The Story of AWS Glue

Mohit Saxena*, Benjamin Sowell†, Daiyan Alamgir, Nitin Bahadur, Bijay Bisht, Santosh Chandrachood, Chitti Keswani, G2 Krishnamoorthy, Austin Lee, Bohou Li, Zach Mitchell, Vaibhav Porwal, Maheedhar Reddy Chappidi, Brian Ross, Noritaka Sekiyama, Omer Zaki, Linchi Zhang, Mehul A. Shah†
Amazon Web Services
glue-paper@amazon.com

ABSTRACT

AWS Glue is Amazon's serverless data integration cloud service that makes it simple and cost effective to extract, clean, enrich, load, and organize data. Originally launched in August 2017, AWS Glue began as an extract-transform-load (ETL) service designed to relieve developers and data engineers of the undifferentiated heavy lifting needed to load databases, data warehouses, and build data lakes on Amazon S3. Since then, it has evolved to serve a larger audience including ETL specialists and data scientists, and includes a broader suite of data integration capabilities. Today, hundreds of thousands of customers use AWS Glue every month.

In this paper, we describe the use cases and challenges cloud customers face in preparing data for analytics and the tenets we chose to drive Glue's design. We chose early on to focus on ease-of-use, scale, and extensibility. At its core, Glue offers serverless Apache Spark and Python engines backed by a purpose-built resource manager for fast startup and auto-scaling. In Spark, it offers a new data structure — DynamicFrames — for manipulating messy schema-free semi-structured data such as event logs, a variety of transformations and tooling to simplify data preparation, and a new shuffle plugin to offload to cloud storage. It also includes a Hive-metastore compatible Data Catalog with Glue crawlers to build and manage metadata, e.g. for data lakes on Amazon S3. Finally, Glue Studio is its visual interface for authoring Spark and Python-based ETL jobs. We describe the innovations that differentiate AWS Glue and drive its popularity and how it has evolved over the years.

PVLDB Reference Format:
Mohit Saxena, Benjamin Sowell, Daiyan Alamgir, Nitin Bahadur, Bijay Bisht, Santosh Chandrachood, Chitti Keswani, G2 Krishnamoorthy, Austin Lee, Bohou Li, Zach Mitchell, Vaibhav Porwal, Maheedhar Reddy Chappidi, Brian Ross, Noritaka Sekiyama, Omer Zaki, Linchi Zhang, Mehul A. Shah. The Story of AWS Glue. PVLDB, 16(12): 3557 - 3569, 2023.
doi:10.14778/3611540.3611547

1 INTRODUCTION

When we started AWS Glue (circa 2016), databases and analytics on the cloud were new and rapidly growing businesses, and AWS had recently released a variety of modern data services. Customers were using systems like Amazon RDS and Aurora for their operational data, Amazon DynamoDB for NoSQL data, Amazon Kinesis for streaming data, and Amazon Redshift for data warehousing. We also saw customers collect and store large amounts of semi-structured data in large object stores like Amazon S3 because of its scalability, durability, and low cost. These datasets range from gigabytes to petabytes and examples include distributed system logs, mobile event streams, clickstreams, adtech data, social feeds, and IoT streams. Customers increasingly built data lakes and turned to big data services like Amazon EMR or Amazon Athena to query the data in place.

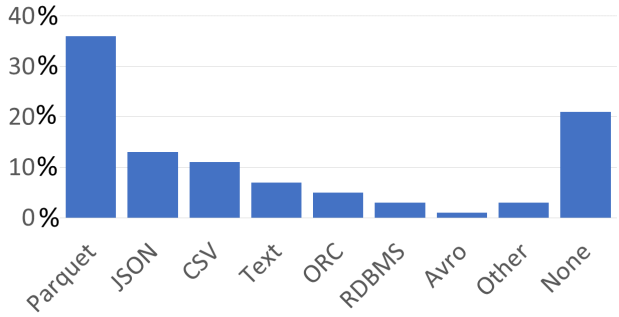


Figure 1: Table data types in the AWS Glue Data Catalog

Facing an explosion in the variety of data and data use-cases, as well as in the types of systems they could use to query that data, customers told us that they lacked tools that made it easy to discover, prepare, and move their data between cloud services. Existing extract-transform-load (ETL) tools, for example, were designed for an earlier era to move data from structured databases into data warehouses, and were difficult to scale. Customer expectations in the cloud were also different. They wanted services rather than software and expected their tools to be elastic and scalable by default. Instead of using those tools, we saw customers write their own ETL scripts and manage their own infrastructure, which was both tedious and brittle. To address this problem, we set out to build AWS Glue with a cloud-oriented perspective on data movement and data preparation, and launched it in August 2017.

An important premise that differentiated and drove our approach with AWS Glue is that ETL is a long tail problem. There is no 80-20 rule for formats, sources, or targets, and our experience shows that

*Corresponding Author
†Work done while at Amazon Web Services

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611547

each use case requires some customization (Section 2). As an example, consider the data types found in the AWS Glue Data Catalog, a metadata store that customers use for organizing and querying their data lake tables and other enterprise datasets. Figure 1 shows a high-level breakdown of the common types as of September 2022 in one of the major AWS regions. While Apache Parquet is the most common, there are a sizable percentage of text-based formats like JSON and CSV, each of which have innumerable variations as well as relational tables and a long tail of other formats. While popular formats have changed over time, this distribution remains long tailed and is a reasonable proxy for the data types stored on S3.

Given the complexity of ETL, we also made the choice to initially target cloud developers and data engineers. They were often tasked with transforming this variety of data into optimized formats like Apache Parquet for data lakes or loading cloud data warehouses. These developers were comfortable writing code, but wanted general purpose tools to help prepare and transform large datasets.

With these factors in mind, we adopted a set of principles for developing AWS Glue. The first is to provide customers with *self-service escape valves* to let them solve their own problems when the system falls short. For example, we made it easy for customers to write code to customize their ETL pipelines or directly edit service-generated scripts for their use cases. The second principle is to be *descriptive, not prescriptive* [21]. The analytics environment is increasingly heterogeneous, and it is not always possible to define a single data model, type system, or query language to support all use cases. Rather than requiring customers to standardize their data before getting started, we let them bring what they have, even if it cannot be used everywhere, so that they can incrementally adopt Glue for new applications and use cases. The final principle is to *minimize undifferentiated work*. While our customers are developers, they did not want to spend time managing infrastructure.

We set about building the AWS Glue service with these principles in mind; its architecture is shown in Figure 2. For data transformation, the AWS Glue ETL stack includes several key components (Section 3). First, *Glue Studio* is a visual interface for job authoring that automatically generates human readable Apache Spark scripts. We learned that even developers appreciate low-code tools that help them get started quickly, and this interface eventually helped us broaden our reach to ETL specialists and data scientists. Second, the AWS Glue ETL runtime includes the core Spark packages and Glue-specific libraries. We chose to build on Apache Spark because it was a general purpose tool that developers were familiar with, and we extended it with libraries designed to make ETL jobs more efficient and resilient. These libraries introduce a new data structure, the *DynamicFrame*, and new transformations that are specifically designed to help prepare and clean deeply nested, semi-structured data. These libraries have been publicly available since 2019 and used by thousands of customers daily to develop their ETL scripts [11]. Third, Glue offers an orchestration system to stitch together multiple jobs into a pipeline and new features to simplify incremental processing.

Glue also offers a serverless interface for running Apache Spark jobs (Section 3.3). Users submit Spark jobs for execution, and the service does the rest. Glue was one of the earliest serverless analytics services in production. Over the years, we have seen customers build on and use this serverless interface in ways we did not expect. For example, customers use much smaller batch sizes when they do not have to worry about keeping clusters highly utilized. We were surprised to find that the median Glue Spark job runtime has dropped steadily and is now less than a few minutes. Also, while originally aimed for batch ETL, users routinely used Glue for interactive debugging and experimentation. This motivated us to push the boundaries of our serverless compute backend to provide faster start times and dynamic auto scaling. Today, Glue jobs often start within a few seconds allowing for interactive execution. Glue also dynamically scales resources up and down during job execution to lower cost and provide better availability.

