ParqOpt: Ingestion-Layer File Converter for Analytical Platforms

Kirill Dikalin
ITMO University
Saint-Petersburg, Russia
dikalinkirill@gmail.com

George Chernishev
Saint-Petersburg University
Saint-Petersburg, Russia
chernishev@gmail.com

Abstract*—In analytical platforms, the ingestion layer accepts files in multiple formats, validates them, and rewrites them into columnar formats while preserving reproducible write semantics under failures and retries. This paper presents ParqOpt, an ingestion service with a split architecture. The control plane accepts requests and persists state, while the worker plane performs conversion in isolated jobs using a C++ core. The pipeline uses streaming conversion on Apache Arrow with bounded buffers, backpressure, adaptive input slicing, and an explicit memory budget, providing controlled bounded-memory execution and predictable behavior for wide schemas and string-heavy data, with effects on absolute peak RSS depending on the dataset and memory budget. For S3-compatible storage, we implement multipart upload with a resume contract based on ListParts and ETag, including validation of uploaded parts and the expected offset of the next part. Correctness is confirmed by deterministic end-to-end tests, including negative resumption scenarios and integrity checks. The evaluation covers differences between local and S3 paths, a controlled internal comparison of planner decisions under fixed ingest semantics, and split-runtime scaling. Writing to object storage reduces throughput and increases end-to-end latency compared to the local path, while higher worker parallelism improves tail latencies and resilience under peak load. The internal comparison shows consistent differences in output size under the same pipeline, whereas throughput and memory effects depend on the input class. Additional confirmation runs extend the evaluation. Overall, the combination of a testable write contract, budget-aware streaming conversion, and split execution provides predictable ingestion behavior and a basis for optimizing columnar write parameters.

I. INTRODUCTION

In analytical platforms, file ingestion into object storage is usually implemented as a separate subsystem that accepts source files, records the write result, and prepares the data for efficient reads in target DBMSs and query engines [1]. In practice, this subsystem includes source connectors, an ingestion bus, object storage, and a consumer in the form of a DBMS or query engine. The overall scheme is shown in Figure 1.

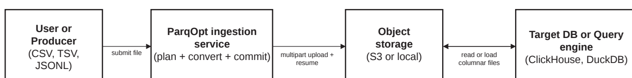

Fig. 1. Workflow of the service

In this paper, we focus on the ingestion bus. It accepts heterogeneous files and rewrites the data into a columnar format with parameters that directly affect output size, conversion speed, and read performance in target DBMSs. For operational applicability, reproducibility and write integrity are essential, because ingestion must behave correctly under repeated operation attempts, worker restarts, and partial failures along the write path.

The key risks and costs are concentrated in three areas. First, when working with object storage through the S3 API, large objects are uploaded via multipart upload [2], where the object is assembled from a set of parts. Under failures, the upload must be resumed correctly based on the confirmed parts and the expected offset of the next part, otherwise an integrity error in the resulting object may remain undetected at upload time. Second, the system uses streaming conversion, that is, sequential reading of the input file with transformation into a `RecordBatch` stream and writing into the target columnar format in object storage. Such conversion is memory-constrained and must maintain an upper bound on consumption as the schema and value distributions vary. Third, format and write-parameter selection is often performed manually or according to a single criterion, which leads to unstable results across different classes of input data.

As we show below in the review of practical solutions (Section III), existing systems do not usually provide, at the same time, a verifiable resume contract for multipart upload, guaranteed bounded-memory behavior for streaming conversion, and automatic selection of columnar write parameters.

To address the risks listed above and make ingestion reproducible under failures, we present ParqOpt, which implements ingestion as a service split into a split runtime and a C++ library core [3], schematically shown in Figure 2. ParqOpt accepts either a file or a link to a file as input, after which processing begins for a job that writes to object storage. The split runtime consists of two parts. The *Intake* component accepts the request, performs initial parameter validation, stores the input in a temporary object in storage when necessary, and prepares the job. The job is then transferred asynchronously to the *Execution* component, which performs processing and publishes the result. This separation decouples the fast request-admission path from long-running data-processing and write operations, allowing the service to scale execution independently of intake while preserving reproducible behavior under failures [4].

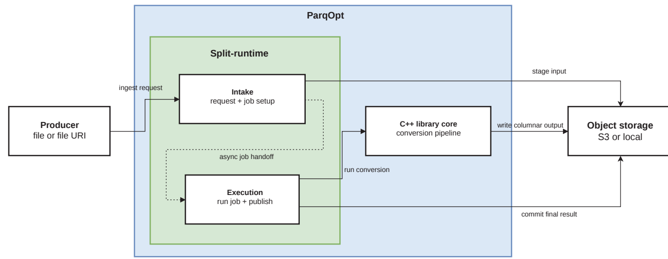

Fig. 2. Overall ParqOpt design with a split runtime and a C++ conversion library core, including Intake, Execution, and interaction with object storage

The main conversion is performed in the C++ library core invoked from *Execution*. Inside the core, input data are read sequentially, transformed into an intermediate tabular representation, profiled for write-parameter selection, and then rewritten into the target format. For JSON, the system uses streaming parsing based on simdjson [5], which makes it possible to process large documents without full materialization in memory. Apache Arrow [6] serves as the model for columnar writing, and the output can be produced in Parquet [7], ORC [8], Avro [9], and Arrow [10] formats.

At the output, ParqOpt produces columnar files in object storage, publishes the final job result, and makes the data available for subsequent reading or loading into target DBMSs and query engines.

To evaluate the proposed solution systematically, we formulate three research questions that cover the correctness of multipart-upload resume, the quality of automatic format and write-parameter selection, and the operational scalability of the split runtime. We then list the contributions implemented in ParqOpt separately.

This paper addresses the following research questions.

1) **RQ1.** Does an explicit resume contract for S3 multipart upload ensure correct and reproducible upload resumption under retries, restarts, and partial failures?
2) **RQ2.** How do planner decisions affect output size and conversion performance under fixed ingest semantics across different datasets and environments, and how stable are these trade-offs across different storage paths and streaming budgets?
3) **RQ3.** How does the split runtime behave under increasing load and under scaling of the control plane and worker plane, and how does this affect submit and completion tail latencies?

The overall contributions of the paper are as follows.

- For **RQ1**, we implement a formal and verifiable resume contract for S3 multipart upload in terms of the set of confirmed parts, their identifiers, the total confirmed volume, and the expected offset of the next part. The contract defines invariants for `commit` and `resume` and is checked by a QA suite in positive and negative scenarios, including missing parts, duplicates, part reordering, and offset desynchronization [11]–[13].
- For **RQ2**, we develop a mechanism for automatic selection of format and write parameters, evaluated through a controlled internal comparison with fixed and counterfactual planner settings. The results show consistent differences in output-file size under identical ingest semantics, whereas the effects on throughput and memory depend on the class of input data. In addition, an NDV-ratio-based encoding-selection rule is used to reduce the risk of degradation on high-cardinality columns and to automate the choice of format, row-group size, page size, and encodings without manual tuning.
- For **RQ3**, we describe a split runtime with separation between the control plane and the worker plane and evaluate the effect of the number of API replicas and workers on submit and completion tails, which provides practical guidance for autoscaling thresholds.

