVAL (the gap parameter G).

```
// After the write transaction commits successfully:

val spark           = SparkSession.active
val freshSnapshot   = metadataManager.loadSnapshot(tableID)
// Single update() call -- avoids version/snapshot mismatch
val deltaVersion    = freshSnapshot.version

freshSnapshot.loadAllRevisions.filterNot(isStaging).foreach { revision =>
  val latestSnap = IndexStatusLoader
    .latestSnapshotVersion(
      tableID.id, revision.revisionID, deltaVersion)(spark)

  val commitsSinceSnap =
```

```
14      latestSnap.map(deltaVersion - _).getOrElse(Long.MaxValue)
15
16    if (commitsSinceSnap >= MetadataConfig.SNAPSHOT_DUMP_INTERVAL) {
17      // Use loadIndexStatus (not build()) so that when a prior snapshot
18      // exists and the delta is pure-append, the incremental path is
19      // taken here too -- O(K + mu*G) instead of O(N).
20      val writtenStatus =
21        freshSnapshot.loadIndexStatus(revision.revisionID)
22      IndexStatusDumper.dump(
23        writtenStatus.cubesStatuses,
24        tableID.id,
25        revision.revisionID,
26        deltaVersion)(spark)
27    }
28  }
```

Listing A.5: IndexedTable – dump trigger (extract)

B

Experiment 1 — Regression Data

Table B.1 reports every measurement used to fit the OLS line $T_{\text{baseline}}(N) = 0.353 \cdot N + 206\,\text{ms}$ ($R^2 = 0.950$, $n = 51$) shown in Figure 5.1 of Chapter 5. Each cell in the column *mean* is the arithmetic mean of three independent trials; *sd* is the sample standard deviation. All trials use profile D1, warm cache, mutation M0 (no structural changes between burn-in and timed measurement). The wide spread in N (73–5072 active files) is produced by varying the scale (S vs M), log depth ($D \in \{1, 10, 50, 200\}$), and cube size.

Table B.1: Experiment 1 raw data: baseline index-load time vs. active file count ($n = 3$ per N value; profile D1, warm cache).

N	n	mean (ms)	sd (ms)
73	3	375.7	56.4
82	3	321.3	58.8
92	3	321.7	98.1
122	3	294.3	6.0
172	3	305.0	22.9
272	3	341.7	4.9
572	3	447.0	5.0
585	3	291.7	18.0
594	3	355.7	103.0
604	3	298.7	12.4
634	3	379.7	109.4
684	3	403.7	135.5
784	3	372.7	17.0
1072	3	630.3	29.2
1084	3	482.3	25.0
2072	3	987.0	8.2
5072	3	2040.7	9.3
OLS fit: $T = 0.353N + 206\,\textit{ms}$, $R^2 = 0.950$, $n = 51$			

C

Experiments 2 and 3 — Per-Run Data

C.1. Experiment 2 — Amortised cost under mixed workloads

Tables C.1 and C.2 extend the median table in Chapter 5 (Table 5.3) with the arithmetic mean and sample standard deviation over all repeats ($n = 12$ per cell). Workload labels: R95 ($r = 0.95$), R80 ($r = 0.80$), R60 ($r = 0.60$), R40 ($r = 0.40$), R05 ($r = 0.05$). All measurements at Scale S, profile D1, warm cache. The ten gap values are split across two tables for readability.

Table C.1: Experiment 2: mean $\pm$ sd of total amortised cost (ms), gaps $G = 0$–10 ($n = 12$ per cell).

Wkld	G=0	G=1	G=2	G=5	G=10
R95	436 ± 153	2429 ± 321	1304 ± 127	658 ± 129	491 ± 82
R80	528 ± 39	2512 ± 332	1560 ± 234	994 ± 201	724 ± 115
R60	879 ± 65	2798 ± 370	1965 ± 458	1248 ± 186	1101 ± 134
R40	1190 ± 86	3116 ± 314	2051 ± 346	1470 ± 109	1547 ± 439
R05	1810 ± 146	3482 ± 779	2683 ± 379	2213 ± 892	2085 ± 277

Table C.2: Experiment 2 (cont.): mean $\pm$ sd of total amortised cost (ms), gaps $G = 20$–500.