Finally, we learned that customers need more than just data transformation to effectively manage their ETL pipelines. Since much of their data was stored in Amazon S3, often without a clear schema, they needed ways to collect and store metadata. The *AWS Glue Data Catalog* provides a scalable metadata service (Section 4). The catalog lets customers model datasets as databases and tables, where tables can refer to data in a variety of stores such as Amazon S3, relational databases, NoSQL stores, and streaming data services. AWS services such as Amazon Athena, Amazon EMR, and Amazon Redshift can natively query tables defined in the data catalog, and users can leverage an Apache Hive Metastore-compatible adapter for use with open source engines like Apache Spark and Presto. Also, to facilitate data discovery and create table definitions automatically, *AWS Glue crawlers* scan objects in Amazon S3 to infer file formats, schemas, table boundaries, and partitions (Section 5) as well as collect metadata from databases and other data services.

AWS Glue has been in production for six years, and during that time we have seen the ETL and data analytics environment evolve considerably. As data lakes have become more popular and customers have adopted an increasingly broad set of tools, we have learned several lessons about building and operating a modern data integration service. In the rest of this paper, we present the architecture of AWS Glue, how it has evolved to meet the changing needs of our customers, and the lessons we learned.

2 USE CASES AND CHALLENGES

Customers use AWS Glue for a wide variety of ETL and data integration applications. In this section, we present a few representative use cases and introduce some of the technical challenges that our customers face that helped motivate the features and architectural decisions described in the remainder of this paper.

2.1 Use Cases

Over the past six years that we have operated AWS Glue, we have seen customers' ETL and data integration use-cases evolve significantly. When we first started, data lakes were still nascent, and many customers were just starting to experiment with analytical queries on datasets outside of traditional data warehouses. As data lakes have matured, customers have developed more and more sophisticated pipelines to process their data.

Figure 3 shows some examples of some common use cases we have seen while operating AWS Glue. Figure 3a shows one early use case we encountered where customers were trying to load semi-structured data from Amazon S3 into Amazon Redshift to take advantage of its query performance and broad set of tools.

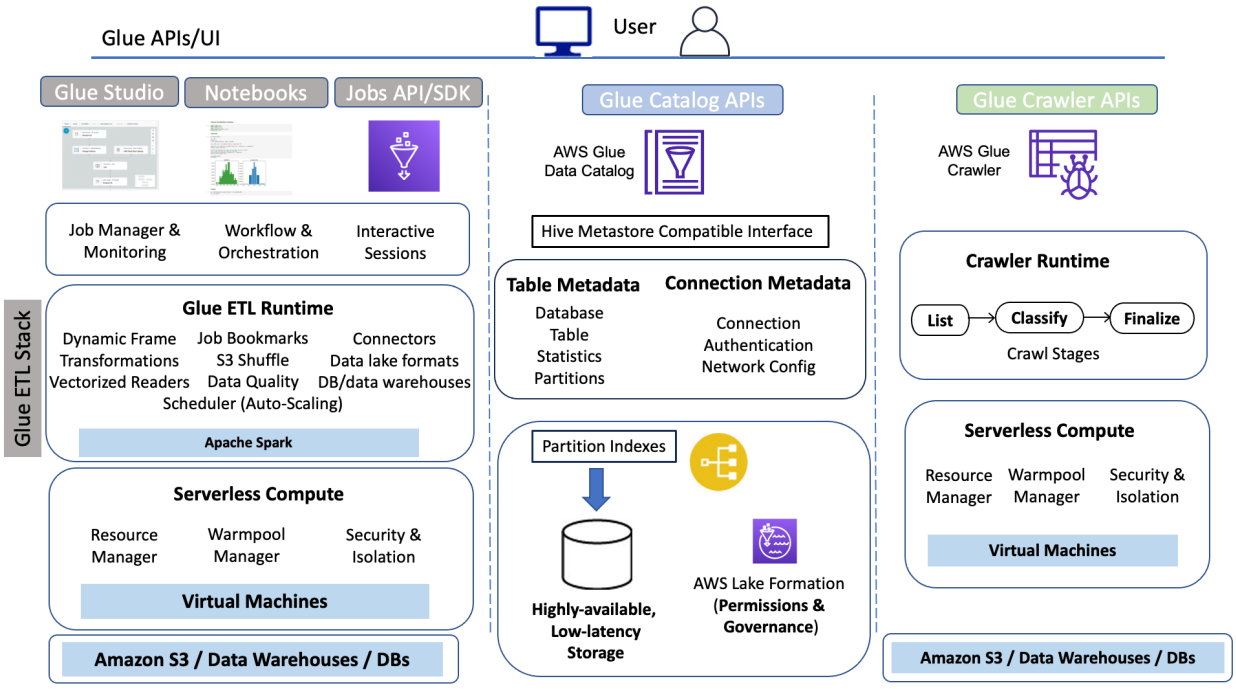
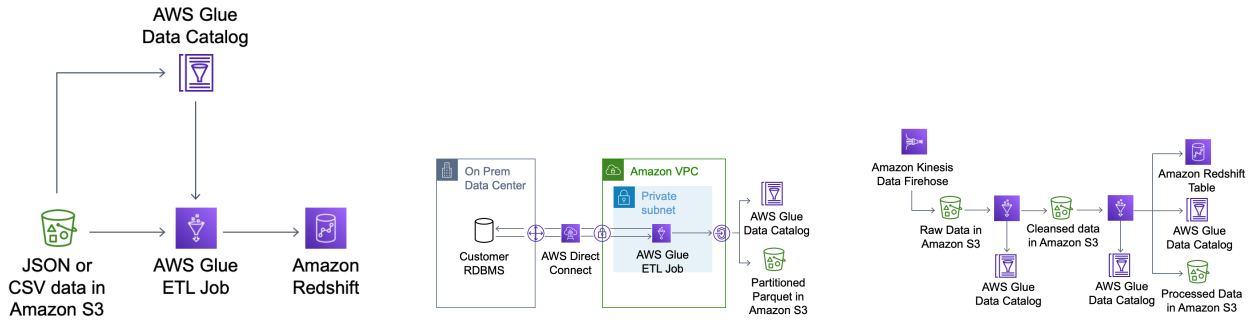


Figure 2: AWS Glue Architecture



(a) Loading a data warehouse **(b) Connecting to an on-prem RDBMS** **(c) Ingesting streaming data**

Figure 3: Sample AWS Glue use cases

Here, Glue has to discover the schema of the source data, populate the data catalog, perform any required transformations, such as unnesting data, and load the data into Redshift. Figure 3b shows a slightly more complex example where a customer copies data from an on-premise database into Amazon S3 in a format like Parquet. Query engines like Amazon Athena can leverage such formats to speed up analytics. Finally, Figure 3c shows a more complex streaming ingestion pipeline. Here the customer puts raw data from a streaming source like Amazon Kinesis into S3, and then performs several phases of transformation to clean and pre-aggregate the data, and ultimately load it into multiple destinations, such as Amazon S3 and Amazon Redshift.

2.2 Challenges

While helping customers with the use-cases described above and others, we encountered a number of technical challenges. We loosely group them based on the stage in the ETL pipeline where customers most often encounter them.

2.2.1 Data Discovery and Ingest. As customers seek to analyze increasingly diverse data, they have to deal with *missing or inconsistent metadata*. Many datasets, particularly semi-structured datasets like logs, often lack associated metadata. For example, JSON data typically does not come with a schema, and it is common for two files in the same dataset to contain slightly different sets of attributes. Even when a customer uses a tool like AWS Glue crawlers to discover the schema, there may still be inconsistencies.