In addition, we implement bounded-buffer backpressure and adaptive `RecordBatch` size estimation based on periodic exact measurements and EWMA [14]; these mechanisms are used in the streaming conversion path and support budget-aware processing and predictable queue behavior for wide schemas and string-heavy columns.

The remainder of the paper is organized as follows. Section II introduces the necessary background. Section III discusses related work and practical ingestion solutions. Section IV describes the ParqOpt architecture, algorithms, and implementation details. Section V presents the experimental setup, testbeds, and metrics. Section VI formulates the evaluation questions and the logic of experimental validation. Section VII reports the experimental results. Section VIII discusses limitations, interpretation of the results, and practical implications. Section IX concludes the paper and outlines directions for future work.

II. Background

Modern ingestion systems include several components. They accept and validate input data, manage schema and types, provide reproducible publication of results to object storage, constrain memory consumption during streaming conversion, and select physical-layout parameters that determine the efficiency of subsequent analytical queries. Some of these tasks are addressed by industrial platforms, while others rely on research results and engineering practice in data formats, physical design, streaming processing, and commit semantics.

A. Object storage and the ingestion path

Modern analytical platforms often separate long-term storage from computation, with object storage serving as the central landing zone for raw and prepared datasets. In such architectures, a data-loading layer is typically placed in front of object storage to accept heterogeneous files, ensure the integrity and reproducibility of writes, and transform the data into layouts that enable efficient scans and predicate push-down. The design space here is constrained by the semantics of object storage, where file-like operations such as rename [15]

are usually implemented as copy followed by delete and can add non-atomic O(data volume) overheads to commit paths.

B. Columnar formats and physical design

Rewriting at ingestion time is motivated by the gap between row-oriented interchange formats and columnar formats for analytics. Columnar formats such as Parquet and ORC [16] store values by column and add structure in the form of pages, statistics, and indexes. This improves compression and allows selective reading of only the required fragments. In Parquet, a key role is played by the set of encodings and the page organization within a `column chunk`. Dictionary encoding stores the dictionary in a `dictionary page` and then writes indexes using the `RLE/Bit-Packing Hybrid`, with fallback to `plain` when the dictionary is not beneficial [17]. Predicate pushdown relies on row-group statistics and the page index, making it possible to discard parts of the data before reading [18]. ORC organizes data into `stripes` and uses multi-level indexes and statistics, including bloom filters, which accelerate data elimination during filtering [8]. For Avro, `schema-on-read` and `schema resolution` rules are central. Correct reading depends on the `writer schema` and the `reader schema`, while union-type restrictions affect schema compatibility and processing cost [9]. As a result, file layout is treated as an optimizable object because it directly affects size, write speed, and read efficiency.

In practice, physical design is determined by writer parameters, most notably row-group size, page size, and column-level encoding decisions. Parquet tuning recommendations emphasize the trade-off involved. Larger row groups improve sequential I/O and amortization of metadata overhead, but at the same time increase buffering and memory requirements along the write path [19]. Analytical engines are sensitive to layout and usually benefit in terms of parallelism when the file is divided into a sufficient number of independent reading units [20]. This motivates automated planning that adapts parameters to data characteristics and resource budgets.

In addition, physical design is affected by partitioning and target file size. These determine the trade-off between the number of objects and metadata cost, as well as the degree of downstream read parallelism.

C. Streaming conversion and memory constraints

A common intermediate representation for ingestion pipelines is an in-memory columnar format that reduces the overhead of transformations between readers and writers. Apache Arrow defines an in-memory columnar layout and IPC serialization formats that are relocatable and allow buffers to be shared across components without copying (zero-copy) [6]. Using RecordBatch-like units as the internal transfer object makes it possible to build pipelines as compositions of operators while delegating low-level buffer management to a standardized model.

JSON loading often becomes a performance bottleneck because documents may be large, nested, and heterogeneous, and schema inference frequently requires scanning a substantial fraction of the payload. Validating parsers with SIMD acceleration [5] show that substantial speedups are possible. Published work on simdjson demonstrates parsing at gigabytes-per-second rates on commodity processors through the use of SIMD instructions and careful validation design. This supports streaming ingestion, where parsing, profiling, and rewriting can be pipelined without full materialization of the source document in memory.

D. Write and resume semantics

The semantics of writing to object storage strongly affect the design of ingestion systems. Multipart upload in S3 splits a large object into independently uploadable parts, and completing the upload requires providing the full ordered list of part numbers and their ETags, after which the service concatenates the parts. Obtaining the list of already uploaded parts is a first-class API operation and enables explicit resumption strategies. At the same time, object rename is implemented as copy followed by delete, and the Hadoop S3A committer documentation emphasizes [15] that renaming many objects is non-atomic and may leave an ambiguous state after failures. This motivates the use of staging, strict validation during resume, and failure-aware commit strategies in ingestion pipelines.

E. Evaluation methodologies

Finally, benchmarking layout optimization at ingestion time requires separating conversion cost from downstream query-side benefits. Industrial decision-support benchmarks such as TPC-DS [21] and TPC-H [22] provide representative query sets and data-generation methodologies and are widely used as datasets even when the official benchmark rules are not followed in full. A practical evaluation design therefore combines end-to-end ingestion throughput and tail-latency metrics with measurements of scan and query-execution time on the selected engine, allowing transparent analysis of the trade-off between ingestion overhead and query-side time savings.

III. Related Work

The following section summarizes practical ingestion solutions and closely related systems that address individual parts of the overall path. These approaches establish the baseline level of guarantees and show which properties are usually left to the user or appear only as configuration best practices.

A. Landscape of practical ingestion solutions

In industry, ingestion is commonly implemented in two dominant ways. The first is through managed cloud services and SaaS platforms, which take responsibility for delivery and scaling and in some cases also provide conversion into columnar formats. For example, Amazon Data Firehose supports conversion of input JSON into Parquet or ORC before writing to S3, using schema information from AWS Glue Data Catalog [23], [24]; AWS Glue is used as managed ETL to rewrite data in S3 into Parquet [25], [26]. In other cloud solutions, ingestion is often implemented as loading

into the DBMS internal storage. BigQuery supports batch load jobs and the Storage Write API for streaming and batch loading [27], [28], while Azure Data Explorer ingests data into tables via mappings for CSV/JSON/AVRO formats and columnar Parquet/ORC formats [29], [30]. The category of managed ingestion and ELT platforms also includes solutions aimed at loading into object storage and the lakehouse stack, such as Rivery and Onehouse OneFlow [31]–[33].

The second path is a self-assembled stack of connectors and frameworks such as Kafka Connect and NiFi [34], [35], in which ingestion into object storage is constructed as a composition of delivery, intermediate processing, and rewriting. This approach provides flexibility but shifts to the user the burden of correct object-storage write semantics under failures, memory control in streaming conversion, and layout-parameter selection [35], [36]. Table I summarizes the characteristic properties of common solutions and shows that detailed and verifiable multipart-upload resume contracts, bounded-memory streaming conversion, and automatic selection of columnar write parameters are usually not guarantees of existing ingestion platforms. Even when a vendor explicitly specifies delivery or ingestion semantics, properties at the level of multipart-upload resume and bounded-memory conversion are more often left undescribed and depend on the implementation.