Wkld	G=20	G=50	G=100	G=200	G=500
R95	406 ± 71	342 ± 51	302 ± 32	289 ± 27	282 ± 23
R80	649 ± 75	598 ± 73	569 ± 65	558 ± 67	533 ± 34
R60	1022 ± 133	971 ± 129	944 ± 89	895 ± 71	859 ± 60
R40	1354 ± 156	1255 ± 106	1272 ± 148	1263 ± 136	1222 ± 95
R05	2076 ± 592	1861 ± 175	1959 ± 517	1813 ± 138	1845 ± 157

C.2. Experiment 3 — Cost model validation

Table C.3 reproduces every individual trial from Experiment 3 (raw file `E8-000001/metrics.csv`). The experiment sweeps snapshot gap $G \in \{0, 50, 100, 200\}$ across three conditions (Scale S D=50, Scale S D=200, Scale M D=50), with five repeats per cell. Legacy (baseline) rows are included only at $G = 0$ since legacy has no snapshot and the gap parameter is not applicable. The fitted slope $\alpha = 21$ ms/commit is derived by OLS regression of the smart-path `index_load_ms` values against G, per condition.

Table C.3: Experiment 3 per-trial index-load times (ms). Legacy rows appear only at $G=0$.

Scale	D	G	Rep	index_load_ms	Method
Scale M, $D = 50$					
M	50	0	1	500	legacy
M	50	0	2	590	legacy
M	50	0	3	499	legacy
M	50	0	4	453	legacy
M	50	0	5	452	legacy
M	50	0	1	474	smart
M	50	0	2	609	smart
M	50	0	3	523	smart
M	50	0	4	537	smart
M	50	0	5	499	smart
M	50	50	1	1319	smart
M	50	50	2	1302	smart
M	50	50	3	1333	smart
M	50	50	4	1414	smart
M	50	50	5	1595	smart
M	50	100	1	2916	smart
M	50	100	2	2750	smart
M	50	100	3	2857	smart
M	50	100	4	2841	smart
M	50	100	5	2518	smart
M	50	200	1	4858	smart
M	50	200	2	4535	smart
M	50	200	3	4351	smart
M	50	200	4	4311	smart
M	50	200	5	4032	smart
Scale S, $D = 50$					
S	50	0	1	305	legacy
S	50	0	2	306	legacy
S	50	0	3	317	legacy
S	50	0	4	307	legacy
S	50	0	5	308	legacy
S	50	0	1	326	smart
S	50	0	2	350	smart
S	50	0	3	322	smart
S	50	0	4	322	smart
S	50	0	5	328	smart
S	50	50	1	1218	smart
S	50	50	2	1239	smart
S	50	50	3	1354	smart
S	50	50	4	1333	smart
S	50	50	5	1545	smart
S	50	100	1	2546	smart
S	50	100	2	2360	smart
S	50	100	3	2389	smart
S	50	100	4	2208	smart
S	50	100	5	2251	smart
S	50	200	1	4599	smart
S	50	200	2	4939	smart
S	50	200	3	4252	smart

Continued on next page

(Table C.3 continued)

Scale	D	G	Rep	index_load_ms	Method
S	50	200	4	4054	smart
S	50	200	5	4858	smart
Scale S, $D = 200$					
S	200	0	1	403	legacy
S	200	0	2	391	legacy
S	200	0	3	413	legacy
S	200	0	4	388	legacy
S	200	0	5	405	legacy
S	200	0	1	415	smart
S	200	0	2	447	smart
S	200	0	3	407	smart
S	200	0	4	402	smart
S	200	0	5	439	smart
S	200	0	1	400	smart
S	200	0	2	403	smart
S	200	0	3	386	smart
S	200	0	4	393	smart
S	200	0	5	398	smart
S	200	50	1	1634	smart
S	200	50	2	1594	smart
S	200	50	3	1466	smart
S	200	50	4	1401	smart
S	200	50	5	1409	smart
S	200	100	1	2401	smart
S	200	100	2	2418	smart
S	200	100	3	2311	smart
S	200	100	4	2184	smart
S	200	100	5	2306	smart
S	200	200	1	4965	smart
S	200	200	2	4706	smart
S	200	200	3	5148	smart
S	200	200	4	5365	smart
S	200	200	5	4920	smart