For example, a single field may appear with different types in different records, which makes the dataset hard to query in traditional analytics tools. With Glue, we want to provide mechanisms for customers to query and transform these datasets, and give them tools to fix inconsistencies over time.

Customers also face challenges *integrating with external systems*. Customers want to use ETL to process data from a wide variety of external systems, including object stores like Amazon S3, NoSQL stores like Amazon DynamoDB, and a variety of relational databases, from those managed in AWS with Amazon RDS to those run on-premise. Accessing these systems requires traversing a variety of network isolation mechanisms, from virtual private clouds (VPCs) and subnets, to separate physical networks for on-premise resources, as well as different authentication mechanisms, such as Amazon's IAM service for AWS-based resources and passwords or secrets for accessing relational databases via protocols like JDBC. Customers need help configuring these options and reusing them across multiple ETL jobs.

Scalability can also be a challenge when working with external systems. Systems like Apache Spark are designed to scale horizontally to process massive amounts of data in parallel, and this can easily overwhelm source systems with different scaling properties. This is particularly important for relational sources, which may run on a single host, but even services like Amazon DynamoDB and Amazon S3 limit the throughput with which clients can access data under a hot partition or single prefix respectively. Customers need mechanisms to throttle their ETL jobs and retry on failure to avoid browning out their source systems.

2.2.2 Reliable Data Processing. Operations like crawling and ETL are often performed on a schedule without manual intervention, so *reliability and hands-off maintenance* are critical. While customers care deeply about performance, reliability is essential, and they will often choose a system with good predictable performance over one that is faster on individual queries but less predictable overall.

Reliability is particularly important given *scale and workload variability over time*. While many ETL jobs are run incrementally on a schedule, in practice data sizes vary for many reasons. For example, customers often need to perform backfill operations where they process years of data at once, and they often develop and test their ETL processes with datasets that are significantly smaller than what they see in production. This can lead to scaling cliffs where performance degrades or jobs fail due to exhausting resources like memory or local disk.

2.2.3 Output and Physical Layout. Since customers directly manage the files that make up a data lake in an object store like Amazon S3, they need tools to help manage *physical layout and data partitioning*. Many big data systems, including Apache Hive, Apache Spark, and Trino, support dataset partitioning, where certain attributes are encoded in the file path. For example, records in data files under the prefix `/year=2022/month=05/day=01` would implicitly have `year`, `month`, and `day` attributes with the corresponding values. Selecting a good partitioning scheme is critical for query performance, as query engines use partition values to prune the set of files they read from the underlying object store. Metadata about each partition is stored in a metadata store like the AWS Glue Data Catalog, and customers need help keeping this metadata up-to-date as new partitions arrive, as well as re-partitioning data to improve query performance.

File size is also critical for query performance in data lakes. The latency of making a request to an object store means that it is essential to store data in larger objects to amortize this per-request overhead. This often creates a tension with streaming ingestion processes, which write small files to reduce the time it takes to make new data available for querying. When customers use AWS Glue to process raw data, we have seen datasets ranging from a single un-splittable 100GB gzip file to millions of files containing a single record each. Customers want to use AWS Glue to help re-organize these datasets to accelerate their query workloads.

3 GLUE ETL STACK

In this section, we describe Glue's ETL stack (see Figure 2), which comprises a user interface, a Glue ETL runtime, a serverless compute backend, a job orchestration system, and key capabilities for making ETL jobs more reliable and efficient.

While working with customers, we learned that ETL was a long-tail business – in most implementations, developers relied mostly on the pre-packaged capabilities, but inevitably needed some custom logic in their implementations to complete their task. Our approach, therefore, was to automate as much of the undifferentiated work as possible, while giving developers a general-purpose programming environment in which they could easily build their solutions. At the time (circa 2015-2016), Apache Spark was an increasingly popular big data engine often used for ETL use cases, and we chose it because of its flexibility and familiarity among developers.

While user interfaces for ETL are not new, our approach was to build one that generated human-readable code which customers could then directly edit and customize. The user interface used the data catalog as a convenient starting point because sources, schemas, and targets were often pre-populated, and the system could focus on ostensibly straightforward transformations to go from source to target. Data transformation, however, is usually not that simple because there are often unexpected variations in the content, structure, and scale of the data.

To make transformations faster and more robust, we added Glue ETL libraries in the runtime to handle the unexpected changes and innumerable variations which are the characteristic of ETL workloads. At the core of these libraries is a new base data structure, a DynamicFrame, which is a collection of self-describing records, DynamicRecords. As a result, DynamicFrames are more efficient for single-pass data transformation jobs that we often saw in ETL workloads. They do not require a schema upfront and can efficiently represent data sets like the GitHub Timeline [20] which contain widely varying record types from a few to hundreds of columns. The Glue ETL libraries also include new transformation operations for flattening and materializing DynamicFrames on-the-fly into column-oriented formats like Apache Parquet in a streaming fashion. Separately, we built vectorized readers for DynamicFrames in a native language (C++) which use hardware parallelism and columnar in-memory formats to speed up processing of raw formats like CSV and JSON.

Configuring and managing infrastructure was another major pain point for developers, so Glue sought to eliminate that altogether through a serverless interface for Apache Spark jobs. Glue's

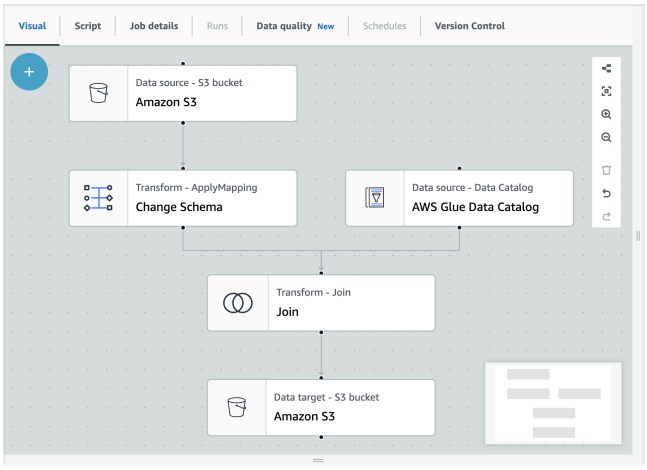


Figure 4: An example job created with AWS Glue Studio

serverless compute backend efficiently and securely provisions clusters for running those jobs. It includes a warmpool manager that maintains pre-provisioned EC2 virtual machines (VMs), a purpose-built Spark scheduler that integrates with the serverless compute to allocate resources to jobs, and mechanisms for security, networking, and isolation. To make serverless compute more efficient, we solved a few important challenges that we think will also translate to other data parallel systems. First, we decoupled Apache Spark executors from depending on local storage, e.g. for intermediate shuffles. Second, we improved job startup time from longer than 8 minutes on average down to a few seconds. Third, the decoupling and fast startup paved the way for a scheduler that dynamically scales resources up and down *(auto scales)* during the execution of individual job runs to achieve better performance, availability, and cost-efficiency for customers.

Glue also includes an orchestration system, Glue Workflows, for composing multiple ETL jobs and running them reliably. It allows stitching together of multiple jobs, automatic scheduling, and convenient features for incremental processing. This includes job bookmarks, which help jobs pickup from where they left off in the previous run. So, developers need not worry about maintaining execution state between job runs. Finally, it provides "bounded execution", a feature that allow Glue Spark jobs to gracefully degrade when the input size overflows the amount of metadata that Spark can handle in a single machine.