B. Closely related practical systems and post-ingestion optimization

The closest practical baseline solutions for conversion at ingestion time can be divided into several classes. Managed cloud services provide conversion to columnar formats as part of delivery pipelines. For example, Amazon Data Firehose [37] can convert input JSON to Parquet or ORC before storing it in S3, using schema information from AWS Glue tables [38], [39]. AWS Glue often serves as a managed ETL foundation for converting raw S3 data to Parquet through Spark-based jobs [40], and official prescriptive patterns describe this process as a standard workflow. These solutions focus on managed scalability and integrations, while layout choices and column-level encoding decisions are typically determined by defaults or user configuration, without explicit data-driven planning at ingestion time.

Another class of baseline solutions is provided by connector ecosystems and streaming-ingestion frameworks. Kafka Connect [41] defines a model of source and sink connectors and is widely used to move data between Kafka and external systems, including object storage. The Confluent documentation for the S3 sink describes configurations [42] in which output formats include Parquet, which conceptually corresponds to event-driven ingestion into object storage. Apache NiFi offers a flow-based ingestion model with explicit queue back-pressure thresholds, which relates to backpressure in bounded-buffer designs [43]. However, such systems primarily address reliable delivery and operational convenience, leaving fine-grained file-layout optimization to later processing stages.

A closely related family of systems targets file optimization after ingestion at the table level in order to mitigate the small-files problem and improve query efficiency. Apache Iceberg documents `rewriteDataFiles` for compacting small data files [44]. Delta Lake provides OPTIMIZE-style rewrites to improve layout [45]. Apache Hudi applies compaction services in merge-on-read designs to merge delta logs into columnar base files [46]. These systems operate after ingestion, usually within transactional table abstractions, whereas an optimizer at ingestion time aims to place data into an efficient layout as soon as they enter the lake, potentially reducing the need for later rewrites.

TABLE I. LANDSCAPE OF PRACTICAL INGESTION SOLUTIONS

System	Mdl	Mode	Obj	Schema	Layout	Ctr
ParqOpt	SH	B/S	+	+	+	+
Amazon Data Firehose	MC	S	+(J→P/O)	Glue	-	±
AWS Glue	MC	B/S	+	Cat/Cr	cfg	-
Rivery	SaaS	B/S	+(Parq→S3)	dep(conn)	-	-
Onehouse OneFlow	MC	B/S	+(tbl fmt)	±(tbl)	±(tbl)	-
BigQuery	MC	B/S	n/a(tbl)	+(auto)	n/a	±
Azure Data Explorer	MC	B/S	n/a(tbl)	+(map)	n/a	±
Open-source stack	SH	B/S	±	dep(comp)	man	dep(impl)

Notation.

Mdl, Mode	deployment model, operating mode
Obj, Schema	object-storage output, schema management
Layout, Ctr	layout autotuning, documented write semantics
SH, MC, B, S	self-hosted, managed cloud, batch, stream
+, ±, -, n/a	available, partial or limited, undisclosed, not applicable
cfg, man, dep	via configuration, manually, depends on the implementation
Cat/Cr	Data Catalog, Crawlers
J→P/O	JSON→Parquet/ORC
Parq→S3	Parquet to S3
tbl fmt	table formats
auto, map	autodetect, mappings
dep(conn), dep(comp), dep(impl)	depends on the connector, depends on the components, depends on the implementation

IV. ARCHITECTURE AND IMPLEMENTATION

Architecturally, ParqOpt is divided into several subsystems. The key elements include URI and filesystem I/O abstractions for uniform access to `file://`, `s3://`, and optionally `hdfs://`; a registry of input and output formats; a layer that builds the canonical tabular representation as a `RecordBatch` stream; a profiler for columns and file char-

Fig. 3. Stages of the `Pipeline Core` pipeline.

acteristics; a writer-parameter planner based on a simulator and cost model; a rewriter; and a service runtime with a task queue and a job-state store.

At the top level, the system follows the control-plane and worker-plane scheme [3], [47]. Clients submit requests over REST, either with a data body or with a source link. The control plane validates parameters, materializes the input into a temporary staging object when necessary, creates a job record in the `Job Store`, and publishes the task identifier to the `External Queue`. The worker extracts jobs from the queue, updates execution states in the `Job Store`, and invokes the `Pipeline Core` library core. The core reads input by URI, builds a batch stream, computes the profile, selects representation parameters, and rewrites the data into the target format. For ingestion scenarios, commit of artifacts to the target path and cleanup of temporary objects are performed after rewriting. Observability is separated into a dedicated plane, and both processes export stage telemetry, with metrics available through `GET /v1/metrics`.

Inside `Pipeline Core`, processing is organized as a five-stage pipeline, shown in Fig. 3. First, the input is opened by URI and converted into a unified read stream for the parsers. Parsing is then performed and a `RecordBatch` stream is constructed as the internal canonical format. Next, profiles and statistics are computed for columns and the file and are used to choose layout and encoding parameters. At the next stage, the planner selects write parameters based on the profile, the simulator, and the cost model. The final stage rewrites the data into the target format and produces a write report. Correctness is enforced at stage boundaries through schema-compatibility checks, admissible type conversions, and validation of ingestion contracts. At the service level, strict job-state transitions and artifact connectivity through the task identifier are additionally enforced.

A. Streaming upload, parallel profiling, and backpressure

Service-side ingestion uses a two-thread design. The upload thread writes the object to staging, while profiling of the same byte stream is optionally launched in parallel through a bounded pipe [48]. Staging is a temporary object or a temporary prefix in the same storage backend, to which the input or intermediate result is written until the pipeline finishes; after successful completion, commit to the final key and cleanup of temporary artifacts are performed. The pipe buffer is bounded in total data volume. Once the limit is reached, the producer blocks until the consumer releases space, implementing the classic bounded-buffer pattern and ensuring an upper bound on peak memory consumption at this stage [49].

To couple contract correctness with ingestion reproducibility, the system applies integrity control via incremental CRC32 [50], [51]. The checksum is updated on every accepted chunk and compared with the expected value at the end. On mismatch, the system terminates the operation as invalid and may preserve the partial multipart upload in S3 under the preserve-partial-upload mode [13], [52]. This policy is consistent with the hard-limits and SLO-target philosophy, in which deterministic reaction to integrity violations is important [53].

The ingestion streaming path provides a fallback to post-upload profiling. If parallel profiling is interrupted or the streaming reader loses correctness at block boundaries, profiling is repeated from the staging object. This path uses the backend filesystem, where buffering and seek semantics are more convenient, which improves robustness without complicating the main fast path.

B. Profiling and planning features

Profiling computes a set of features for columns and the file. These include volume and the number of values, the fraction of missing values, an estimate of uniqueness, repeatability indicators, delta characteristics for numeric types, and simple entropy proxies for floats. These features are used to select encoding classes and layout parameters. In particular, for Parquet dictionary encoding, the dictionary is stored in the dictionary page and indexes are written using the RLE/bit-packed hybrid [17].

In ParqOpt, uniqueness receives special attention because it determines the usefulness of the dictionary and the risk of writer-path degradation. The planner estimates the ratio of unique values to the number of values and selects either `dictionary encoding` or `plain encoding` [17] for string columns, taking into account the optimization target mode [54]. A stricter choice is applied in modes where throughput and tail latencies are important, because large dictionaries increase CPU cost, memory usage, and compression sensitivity. A softer choice is admissible in the disk-saving mode, where output size is the priority.

Layout parameters, primarily `row group` size and page size, are selected within practical ranges and scaled according to the write target. The constraints are defined by the trade-off among metadata amortization and compression cost, the writer bounded-memory behavior, and read granularity. Excessively large `row groups` reduce the number of parallelization units and may worsen scan tail latencies, whereas excessively small ones increase the metadata fraction and the number of seeks. Similarly, excessively small pages increase overhead and the frequency of compression, whereas excessively large pages increase the decode time for a single page and reduce the effectiveness of data skipping under predicate pushdown because of the coarser page-index granularity.

C. Cost functions, scoring, and winner selection

The project uses two forms of objective function intended for different levels of decision making.

The first form is used in the CLI auto-optimization tool, which compares several target formats relative to the input baseline. For each candidate, a single score is computed as a weighted combination of three normalized components. The

size component reflects how much smaller or larger the output is than the baseline. The throughput component reflects how much faster or slower conversion is than the baseline. The latency component reflects how latency changes. The winner is the candidate with the minimum score under the given weights, where the weights define the trade-off among disk savings, conversion speed, and latency.

The second form relates to column-level planning and is implemented in the cost module. Here, the size contribution, the approximate CPU cost of encoding and compression, the potential read benefit from indexes and skipping, and a proxy for read parallelism associated with the number of `row groups` and the expected scan granularity are estimated. These estimates are aggregated into the plan cost, after which the configuration with the best trade-off is selected. In this formulation, parallelism is treated as a factor that reduces effective read latency when the partitioning granularity is sufficient, which is consistent with practical execution of scans over columnar files.

For ORC, similar sources of read-side gains are associated with file-, stripe-, and row-group-level indexes, including statistics and bloom filters [55], [56]. For Avro, schema compatibility is central [9]. The schema is the canonical Arrow schema inferred during parsing; it is used to verify admissible transformations and to construct the writer schema of the target format. For full JSON, type conflicts may arise within one field; in Avro, this is expressed through unions, which affects compatibility and processing cost. Binary data do not carry full type information, correct reading depends on the writer schema, and type evolution together with union types directly affects the handling of type conflicts, which is important for full-JSON scenarios.

D. Rewriting with a memory budget and batch-memory estimation

The rewriter uses an explicit memory budget, which is critical for streaming execution on mid-sized machines. The budget is applied when choosing the CSV read-block size, the prefetch depth for batches, and the limits of internal queues between producer and consumer. A practical heuristic ties the read-block size to a fraction of the budget, for example about one eighth, with hard lower and upper bounds in the range of tens of kilobytes and several megabytes. This maintains a balance between the overhead of overly small blocks and the risk of memory inflation from overly large ones.

A key part of the implementation is estimating the size of a `RecordBatch` without performing a full serialization pass for every batch. An adaptive estimator is used. It periodically performs an exact measurement, by default once every 16 batches, and then updates a smoothed estimate of bytes per row via EWMA (exponentially weighted moving average). Between exact measurements, the most recent smoothed estimate is used, which reduces the cost of estimation and provides a stable signal for adaptive input slicing and backpressure tuning.

The exact measurement relies on traversing `ArrayData` [6], [57] and summing the sizes of the unique buffers in the batch, including the dictionary. It is important to account for shared buffers and dictionary structures so that the estimate is not inflated by repeated counting of the same buffers in the presence of references and nested structures. The resulting estimate is used to decide the size of the next batch and the admissible queue depth, which helps preserve bounded-memory rewriting behavior for wide schemas and string columns.

E. Resume semantics for S3 multipart upload

For S3-compatible staging and commit objects, the correctness of resumption is determined by the consistency between the parts already uploaded in a multipart session and the expected offset from which the upload must continue. At the start of resume, the system uses the stored multipart-upload token, restores the context of the unfinished upload, requests the list of confirmed parts from storage via `ListParts`, extracts their identifiers (ETags), and computes the total confirmed data volume on the S3 side [13]. If an expected resume offset in bytes is specified for the operation, it is checked strictly against the sum of the confirmed part sizes. A mismatch is treated as state desynchronization and leads to refusal to continue uploading within the current session.

After successful validation, uploading continues from the next part. Upon completion of the multipart upload, an ordered list of parts with their part numbers and ETags required for `CompleteMultipartUpload` is formed, after which server-side assembly of the object is performed by concatenating the parts [52]. Such a contract makes resume verifiable and reduces the risk of silent integrity errors under retries, restarts, and partial failures.

V. EXPERIMENTAL DESIGN

The experimental design links the three research questions to observable system properties. The first group of experiments examines ingestion correctness and reproducibility under retries and partial failures. The second group examines the effects of the storage environment, planner decisions, and the streaming mode on output size, throughput, and memory under fixed ingest semantics and different operational budgets. The third group examines scaling of the split runtime with respect to the number of control-plane and worker-plane replicas and is reflected in tail latencies.

The term matrix is used to denote a systematic enumeration of configurations as a Cartesian product of factors, for example storage backend, input size, and data scenario. Each combination of factors forms a separate run case with fixed parameters and measurable metrics.

A. VM testbed for the E2E split runtime

The testbed includes Ubuntu 20.04, 8 vCPU, and 32 GiB RAM. Execution is organized as the split architecture `parqopt_service + parqopt_worker`. The `parqopt_service` process implements the control plane

and is responsible for request admission, validation, job creation, writing state to the job store, and publishing the task to the queue. The `parqopt_worker` process implements the worker plane and executes the conversion pipeline, from input loading and parsing to result writing.

Redis (`BLPOP`) [58] is used as the queue, and the job store is implemented on SQLite [59]. For object storage, an S3-compatible backend based on MinIO [60] is used. Two environment configurations are used for E2E runs. `local` uses a local storage backend, and `S3` uses S3-compatible storage and additionally exercises multipart upload and resume semantics. Hereafter, the local path denotes staging and commit to the local filesystem, while the S3 path denotes staging and commit to S3-compatible object storage.

The E2E experimental set includes the following runs.

- A QA suite for validating ingestion contracts. This is a set of deterministic end-to-end tests exercised through the public API that define positive and negative scenarios and validate expected job outcomes.
- A local-vs-S3 matrix, in which the storage backend and input size are varied. The purpose of this matrix is to evaluate the effect of object storage on end-to-end latency and throughput for the same input data.
- A scaling matrix over the number of control-plane and worker-plane replicas. The purpose of this matrix is to measure the effect of scaling on submit and completion tail latencies.
- Soak testing [61] to evaluate the resilience of the control plane and the job store under sustained concurrent load.