3.1 User Interface

To make it easier for users to get started with AWS Glue, we built a visual ETL interface and code generation mechanism for ETL scripts. This capability has gone through several iterations over the years, but all of them rely on an intermediate representation of an ETL script as a DAG (directed acyclic graph). This DAG is similar to a simple query plan where nodes correspond to data sources or transformations, which can be simple relational operations like filter or join or ETL-specific operations like flattening nested fields.

Figure 4 shows an example of an ETL job created using the latest *Glue Studio* UI [7], which allows customers to create ETL scripts visually. This example takes one data source from Amazon S3 and

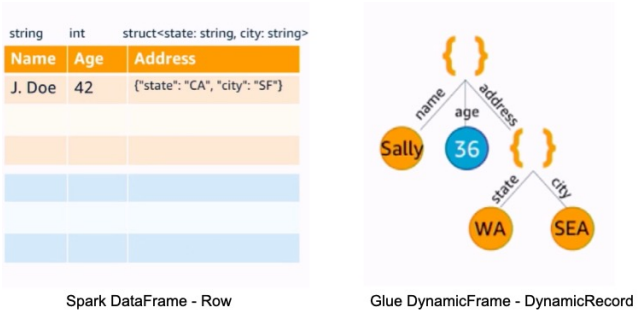


Figure 5: Spark DataFrame with Rows and Glue DynamicFrame with DynamicRecord

performs an *ApplyMapping*, which is used to restructure or flatten nested objects. Then, it joins the result with a table from the Data Catalog and writes the output to Amazon S3. Users can customize each node in the DAG, and when they are ready they can generate code. They can also choose to edit the script directly to add logic that may not be easy to embed in a DAG.

3.2 Glue ETL Library with DynamicFrames

When we started developing Glue, one of the challenges we saw customers face over and over again was dealing with the realities of messy, semi-structured data. At the time, many customers were just starting to use Spark.

Most popular big-data query engines, including Spark, require a schema before they can be used to query or transform datasets. For example, Spark relies on Dataframes, which are organized as a collection of nested arrays (Rows). Spark handles this process by performing schema inference during the ingestion process. This works well for formats such as Avro and Apache Parquet, where the schema can be extracted without reading the complete dataset, but it a requires a full scan of the data for formats like JSON that do not have a declared schema. This additional pass can add significant execution time for large datasets. In some cases it may not be possible to infer a schema, as some fields may appear with multiple types, either due to corruption or legitimate schema changes.

To make it easier for our customers to work with this kind of messy data, we built Glue's ETL runtime library on top of a new data-structure called the *DynamicFrame*. Rather than requiring a schema up-front, DynamicFrames embed schema information in each record and compute a global schema only when required. This means that many ETL jobs can avoid the cost of computing a schema all together, and those that do need a schema compute it as late as possible.

Internally, DynamicFrames are stored as Spark RDDs of *DynamicRecords*, which are tree-based data structures containing both column information and data values. DynamicRecords are self-describing and support all of the standard data types found in Spark, including complex types such structs, maps, and arrays. The Glue ETL libraries include readers to create DynamicFrames from many common file formats, including Avro, CSV, JSON, ORC, and Apache Parquet, and data sources, including relational databases over JDBC and common NoSQL stores like Amazon DynamoDB and MongoDB.

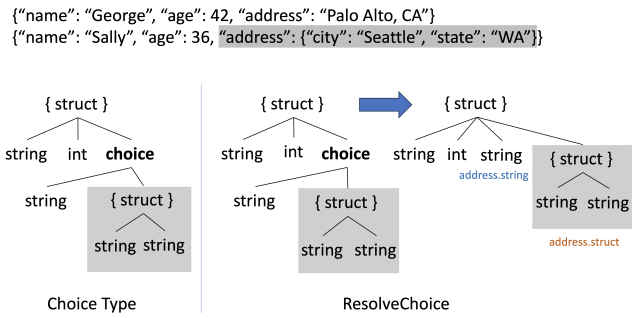


Figure 6: Choice types and ResolveChoice transform in Glue

DynamicFrames support a number of standard transformations such as selection and projection, and have support for UDFs in Python and Scala for filtering and transforming individual records. These operations can be performed record-at-a-time without ever computing a local schema. DynamicFrames also include a number of transformations specially designed for working with deeply nested data. For example, *relationalize* is a transformation that is designed to make it possible to prepare data for ingestion into a relational database without requiring any additional input from the user. It does two things. First, it flattens nested structs so that all fields are at the top-level, and second, it pivots arrays and extracts them into a separate table. This is applied recursively to support data with arbitrarily nested structs and arrays.

While Glue DynamicFrames provide a number of capabilities for ETL, they're not as full-featured as Spark Datasets, particularly for analytics-type operations such as joins and complex aggregations. For those types of operations, users can convert DynamicFrames to Datasets simply by using the `toDF` method. This conversion does require a schema and may trigger an additional pass over the data, but we do see a common pattern of customers using DynamicFrames to read and filter data, taking advantage of their flexibility, and then converting them to Dataframes for further processing. Both data structures are supported in Glue.

3.2.1 Schema Inference. While many ETL transformations can be performed without requiring a global schema, they are sometimes required. For example, one transformation in Glue drops all fields in which every value is null. This is useful because a more specific type cannot be inferred and many ETL targets don't support fields with a null type.

Our schema inference algorithm is similar to that used by Spark, except that it is designed to return a valid schema for any possible set of records, even when there is a schema conflict. We inspect every record and union the structures – field name and types – that we encounter. To keep the schema concise, we also union the schemas found within nested arrays. Unlike Spark, however, we track nulls, absence of values (null-type), as well as schema conflicts that cannot be easily resolved. We accomplish this by introducing a union type, which we call a *ChoiceType*, that records every possible type taken by a specific field in a DynamicFrame. For example, Figure 6 shows an example of part of a schema that might be inferred by Glue. We also use ChoiceTypes to represent the type of arrays with heterogeneous elements. The addition of ChoiceTypes, along with a few other extensions such a NullType to represent the absence of a value, allow us to compactly represent any collection of records. For more details, see [31].

ChoiceTypes give users the ability to incrementally transform and improve data even when a simple schema cannot be inferred. To facilitate cleaning, Glue provides the *ResolveChoice* transformation, which allows users to specify a policy to indicate what should happen when a ChoiceType is encountered. Options include casting the data to a common type, keeping only a single one of the variants, or retaining both as part of a struct field or as top-level columns. Finally, the *match catalog* option allows the user to specify a Data Catalog table and resolves ChoiceTypes by attempting to cast to the type of the corresponding field in the Catalog.

The ResolveChoice transformation can be applied either to specific fields or as a default to any ChoiceType in the DynamicFrame. For example, you can specify that all ChoiceTypes should be resolved by making each choice a top-level column with the type in the column name. This works well in the case where users are unfamiliar with their data, but it does require an extra pass over the data to infer the schema and determine which fields are ChoiceTypes. If a certain field is known to be a ChoiceType, it can be resolved directly. This does not require the schema to be computed, as the resolution operation is applied directly to each record.

3.2.2 Glue Parquet Writer. When we launched Glue, one particularly common use case was converting raw data into Parquet format [4]. Parquet is a binary columnar format that is well supported across the different analytics engines and can significantly accelerate queries compared to formats like JSON. Many customers use Glue to perform basic data cleaning operations and then write the data into partitioned Parquet for further analysis in a system like Athena or Redshift Spectrum.

As we worked with customers on this use case, we realized that they were not getting the benefit of using DynamicFrames because the Parquet writer requires a schema up-front. Parquet files are organized as a sequence of *row groups*, each of which stores a subset of the rows in the file in a columnar fashion. Metadata, including the schema, is stored in the footer of the file. Nested data is serialized using a layout scheme first proposed in the Dremel system [23]. Values have an associated *definition level*, which is used to track the presence of optional fields, and *repetition level*, which is used to identify instances of repeated fields. The presence of these fields depends on the schema. If a field is required it does not have a definition level, and if it cannot be repeated it does not have a repetition level.

In 2019, we introduced the Glue Parquet writer to eliminate the need for a schema up-front. The Glue Parquet writer incrementally builds the first row group in memory before setting the schema or flushing anything to disk. Every time a new field is discovered, the writer instantiates a new column and sets the definition and repetition levels appropriately. Once the amount of data stored in memory exceeds a configurable limit, 128 MB by default, the first row group is flushed and the schema for the file is fixed. Subsequent row groups and the file footer are written as usual.

If we find a new field after having written out the first row group, we flush the existing file and start a new file with the larger schema. In the worst case, where every record has a different schema, the

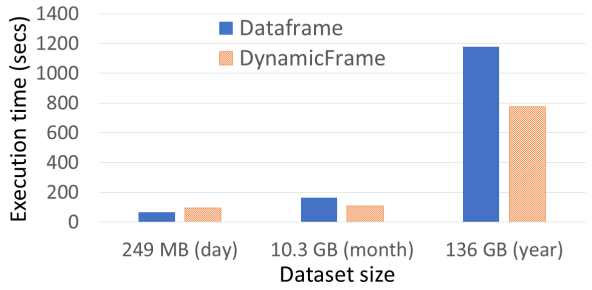


Figure 7: Glue DynamicFrames versus Spark Dataframes for filtering and converting the GitHub timeline (lower is better).

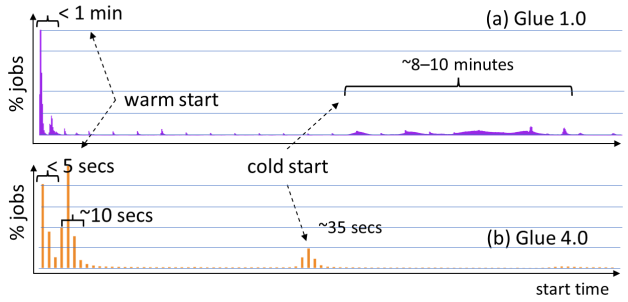


Figure 8: Job start time distribution in Glue 1.0 and Glue 4.0. Note, the startup time axes in (a) and (b) are scaled differently.

writer would create a new file for every record, but in practice we have not found this to be a problem. The first 128 MB typically contain the majority of fields, and the impact to the average file size is minimal. Some systems consuming the resulting Parquet files do require different options when reading collections of files that have different schemas. For example, Spark requires that the `mergeSchema` parameter be set to true so that it reads the schema from all of the files instead of just one.

3.2.3 Experiment. Figure 7 compares the performance of Glue ETL libraries and Apache Spark 3.3.0 processing the GitHub timeline for various durations [12]. The input is a collection of gzipped JSON files that record public GitHub API accesses for a day (22nd, 249MB, 48 files), month (January, 10.3GB, 1488 files), and year (2017, 136GB, 8699 files). It contains over 30 event types whose aggregate schema has 751 distinct attributes. The experiment measures the execution time for selecting only ForkEvents (2.9% selectivity), projecting their payload, and writing the output to Parquet. We run the job in the Glue 4.0 environment using 10xG.1x instances each having 4 vCPU, 16GB of RAM, and 64GB of SSD. The DynamicFrame runs use the Glue Parquet writer, so the entire job requires only one pass over the data, while Dataframe runs use the default Parquet writer, and we configure Spark to compute the entire schema to avoid missing attributes. For small sizes, DynamicFrames are competitive with Dataframes, and as input sizes grow, DynamicFrames are 1.5x faster because they avoid the upfront pass for computing the schema. DynamicFrames have small memory overhead; the maximum memory usage is 8.9% for DynamicFrames versus 7.3% for Dataframes during the run. DynamicFrame output is also more succinct including only the 90 columns of ForkEvents, instead of Dataframes which include all 751 columns, most of which are null. This experiment shows DynamicFrames are better suited for common single-pass filtering, transformation, and conversion jobs, which are often the first step of data preparation.

3.3 Serverless, Fast Startup, and Auto-scaling

From its start, Glue has offered a serverless interface to eliminate the undifferentiated work of managing infrastructure. Glue jobs consist of a script and context metadata such as access control parameters required for execution. Users submit scripts via the Glue Jobs API [9] and the system does the rest. Glue relieves customers from configuring security, networking, software and its dependencies. It uses VM-level isolation across jobs, which provides stronger protection than containers, and isolates resources at the network level with various auditable defense in depth strategies. It also segregates and protects data in transit and at rest. In addition, Glue relieves customers from allocating, tuning, and scaling clusters. Because of the variability in data preparation workloads, customers often either over-provision leading to increased cost, or under-provision risking performance and stability. So, we set out to offer efficient resource allocation and scaling on a per-job level.

Our initial approach in Glue 1.0 was cluster-based and intended for mostly batch workloads. On job start, we choose from three options: (a) run the job on a previously allocated cluster for the user, (b) allocate from a service-wide warmpool of "T-shirt"-sized clusters, (c) provision a new cluster from EC2. Jobs only start after the entire cluster is allocated. The scheduler retires allocated clusters after a fixed idle period to reduce costs, and uses rule-based heuristics to provision more capacity in the warmpool to fill demand. Figure 8(a) shows the distribution of job start times for Glue 1.0 [13]. When clusters are already provisioned (warm start - (a) and (b)), we see start latencies less than one minute. When a new cluster needs to be provisioned, however, the latencies jump to 8-10 minutes and are highly variable. In these cold start cases, clusters are larger and must wait until the last machine is provisioned before starting.

Since serverless makes running jobs much easier and customers do not pay for slow start or idle resources between jobs, we saw customer behavior shift. They built more micro-batch pipelines and interactive applications on Glue, which resulted in the median job runtime dropping below a few minutes. Customers found these start times painful, especially when job runtimes were short.

To speedup start times, we introduced a new resource manager and lighter weight Spark application stack with Glue 2.0 in 2020. Glue 2.0 schedules a job on a dynamically-sized cluster, and the job starts as soon as the first instance is ready. We modified Spark's scheduler [27] to run executors on workers from our resource manager, instead of a cluster-based one like YARN. When needed, our resource manager allocates workers from (a) a service-wide warmpool of instances with Spark initialized or (b) provisions new instances from EC2. The warmpool uses ML models to forecast how

many EC2 instances are needed for each region and availability zone based on incoming demand as well as intra- and inter-day variability. Figure 8(b) shows that start times are mostly under 10 seconds and often under a couple seconds (Glue 2.0 to 4.0 all have the same job start times). Cold start times are shorter, rarer, and less variable because the warmpool often fulfills demand, and jobs need not wait for an entire cluster.

With faster startup and a dynamic scheduler, we had the opportunity to further optimize costs for customers, especially for streaming workloads. Glue 3.0 introduced *auto scaling* which dynamically tunes cluster size during a job. To do so, we solved two key challenges for auto scaling. First, while the Glue 2.0 scheduler allows for scaling up, we needed a way to scale down during periods of inactivity without losing intermediate state. To do so, Glue extends Spark's shuffle tracking algorithm [29] to avoid retiring workers with intermediate shuffle data that need downstream processing. Second, since resizing happens more frequently in intra-job scaling, we dampen resizing based on inactivity within and across jobs to avoid high churn on global compute resources. With auto scaling, customers get a truly serverless experience.