1) Data scenarios: The experimental matrices use three input-data scenarios that differ in value distributions and schema structure. This matters because encoding and write-parameter decisions depend not only on column statistics after parsing the input into Arrow, but also on schema width and the dominant data types.

Let the input contain N rows and a set of columns $c \in \Sigma$. For each column, we define the cardinality ratio

$$r_c = \frac{\mathrm{NDV}(c)}{N}.$$

where $\mathrm{NDV}(c)$ is the number of distinct values in column c.

- `typical`. A typical tabular input containing low-cardinality categorical fields and numeric measures.
- `high_cardinality`. An input dominated by columns for which $r_c \approx 1$.
- `wide_string_heavy`. An input with a wide schema and a predominance of string fields, creating higher pressure on buffering, encodings, and write-parameter selection.

B. Local testbed for format experiments and the streaming path

A separate local testbed is used for runs in which the primary evaluation object is format selection and the effect of the streaming mode on memory. Several types of micro-benchmark inputs are used, as well as a streaming load test to compare `stream_off` and `stream_on` in terms of max RSS [62], [63] and time. Here, `stream_on` enables the streaming rewriting mode with bounded-memory constraints, while `stream_off` uses the baseline mode without these constraints for controlled comparison. The goal of the streaming matrix is to evaluate budget-aware operational behavior and the dependence of max RSS and throughput on the scenario and memory budget, rather than to claim that streaming monotonically reduces absolute peak RSS for every configuration. The streaming load test is a controlled pipeline profile that checks the effect of the streaming mode on max RSS and conversion time for controlled inputs and verifies that the streaming path preserves end-to-end conversion. By streaming path, we mean a conversion mode in which the input is processed incrementally and passed through the pipeline via bounded queues, which maintains an upper bound on memory consumption.

C. Metrics

The experiments record metrics that cover the split-runtime E2E service behavior and the quality of the conversion result.

1) **Split-runtime tail latencies.** The quantiles $submit\ p95$ and $completion\ p95$ are measured. The submit metric belongs to the control plane and measures the latency from request arrival to the moment when the job has been created, the state has been persisted in the job store, and the task has been published to the queue. The completion metric belongs to the worker plane and measures pipeline execution time on the worker from the start of job processing to the terminal state.
2) **E2E latency.** The time from request arrival to obtaining the final result is measured.
3) **Ingest throughput.** Let T_{local} and T_{s3} denote conversion throughput in MB/s in the `local` and `S3` environments.
4) **E2E latency for environment comparison.** Let L_{local} and L_{s3} denote p95 end-to-end latency in milliseconds in the `local` and `S3` environments.
5) **Memory.** Max RSS of the process is recorded for the streaming runs.
6) **size_ratio.** The ratio of output size to input size, $size_ratio = \mathrm{bytes}(out)/\mathrm{bytes}(in)$.

For concise environment comparison, degradation ratios are used

$$\rho_T = \frac{T_{local}}{T_{s3}}, \qquad \rho_L = \frac{L_{s3}}{L_{local}}.$$

Here, ρ_T shows by how many times throughput in the `S3` environment is lower than in `local`, while ρ_L shows by how many times end-to-end latency in `S3` is higher than in `local`. In our runs, both ρ_T and ρ_L are greater than 1 in all cases, which indicates consistent degradation of the S3 path. For example, $\rho_T = 2.965$ means $T_{s3} \approx T_{local}/2.965$, while $\rho_L = 2.950$ means $L_{s3} \approx 2.95 \cdot L_{local}$.

VI. Evaluation

The evaluation follows the three research questions and links each experiment to verifiable system properties.

TABLE II. QA RESULTS FOR INGESTION CONTRACTS

Target	Total	Passed	Notes
local	11	10	one failed resume case
S3	11	11	all checks passed

TABLE III. LOCAL VS S3, DEGRADATION RATIOS

Scenario	Size	ρ_T	ρ_L
typical	100MB	2.965	2.950
typical	1GB	3.455	3.451
high_cardinality	100MB	4.092	4.022
high_cardinality	1GB	3.474	3.451

RQ1. Correctness of ingestion contracts and resilience to failures. The effects checked include the QA suite, including idempotency [64], response to `checksum mismatch`, enforcement of `size limit`, correctness of resume, and reaching a terminal job state.
RQ2. Effect of planner decisions, storage environment, and streaming mode on the ingestion result. The effect of the storage backend is evaluated through the local-vs-S3 matrix and the degradation ratios ρ_T and ρ_L. A controlled internal comparison of planner decisions under fixed ingest semantics is conducted separately. For the streaming mode, the dependence of max RSS and throughput on the scenario, input size, and memory budget is evaluated.
RQ3. Operational scalability of the split runtime. The behavior of the control plane and the worker plane is evaluated as the number of replicas scales. The main metrics are $submit\ p95$ and $completion\ p95$. In addition, the resilience of the job store and control plane is examined under soak testing.

VII. EXPERIMENTAL RESULTS

A. Question 1, ingestion correctness and resilience to failures

1) QA results for ingestion contracts: Table II summarizes the QA results for ingestion contracts. For both targets, 21 out of 22 end-to-end checks completed successfully. The only unsuccessful case was related to a resume-offset mismatch on the local append path. All QA checks for S3 passed successfully, including negative resume scenarios and integrity checks. Therefore, the remaining correctness gap is limited to the local append-resume path and does not affect the evaluated S3 multipart-resume contract.

B. Question 2, size-performance trade-off, environment effects, and streaming mode

1) Local vs S3 matrix and degradation ratios: To diagnose the effect of object storage, we use the degradation ratios ρ_T and ρ_L. Table III shows consistent degradation in throughput and end-to-end latency for the S3 path at medium input sizes. This is consistent with the fact that staging and commit over object storage involve copy+delete operations.

TABLE IV. AUTOMATIC PLANNER RELATIVE TO FIXED DEFAULT WRITER SETTINGS

Scenario	Rows	Δ size ratio	Δ thr	Δ max RSS
high_cardinality	1M	−0.0017	−4.41	3.46
high_cardinality	5M	−0.0017	−1.57	−18.32
typical	1M	−0.0031	0.00	−0.14
typical	5M	−0.0031	−2.32	−4.82
wide_string_heavy	1M	−0.0029	−17.56	50.20
wide_string_heavy	5M	−0.0029	−0.61	126.94

2) Controlled internal comparison of planner decisions under fixed ingest semantics: To evaluate the planner effect without conflating it with semantic differences between external tools, we conducted a controlled internal comparison of 24 cases on the same ingestion pipeline, the same datasets, and the same machine. The comparison covered the automatic planner, fixed default writer settings, and two additional variants in which NDV-based dictionary selection was disabled or row-group and page targets were fixed. All 24 runs completed successfully. Table IV reports the central matched comparison, namely the automatic planner relative to the fixed default configuration.