3.3.1 Interactive Execution. Although we originally targeted batch ETL, we quickly learned that customers needed an interactive experience for testing and debugging scripts in the Glue environment. So, for use with notebooks and local interactive development environments (IDEs), we offered Glue *development endpoints* at launch. They provided an always on experience at the cost of keeping clusters running. Glue's user base broadened as it grew, and customers started using Glue for data exploration, experimentation, and powering interactive applications like data wrangling. Though not ideal, customers would sometimes use the Jobs API for interactive applications to avoid development endpoint costs. So, in 2022, enabled by fast start and auto scaling, we introduced the Glue interactive sessions API [8] and Glue Studio Notebooks [10]. With these, customers can submit granular Spark statements that execute immediately as an extension of their development environment. Interactive sessions provide an open-source Jupyter kernel that integrates with IDEs such as PyCharm, IntelliJ, and VS Code. Glue Studio Notebooks further simplifies development by providing an in-browser integrated notebook environment, so users do not need to install IDEs.

3.4 Decoupling Storage for Cloud Shuffle

Spark uses a shuffle operator to redistribute data for large stateful transformations like Join and GroupBy, and the default shuffle materializes intermediate data onto local storage [33]. When customers process skewed datasets or under-provision local storage, their workloads often run into both memory and disk limits on individual workers. Before 2021, Glue customers only had two options to address out-of-disk failures: scale out and provision more hosts or re-partition their datasets. While this helps in some cases, it still does not guarantee reliable execution, since data skew can lead to scenarios where disk space is exhausted on just a few workers. Even when adding hosts does help, it costs customers more, as they pay for additional compute just to get more disk space.

In 2021, we introduced the cloud shuffle storage plugin that instead materializes to Amazon S3, thereby completely decoupling

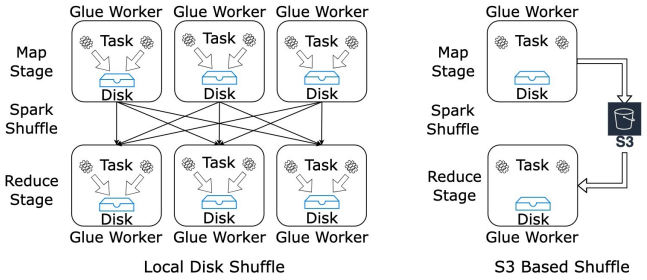


Figure 9: Decoupling compute and storage in Glue

storage and compute for Apache Spark (see Figure 9). Amazon S3 offers highly available, low-cost, and elastic storage. On the other hand, existing approaches such as Cosco [14], Zeus [26] and Magnet [28] require managing an additional storage fleet for shuffle. This plugin required us to extend components in Spark such as the block manager and shuffle reader and writers. We also added support for multi-part uploads and jitter-reducing strategies for optimizing I/O to Amazon S3.

In 2022, we also extended the plugin to operate for other cloud storage provider implementations with Glue versions 3.0 and 4.0, and released the software binaries for customers and the community to use in any Spark environment [22].

3.5 Performance with Vectorized Readers

A large fraction of Glue workloads transform data in raw formats such as CSV and JSON to modern formats such as Apache Parquet. One of the key bottlenecks for such ETL workloads is the CPU cost to deserialize the data read from S3 into memory. While over the years, Amazon S3 bandwidths have improved with advancements in networking, CPU and memory bandwidths have not kept up, especially with (Java) byte code execution.

To address this bottleneck, we introduced native SIMD vectorized readers in Glue in 2021. This required re-implementation of the Glue readers for raw formats such as CSV and JSON using a native language (C++) and use of SIMD vectorized CPU instructions. Vectorization helps us to use CPU micro-parallelism for the different steps in the reading data, thereby speeding up parsing, tokenization and indexing. Glue's vectorized readers also read data into an in-memory columnar format based on Apache Arrow [1] for improved memory bandwidth utilization and to reduce the additional cost for conversion from in-memory row to on-disk columnar formats such as Apache Parquet.

Figure 10 compares the performance of native SIMD vectorized readers against Java based implementations in Glue 4.0 to convert large datasets from CSV to Apache Parquet. It uses the largest `store_sales` table in the TPC-DS benchmark dataset [30] (3 TB) stored in Amazon S3 and 60 G2.X workers on AWS Glue. All values in `store_sales` table are numeric. With schema enforcement, we cast values into numeric data types, and without schema enforcement, we cast them to strings. Enforcing schema to numeric types allows for more compact in-memory representations and hence faster deserialization, and no schema enforcement gives more flexibility. This experiment shows Glue's native vectorized readers are

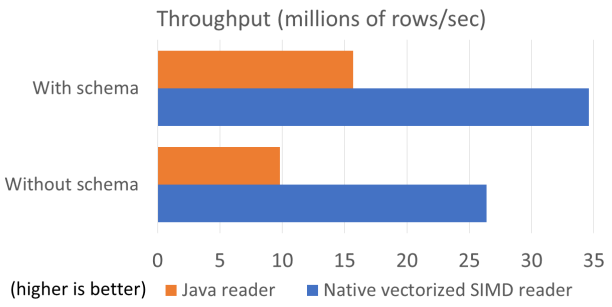


Figure 10: Glue job performance with vectorized CSV readers

nearly 2.2x faster than Java based implementation with schema enforcement and about 2.7x faster without schema enforcement.

3.6 Glue Workflows & Incremental Processing

In practice, customers often need to compose multiple jobs into pipelines that can be executed on a schedule to repeatedly process new data. To simplify this process, we built an orchestration layer into Glue that allows customers to build higher level pipelines called *workflows*. Glue workflows allow customers to build pipelines from multiple Glue crawlers, Glue Spark jobs, and Glue Python jobs that are executed together. When defining workflows, customers can define parameters to be passed between jobs, special tasks to be performed in the event of failures, and triggers to start the workflow based on a schedule or a combination of events. Customers can monitor the progress of an entire workflow or drill down into each job for troubleshooting.

Figure 11 shows an example Glue workflow that is triggered when 1000 new objects are added to an Amazon S3 bucket using Amazon EventBridge [18]. The workflow then starts a sequence of jobs and crawlers to parse the data, register the schema in the Glue Data Catalog, and write output to Amazon Redshift.

To make it easy for customers to process new data as it arrives, we built a new construct in the Glue ETL library called *Glue Job Bookmarks*. A job bookmark is the state associated with an execution of a Glue job (job run) that can be used to track the data it processed. When job bookmarks are enabled, jobs pickup from where they left off. Bookmark state is committed on job completion and is used in subsequent job runs to skip already processed data. Glue job bookmarks simplify incremental processing of Glue catalog tables, S3 bucket locations with CSV, JSON, Parquet, ORC, Avro file formats, and JDBC sources such as relational databases (MySQL, SQLServer, Aurora) with the use of one or more columns as bookmark keys.

Customers also face challenges with large initial loads. We commonly see customers with millions of files in an S3 prefix, and processing these all at once can cause job failures due to memory limits in individual Spark workers. To address this problem, we introduced the ability to *bound* the execution of a job run by limiting the number of files or dataset size processed per job. Customers can then execute the job multiple times to complete the initial load.

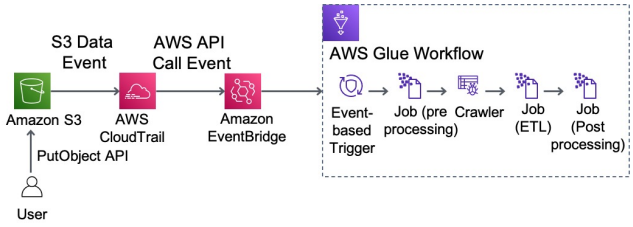


Figure 11: Data pipeline orchestration with Glue workflows

While this is a simple approach, it works surprisingly well in practice and helps customers reliably complete large migrations without having to worry about complex performance tuning.

3.7 Monitoring Pipelines and Data Quality

Today, customers also want to observe the quality of their data as it gets transformed and monitor it closely to avoid data downtime or inconsistency. Glue provides different mechanisms for customers to monitor and trouble-shoot failures such as logs and metrics in Amazon CloudWatch and detailed execution plans in Spark UI. These mechanisms give customers fine-grained information that they can use to alert and avoid downtimes for their pipelines. In 2021, we also built Glue job insights [17] to further simplify diagnosis or root cause analysis of errors for customers. With job insights, customers can now quickly retrieve meaningful error messages, line number of their application code which was last executed before the failure, and rule-based recommended action to fix the issue. With these mechanisms, customers can not only quickly alert on their pipeline failures, but also trace the issue back to their application logic and identify resulting data inconsistencies.

In 2022, we released tools to help customers evaluate and monitor quality of both in-transit and at-rest data. Built on top of the open-source DeeQu framework [19], our solution allows customers to express their data quality rules against dimensions such as data accuracy, freshness, and integrity. Customers can select from recommended data quality rules or implement their own rules using a Data Quality Definition Language. They can then run data quality checks which evaluate data quality using these rules and compute a quality score. Customers can monitor the score to further take action or decide if the dataset is fit for use, and they can publish these metrics to Amazon CloudWatch.

3.8 Connectivity for Data Integration

We launched AWS Glue with support for reading and transforming data from a variety of popular AWS storage services, including Amazon S3, Amazon RDS, and Amazon Redshift. As Glue has grown into a full-fledged data integration service, customers increasingly rely on it to connect to their data, wherever it is stored. We have found this to be reminiscent of Metcalfe's law [25] – the value of the platform increases quadratically with the number of sources and sinks it supports. Thus it makes sense for us to support as many formats, sources, and sinks as possible, and to make it possible for customers to extend to their own legacy or unique formats. To meet this challenge, we invested in connectors along two dimensions. First, we continued to optimize high volume data sources like Amazon S3 and Amazon Redshift to improve performance. Second, we

have greatly expanded the variety of data sources in AWS Glue to support new storage formats and SaaS services.

Since it is so widely used for data lakes, Amazon S3 has been a particularly important data source for AWS Glue customers. Glue launched with basic support for common open file formats such as CSV/Text, JSON, Avro, XML, and columnar storage formats such as Apache Parquet and ORC. Since then, we introduced a variety of optimizations to improve performance and reliability. These include (a) access path optimizations such as automatically batching small files into tasks to reduce per task overhead, (b) mechanisms to reduce the amount of data that has to be read, such as partition pruning with the Glue Data Catalog or Job Bookmarks, and (c) support for new features, such as special handling of S3 storage classes like Amazon Glacier or fine-grained access control with AWS Lake Formation. In 2022, we also built native support for open data lake table formats such as Apache Hudi [2], Apache Iceberg [3] and Linux Foundation Delta Lake [5] into AWS Glue. These formats provide transactional (ACID) guarantees to data lake tables in S3. We have also enhanced our native connectors to other AWS services including Redshift [6], DynamoDB [15], and Aurora [32].

In addition to improving native connectors to high-volume data sources, we also introduced support for a wide variety of new data sources. In 2020, we introduced AWS Glue custom connectors, which allow Glue to connect to a wide range of SaaS applications such as Salesforce, cross-cloud data stores such as Google BigQuery, and data warehouses such as Snowflake [16]. Customers can subscribe to these connectors in the AWS Marketplace and use them in their Glue jobs. To facilitate the development of new connectors, we also released a collection of SDKs that make it easy to adapt existing connectors built against the Spark DataSource API, JDBC, or the Amazon Athena Federated Query API. Finally, we established a certification process for developers to add a new Glue connector to the AWS Marketplace [16].

4 THE AWS GLUE DATA CATALOG

The AWS Glue Data Catalog is a metadata repository for datasets that customers work with in AWS. It stores information such as data location, schema, and format for tables that may reside in Amazon S3 or a variety of other databases and NoSQL stores. Like the rest of AWS Glue, the Data Catalog has evolved to meet the changing needs of our customers, and has grown to be a central component in many AWS analytics services.

4.1 Background

As customers increasingly build and manage persistent data lakes in data stores like Amazon S3, they need mechanisms to discover and manage metadata about their datasets. Query engines need metadata such as schemas and data locations to plan and execute queries, and customers rely on metadata for data discovery and governance across large organizations. Traditional databases store this metadata in an internal catalog, but in a data lake setting where many engines can be used to query the same datasets, the metadata must be decoupled from the query engine.

The open source community pioneered a solution in this space with the Hive metastore, which has become a de-facto standard in the Hadoop ecosystem for metadata management [24]. It provides a common interface for accessing metadata about databases, tables, and partitions, and it is widely supported by open-source query engines such as Apache Hive, Trino, and Apache Spark. While the Hive metastore is widely deployed and battle-tested, it has some limitations that make it insufficient for managing large data lakes. First, it becomes yet another system that a data lake administrator has to manage. The standard implementation of the Hive Metastore uses a relational database, and customers are responsible for provisioning, scaling, and patching the metastore. Performance is also a challenge, and users often have to shard large Hive metastores, which introduces an extra layer of complexity.

4.2 Data Model and Architecture

The Glue Data Catalog provides a set of public AWS APIs for customers to store and retrieve metadata. Since one of our goals was to provide a managed replacement for the Hive Metastore, we largely adhere to the Hive Data Model and provide CRUD APIs for *databases*, *tables*, and *partitions*. Among other items, the table metadata includes the type of data (e.g. JSON), the schema, how the data is partitioned (e.g. year, month, and day), and the specific Hive Serdes that can be used to read the dataset. Query engines and other users can either process this JSON directly, or use an open-source adapter to translate the output into the metastore interfaces expected by Apache Hive or Apache Spark.

While we follow Hive conventions in order to achieve compatibility with query engines, we intentionally do not enforce compliance with the Hive data model. For example, the schema is defined as a list of columns, but the types are simply text fields that callers can fill in with any value. While the Glue console and crawlers (Section 5) attempt to be Hive compatible, a sizable fraction of tables contain at least one data type not in the Hive standard type system, which often come from customers building custom applications on the Glue Data Catalog or query engines with richer type systems than Hive. We see similar patterns with other Hive-specific fields like `InputFormat` [24]. This choice not to validate is not without tradeoffs – it means that some tables cannot be read by some query engines, but we found that the data lake space is too diverse and too fast-moving to enforce a unified data model for everyone.

While customers most commonly use the Glue Data Catalog to store metadata about datasets stored in S3, the model is flexible enough to catalog datasets from a wide variety of different sources, from relational databases to NoSQL databases like MongoDB, and streaming sources like Amazon Kinesis or Apache Kafka. To support these data sources, the Glue Data Catalog also supports *connection* objects, which provide information on the physical connection requirements for specific data stores. A relational database in AWS may be configured so that it is only accessible from a specific Virtual Private Cloud (VPC), and the connection stores information about the subnet and security group required to connect, as well as either encrypted credentials or a reference to login information in AWS Secrets Manager. Even for S3, customers can use connections to control how their data is routed in order to enforce security or data sovereignty requirements. Connections are stored in the Glue Data Catalog and can be referenced by tables and passed to ETL jobs.

The Glue Data Catalog is built on-top of low latency and highly scalable storage. Its storage implementation also offers predictable

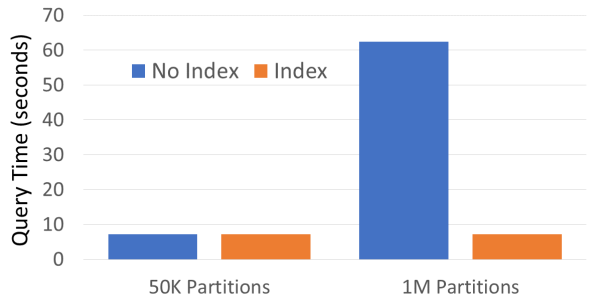


Figure 12: Query performance with partition indexes

performance and high availability for hundreds of thousands of customers monthly. While standard storage optimizations worked well for problems such as skewed data and atomic table updates, as customers started working with larger-and-larger datasets, partition pruning became another bottleneck. As described in section 2.2.3, big data query engines make heavy use of partitioning to improve query performance by skipping files. By default, query engines enumerate all partitions for a table and filter them client-side based on the query predicate. This is reasonable when the number of partitions is small, but partition enumeration can become a significant bottleneck when querying tables with millions of partitions.