The table reports automatic planner minus fixed default. Negative values are preferable for size ratio and max RSS, whereas positive values are preferable for throughput. The automatic planner produced a smaller output in all six matched cases, with size-ratio improvements ranging from 0.0017 to 0.0031. The effects on throughput and max RSS depended on the scenario. For `typical`, the planner remained close to the fixed baseline while preserving the gain in output size. For `high_cardinality`, it improved size ratio in both matched cases and reduced max RSS in the larger run. For `wide_string_heavy`, it also improved size ratio, but this came with lower throughput and higher max RSS. These results show that planner decisions measurably affect layout outcomes within the same ingestion task, even though the direction of the trade-off depends on the input-data class.

3) Streaming matrix by memory budget and input size: To evaluate the behavior of the streaming path beyond a single controlled example, we ran a matrix of 96 runs across three scenarios, four input sizes, four memory budgets, and two execution modes. All 96 runs completed successfully. Thus, the streaming path was exercised across a broad configuration space.

Table V aggregates the results of 48 matched comparisons across the three scenarios. For each scenario, it shows in how many of the 16 paired comparisons `stream_on` relative to `stream_off` yielded lower max RSS, higher max RSS, and higher throughput. Across all three groups, the median throughput delta remained positive and was +0.76 MiB/s for `typical`, +3.41 MiB/s for `high_cardinality`, and +4.21 MiB/s for `wide_string_heavy`. This shows that enabling the streaming path in the studied matrix was not accompanied by a systematic degradation in processing speed.

The effect on max RSS proved to depend on the sce-

TABLE V. STREAMING MATRIX SUMMARY ACROSS MATCHED PAIRS

Scenario	Lower max RSS	Higher max RSS	Higher throughput
typical	6	10	9
high_cardinality	5	11	11
wide_string_heavy	5	11	11

nario and the memory budget. Across all 48 matched pairs, stream_on reduced max RSS in 16 cases and increased it in 32 cases. The most notable reductions were observed in some configurations with wide string-heavy schemas; for example, max RSS decreased by 49.75 MiB for wide_string_heavy at 1M rows and a 32 MiB memory budget. Therefore, the results indicate that the streaming path should be interpreted not as a mechanism that universally minimizes absolute peak RSS, but as an execution mode with an explicit memory budget and bounded queues.

Taken together, the matrix supports a more precise formulation of the bounded-memory claim. The streaming path provides controlled and robust execution across the entire tested configuration space and makes it possible to specify an explicit memory budget for processing. Its primary effect is control over the execution mode while preserving stability and, in many configurations, positive throughput dynamics. A reduction in absolute max RSS is possible and is observed in some configurations, but it is not a universal result for every combination of scenario, input size, and memory budget.

4) External reference point based on common conversion libraries: Table VI complements the internal planner comparison with an external calibration of absolute output-size ratio and conversion-throughput values on fixed datasets written to S3. It shows the range of final size and conversion speed in which ParqOpt falls relative to DuckDB and PyArrow on the same input data. Because these systems differ in end-to-end ingest semantics, the table is not used to infer planner superiority. Conclusions about the effect of the planner are drawn from the controlled internal comparison in Table IV.

5) Additional runs beyond the baseline VM configuration: The main split-runtime matrix was executed on the 8 vCPU and 32 GiB RAM VM described in Section V. To check the applicability of the results beyond this baseline configuration, we additionally confirmed artifact generation in YC Object Storage for 1 GB and 5 GB inputs in the high_cardinality and wide_string_heavy scenarios. We also performed local confirmation experiments at 20 GB for the same two scenarios. Table VII summarizes the local results for 20 GB.

These additional runs show that the observed behavior is not limited to only the smallest demonstration inputs. At the same time, they do not replace a separate evaluation on distributed object storage.

C. Question 3, operational scalability of the split runtime

1) Scaling matrix, submit and completion tails: The scaling matrix reflects the effect of scaling on tail latencies. Table VIII shows that increasing the number of workers reduces $completion\ p95$ because of worker-plane parallelism, most clearly in the transition from 2 to 8 workers. At the same time, $submit\ p95$ remains constrained by control-plane overhead and serialization of job-store operations on the current testbed, so increasing the number of replicas does not yield a linear gain and in some configurations increases queue-submission tail latencies.

2) Soak testing of the control plane: The soak test includes 700 requests in total, of which 696 were accepted and 4 were rejected. The case-level results are summarized in Table IX. In all 4 cases, the cause of rejection was the sqlite database is locked error, which corresponds to contention for the single writer in SQLite as the number of concurrent requests increases.

VIII. DISCUSSION

A. Question 1, ingestion correctness and resilience to failures

The QA suite covers positive and negative scenarios in which it is important to reproducibly reach correct terminal job states under retries and partial failures. Expressing ingestion contracts as explicit testable invariants reduces the risk of incorrect resume behavior, which is difficult to detect in load runs. The remaining local unsuccessful case shows that the local append-resume path is currently less mature than the evaluated S3 multipart-resume path and should be regarded as a limitation of the current implementation.

B. Question 2, size-performance trade-off, environment effects, and streaming mode

The local-vs-S3 matrix shows that moving to object storage causes consistent degradation in throughput and end-to-end latency at medium sizes. This is consistent with the fact that staging and commit operations over S3-compatible storage add network and object overheads that are absent on the local path.

The extended streaming matrix shows that the effect of stream_on on absolute max RSS depends on the scenario and the memory budget. In 16 out of 48 matched comparisons, streaming reduced max RSS, whereas in 32 comparisons it increased it. At the same time, throughput improved in 31 out of 48 comparisons. Therefore, the data support the claim of controlled budget-aware execution and stable pipeline behavior across different classes of input data, but not the claim of universal peak-RSS reduction.

The controlled internal comparison of the planner provides the most appropriate basis for conclusions about its contribution, because it fixes the ingestion pipeline and changes only the planner decisions. In this setup, the automatic planner consistently improves output size relative to fixed default writer settings, although for wide_string_heavy degradations in throughput and max RSS remain. This shows that the quality of planner decisions is measurable and practically useful,

TABLE VI. EXTERNAL S3 REFERENCE, SIZE RATIO AND THROUGHPUT

Dataset	Target system	size_ratio	throughput (MB/s)
tpch_sf1	ParqOpt	0.495	20.14
	DuckDB	0.209	25.07
	PyArrow	0.135	24.71
synth_wide_w80_256mb	ParqOpt	0.797	11.54
	DuckDB	0.367	15.70
	PyArrow	0.155	16.23
json_nested_128mb	ParqOpt	0.209	11.02
	DuckDB	0.148	14.31
	PyArrow	0.044	11.18

TABLE VII. LOCAL CONFIRMATION RUNS AT 20 GB

Scenario	Throughput (MiB/s)	Max RSS (MiB)	Size ratio
high_cardinality	51.33	633.55	0.2873
wide_string_heavy	35.33	549.87	0.3176

TABLE VIII. SCALING MATRIX, P95. CP IS THE NUMBER OF CONTROL-PLANE REPLICAS. WORKERS IS THE NUMBER OF WORKER THREADS IN THE WORKER PLANE.

CP	Workers	Submit p95 (s)	Completion p95 (s)	Util.
1	2	2.64	8.92	0.965
1	4	3.03	9.04	0.901
1	8	5.06	4.32	0.810
2	2	3.01	9.20	0.963
2	4	4.35	7.82	0.794
2	8	8.56	8.62	0.803
4	2	4.88	11.44	0.889
4	4	5.83	8.56	0.854
4	8	9.32	9.29	0.800

although further refinement is required for difficult schema classes.

The comparison with DuckDB and PyArrow on S3 should be interpreted as an external calibration context for absolute size_ratio and throughput values on identical input data. Because these systems differ in their full ingest semantics, the table is not used as a basis for claiming planner superiority. Such conclusions are drawn from the controlled internal comparison.

The comparison with reference tools on S3 shows that the size_ratio and throughput values lie within a comparable range, while the trade-off profiles differ across classes of input data. ParqOpt is evaluated as an end-to-end ingestion service, where the cost includes contract checks, staging and commit phases, and input unification; therefore, the table reflects only the level of the writer path and conversion and does not claim equivalence of execution semantics. This comparison is used as a practical reference point for the writer path and conversion during writes to S3, without comparing ingestion and resume contracts.

C. *Question 3, operational scalability of the split runtime*

The scaling matrix shows that increasing the number of workers reduces completion tails because the available parallelism on the worker plane increases. The growth of submit tails with increasing API replicas and concurrency is consistent with the limitations of the SQLite-based job store. The soak test confirms that, in the SQLite configuration, concurrent-write limitations emerge as the number of concurrent requests rises and require either serialized writes or migration to a multi-user state store; this is an engineering issue that can be addressed by replacing the job store with a more concurrent state-storage system.

IX. CONCLUSION

This paper presented ParqOpt, an ingestion service for accepting heterogeneous files and rewriting them into columnar formats ahead of object storage. The system combines a C++ conversion library core with a split runtime that separates request admission from execution of heavy pipeline tasks on workers, which simplifies operation and scaling.

For RQ1, a formal resume contract for S3 multipart upload was introduced, based on confirmed parts, their identifiers, and the expected offset of the next part. The contract is verified by a deterministic QA suite in positive and negative scenarios, making resume correctness a reproducible and observable property of the system.

For RQ2, we showed that the ingestion outcome is strongly affected by the storage environment, planner settings, and the streaming mode. The local-vs-S3 matrix establishes a consistent difference in end-to-end latency and throughput between the two paths, consistent with the specifics of staging and commit over object storage. The controlled internal comparison shows stable differences in output size under fixed ingest semantics, whereas the effects on throughput and memory depend on the class of input data. For the streaming path, the extended matrix supports the claim of a budget-aware and controlled execution mode, but not of universal reduction in absolute max RSS.

TABLE IX. SOAK TESTING

Case	Requests	Accepted	Rejected	Submit p95 (ms)	Completion p95 (ms)
typical_1MB	300	299	1	9303.02	6900
typical_10MB	180	179	1	8720.73	23539
high_cardinality_10MB	180	179	1	8001.03	40991
typical_100MB	40	39	1	11248.21	34204

For RQ3, we showed that scaling the worker plane reduces completion tails. In the SQLite configuration, a limitation on concurrent writes to the job store appears under load, which defines a practical boundary of the current setup and indicates the need for a more concurrent state store as load increases.

Future work includes extending the set of datasets and target modes for automatic write-parameter selection, conducting a separate evaluation on larger inputs and distributed object storage, optimizing commit cost on object storage, and migrating the job store to a multi-user state-storage system in order to eliminate the concurrency limitation in the control plane.

REFERENCES

[1] M. Armbrust, A. Ghodsi, R. S. Xin, and M. Zaharia, "Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics," 2021. [Online]. Available: https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

[2] A. W. Services, "Amazon s3 multipart upload overview." [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/userguide/mpuoverview.html

[3] "Kubernetes components," Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/

[4] A. Nadeem and M. Z. Malik, "A case for microservices orchestration using workflow engines," in *Proceedings of the 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER 2022)*. ACM, 2022, pp. 6–10. [Online]. Available: https://dl.acm.org/doi/10.1145/3510455.3512777

[5] G. Langdale and D. Lemire, "Parsing gigabytes of json per second," p. 941–960, Oct. 2019. [Online]. Available: http://dx.doi.org/10.1007/s00778-019-00578-5

[6] "Arrow columnar format," Apache Arrow Specification. [Online]. Available: https://arrow.apache.org/docs/format/Columnar.html

[7] "Concepts," Apache Parquet Documentation. [Online]. Available: https://parquet.apache.org/docs/concepts/

[8] "Orc specification v1," Apache ORC Documentation. [Online]. Available: https://orc.apache.org/specification/ORCv1/

[9] "Apache avro specification," Apache Avro Documentation. [Online]. Available: https://avro.apache.org/docs/1.11.1/specification/

[10] "Apache arrow format specification (serialization and ipc)," Apache Arrow Documentation. [Online]. Available: https://arrow.apache.org/docs/format/index.html

[11] F. F. Okumus, A. Ramic, and S. Kugele, "A systematic mapping study on contract-based software design for dependable systems," 2025. [Online]. Available: https://arxiv.org/html/2505.07542v1

[12] "S3a committers: Architecture and implementation," Apache Hadoop Documentation. [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-aws/tools/hadoop-aws/committer_architecture.html

[13] A. W. Services, "Amazon s3 api reference: Listparts." [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/API/API_ListParts.html

[14] NIST/SEMATECH, "Ewma control charts," NIST/SEMATECH e-Handbook of Statistical Methods. [Online]. Available: https://www.itl.nist.gov/div898/handbook/pmc/section3/pmc324.htm

[15] "Hadoop-aws module: Integration with amazon web services," Apache Hadoop Documentation. [Online]. Available: https://hadoop.apache.org/docs/r3.4.1/hadoop-aws/tools/hadoop-aws/index.html

[16] C. Liu, A. Pavlenko, M. Interlandi, and B. Haynes, "A deep dive into common open formats for analytical dbmss," p. 3044–3056, Jul. 2023. [Online]. Available: https://doi.org/10.14778/3611479.3611507

[17] "Data pages: Encodings," Apache Parquet Documentation. [Online]. Available: https://parquet.apache.org/docs/file-format/data-pages/encodings/

[18] "Parquet page index," Apache Parquet Documentation. [Online]. Available: https://parquet.apache.org/docs/file-format/pageindex/

[19] "Configurations: Row group size," Apache Parquet Documentation. [Online]. Available: https://parquet.apache.org/docs/file-format/configurations/

[20] "Tuning workloads: The effect of row groups on parallelism," DuckDB Documentation. [Online]. Available: https://duckdb.org/docs/stable/guides/performance/how_to_tune_workloads.html

[21] "Tpc-ds standard specification, version 4.0.0," Transaction Processing Performance Council. [Online]. Available: https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-DS_v4.0.0.pdf

[22] "Tpc benchmark h standard specification, revision 3.0.1," Transaction Processing Performance Council. [Online]. Available: https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.1.pdf

[23] "Convert input data format in amazon data firehose (record format conversion)," Amazon Web Services. [Online]. Available: https://docs.aws.amazon.com/firehose/latest/dev/record-format-conversion.html

[24] "Dataformatconversionconfiguration (amazon data firehose api reference)," Amazon Web Services. [Online]. Available: https://docs.aws.amazon.com/firehose/latest/APIReference/API_DataFormatConversionConfiguration.html

[25] "Using the parquet format in aws glue," Amazon Web Services. [Online]. Available: https://docs.aws.amazon.com/glue/latest/dg/aws-glue-programming-etl-format-parquet-home.html

[26] "Streaming etl jobs in aws glue," Amazon Web Services. [Online]. Available: https://docs.aws.amazon.com/glue/latest/dg/add-job-streaming.html

[27] "Introduction to loading data (bigquery)," Google Cloud. [Online]. Available: https://docs.cloud.google.com/bigquery/docs/loading-data

[28] "Introduction to the bigquery storage write api," Google Cloud. [Online]. Available: https://docs.cloud.google.com/bigquery/docs/write-api

[29] "Ingestion mappings (kusto/azure data explorer)," Microsoft. [Online]. Available: https://learn.microsoft.com/en-us/kusto/management/mappings?view=microsoft-fabric

[30] "Supported ingestion formats in azure data explorer," Microsoft. [Online]. Available: https://learn.microsoft.com/en-us/azure/data-explorer/ingestion-supported-formats

[31] "Load data from (almost) any source into parquet files," Rivery. [Online]. Available: https://rivery.io/blog/load-data-parquet-files/

[32] "Etl s3 (amazon s3 integration)," Rivery. [Online]. Available: https://rivery.io/integration/s3/

[33] "Oneflow data ingestion," Onehouse. [Online]. Available: https://www.onehouse.ai/product/oneflow

[34] "Kafka connect user guide," Apache Software Foundation. [Online]. Available: https://kafka.apache.org/41/kafka-connect/user-guide/

[35] "Apache nifi user guide (back pressure)," Apache Software Foundation. [Online]. Available: https://nifi.apache.org/docs/nifi-docs/html/user-guide.html

[36] "S3a committers: Architecture and implementation," Apache Software Foundation. [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-aws/tools/hadoop-aws/committer_architecture.html

[37] "Convert input data format in amazon data firehose," Amazon Data Firehose Developer Guide. [Online]. Available: https://docs.aws.amazon.com/firehose/latest/dev/record-format-conversion.html

[38] "Aws glue spark and pyspark jobs," AWS Glue Developer Guide. [Online]. Available: https://docs.aws.amazon.com/glue/latest/dg/spark_and_pyspark.html
[39] M. Saxena *et al.*, "The story of aws glue," 2023. [Online]. Available: https://www.vldb.org/pvldb/vol16/p3557-saxena.pdf
[40] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: A unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016. [Online]. Available: https://dl.acm.org/doi/10.1145/2934664
[41] "Connector development guide," Apache Kafka Documentation. [Online]. Available: https://kafka.apache.org/41/kafka-connect/connector-development-guide/
[42] "Amazon s3 sink connector for confluent platform," Confluent Documentation. [Online]. Available: https://docs.confluent.io/kafka-connectors/s3-sink/current/overview.html
[43] "Apache nifi user guide," Apache NiFi Documentation. [Online]. Available: https://nifi.apache.org/docs/nifi-docs/html/user-guide.html
[44] "Spark procedures: rewrite_data_files," Apache Iceberg Documentation. [Online]. Available: https://apache.github.io/iceberg/docs/1.4.3/spark-procedures/
[45] "Optimizations (oss): Optimize and compaction," Delta Lake Documentation. [Online]. Available: https://docs.delta.io/optimizations-oss/
[46] "Compaction," Apache Hudi Documentation. [Online]. Available: https://hudi.apache.org/docs/1.0.1/compaction/
[47] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems (EuroSys 2015), Bordeaux, France, April 21–24, 2015.* ACM, 2015, pp. 18:1–18:17. [Online]. Available: https://dl.acm.org/doi/10.1145/2741948.2741964
[48] "pipe(7) - linux manual page," Linux man-pages project. [Online]. Available: https://man7.org/linux/man-pages/man7/pipe.7.html
[49] "Reactive streams specification for the jvm," GitHub repository. [Online]. Available: https://github.com/reactive-streams/reactive-streams-jvm
[50] "Amazon s3 api reference: Checksum," Amazon Web Services
[62] "getrusage(2) - linux manual page," Linux man-pages project. [Online]. Available: https://man7.org/linux/man-pages/man2/getrusage.2.html
Documentation. [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/API/API_Checksum.html
[51] "Check the integrity of data in amazon s3 with additional checksums," Amazon Web Services Hands-on Guides. [Online]. Available: https://docs.aws.amazon.com/hands-on/latest/amazon-s3-with-additional-checksums/amazon-s3-with-additional-checksums.html
[52] A. W. Services, "Amazon s3 api reference: Completemultipartupload." [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/API/API_CompleteMultipartUpload.html
[53] "Service level objectives," Site Reliability Engineering (Google SRE Book). [Online]. Available: https://sre.google/sre-book/service-level-objectives/
[54] J. L. J. Pereira, G. A. Oliver, M. B. Francisco *et al.*, "A review of multi-objective optimization: Methods and algorithms in mechanical engineering problems," 2021. [Online]. Available: https://link.springer.com/article/10.1007/s11831-021-09663-x
[55] "Indexes," Apache ORC Documentation. [Online]. Available: https://orc.apache.org/docs/indexes.html
[56] A. S. Foundation, "Apache orc specification." [Online]. Available: https://orc.apache.org/specification/
[57] "Arrays: arrow::arraydata," Apache Arrow C++ API Documentation. [Online]. Available: https://arrow.apache.org/docs/6.0/cpp/api/array.html
[58] "Blpop," Redis Command Documentation. [Online]. Available: https://redis.io/docs/latest/commands/blpop/
[59] "Atomic commit in sqlite," SQLite Documentation. [Online]. Available: https://sqlite.org/atomiccommit.html
[60] I. MinIO, "Minio documentation." [Online]. Available: https://min.io/docs/
[61] "Certified tester foundation level syllabus - performance testing (ct-pt)," International Software Testing Qualifications Board (ISTQB), 2018. [Online]. Available: https://istqb.org/wp-content/uploads/2024/11/ISTQB-CT-PT_Syllabus_v1.0_2018.pdf
[63] "The /proc filesystem, /proc/pid/status fields, table 1-2," Linux Kernel Documentation. [Online]. Available: https://docs.kernel.org/filesystems/proc.html#id10
[64] R. Fielding, M. Nottingham, and J. Reschke, "Rfc 9110: Http semantics. section 9.2.2. idempotent methods." 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9110.html#name-idempotent-methods