To address this, we added support for partition indexes in 2020. Customers can create an index on one or more partition attributes, and they will be stored separately with support for efficient range queries. This means that query engines can push partition predicates all the way down to the Glue Data Catalog and only retrieve the matching partitions. Figure 12 shows the benefits of a partition index on a simple query that performs a count distinct over a single partition. When the table has only 50,000 partitions, the index does not make significant performance difference, but with one million partitions the query is 8.6 times faster with the partition index.

4.3 Extensibility and Discovery

While it started as a replacement for the Hive Metastore, the success of the AWS Glue Data Catalog shows the value of having a centralized, managed, and easily accessible repository for metadata. Today, the Glue Data Catalog serves as the main metadata store for data integration with Glue ETL jobs, query engines such as Amazon Athena and Amazon Redshift, and is widely used from Apache Spark and Apache Hive on Amazon EMR. Beyond query engines, the Glue Data Catalog is becoming a central integration point for services that need to interact with customer datasets. For example, AWS Lake Formation allows customers to enforce fine-grained access control policies on entities in their data catalogs, and the AWS Glue Schema Registry allows customers to specify schemas separately from tables so that they can be reused by analytics applications and streaming services like Amazon Kinesis.

5 GLUE CRAWLERS

The Glue Data Catalog provides a repository to store metadata about the data lake, but it is only valuable insofar as it is kept up-to-date with accurate information. Glue ETL scripts are one way to populate the Data Catalog – scripts can be configured to register their output directly in the Data Catalog – and customers can also use DDL statements from systems like Amazon Athena to create databases and tables directly. While many customers opt for these solutions, they do not address all use cases. For example, customers have large amounts of existing data in S3 that would be impractical to re-process using Glue ETL, and customers may not know the schema of their data up-front to use in DDL statements. One of our goals with Glue is to support a wide variety of semi-structured data, for example JSON logs, which may have undefined or rapidly changing schemas.

AWS Glue crawlers help address these challenges by scanning data in S3 and automatically populating tables and partitions in the Data Catalog without requiring manual configuration. Customers simply specify a set of S3 prefixes and a destination database in the Data Catalog, and Glue will crawl the files under those prefixes, identify their types and schemas, and create or update the appropriate tables and partitions in the Data Catalog. Crawlers can be used to identify schema changes in rapidly evolving datasets or to register new partitions after an hourly ingestion. Crawlers have been running in production since the launch of Glue and currently process tens of billions of files per day.

In this section, we focus on crawling data stored in Amazon S3, as this makes up the vast majority of data lakes in AWS. Crawlers can also be used to crawl other systems, such as relational databases (via JDBC), or NoSQL stores such as Amazon DynamoDB.

5.1 Architecture

Glue crawlers list and scan Amazon S3 files in parallel, infer their type and schema, and then perform a post-processing step, which we call the *finalizer*, to analyze the inferred schemas and populate the Data Catalog with the appropriate tables and partitions. We describe these stages below.

Listing and Classification. The first stage of crawling is to enumerate the files in a collection of S3 prefixes. These files are batched into tasks that are executed in parallel during the classification stage. The primary job of this phase is to determine the file format, compression type, and schema of each file in the dataset. This metadata is aggregated at the prefix level and stored for future processing by the finalizer.

In order to limit the amount of data that we must scan, each crawler looks at only the first megabyte of each file and uses the same algorithm described in Section 3.2.1 to infer the schema. While this means that it is possible that the crawler might infer only a subset of the actual schema, for example missing attributes that are not present in the first megabyte, we find this to be rare in practice, and systems like Glue ETL have additional logic for automatic schema inference to handle fields not present in the Data Catalog.

Glue crawlers identify file types and schemas using a collection of *classifiers*, each of which is responsible for determining whether a file matches a specific format. These classifiers use a variety of format-specific techniques to determine the file format and extract

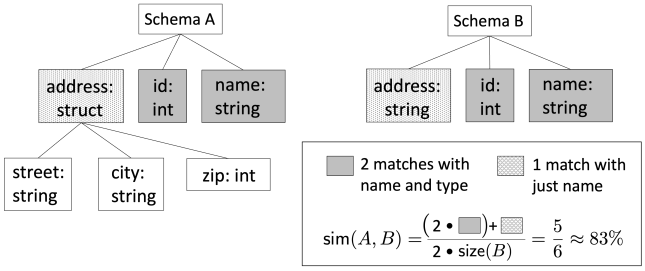


Figure 13: Computing schema similarity

the schema. For some file types this is straightforward. For example, all Apache Avro files start with a common four byte magic number `Obj1` and contain the file schema serialized as JSON in the header. Text based formats such as CSV require additional heuristics. For example, to infer the delimiter, the crawler will try to parse a few lines using common delimiters and see which produces more consistent records. In a few cases, the crawler may need to fetch additional data from S3. This is only required for binary files like Apache Parquet, which store the schema explicitly in the footer.

Finalizer. Once the crawler has completed classifying each file in the dataset, the *finalizer* is responsible for identifying tables and partitions and populating the Glue Data Catalog. The primary task is to categorize each prefix in S3 as either a table or a partition. For example, consider a prefix like `/Orders/EMEA/2022/01/09/`. Based on experience, we would expect this to correspond to a table called Orders that is partitioned by region, year, month, and day. To automate this, we start with the assumption that partitions in a table are likely to have the same or similar schemas, whereas the schemas of two different tables may differ significantly. Note that we say similar schemas – when dealing with semi-structured data, it's very likely that two files may have slightly different schemas even within a single partition. For example, in an event stream, the fields in a record may depend on the type of event, and not every event may appear in every file.

To formalize this, we define a simple similarity metric between two schemas. A field present in both schemas is worth one point if the names match, and two points if both the names and the types match. For schemas A and B, call this $\mathsf{intersect}(A, B)$. Then the similarity between A and B is $\mathsf{intersect}(A, B)/(2{\cdot}\min(\mathsf{size}(A), \mathsf{size}(B)))$, where $\mathsf{size}(A)$ is the number of fields in the schema A. Figure 13 shows an example. In this example schemas A and B have the same name and type for the `id` and `name` fields, but the `address` field is a struct in schema A and a string in schema B. As depicted, we compute the similarity between these two schemas as 83%.

The finalizer traverses the metadata collected during the classification stage and computes the schema similarity at each sibling prefix. If each prefix similarity is above a percentage threshold, then we infer the prefixes as partitions. Anything less than that threshold and we treat them as separate tables. This is clearly a heuristic and it is not foolproof, but we find that it does a reasonable job in allowing for the natural variance in schemas between partitions.

Recrawling. Glue crawlers allow customers to incrementally crawl or recrawl only the new S3 partitions that were added since the last crawl run. Using an S3 events based crawler, customers can reduce crawl times significantly as crawls are now targeted to changed folders. These options save on both time and cost for updating the Glue Data Catalog.

Extensibility. Semi-structured data often does not conform to a classification format or schema. In this case, Glue crawlers provide customers with the flexibility to use custom classifiers, which can be defined as a grok pattern, XML tag, or JSON/CSV based pattern. Custom classifiers are evaluated before built-in classifiers to ensure they take priority when multiple classifications match.

6 CONCLUSION

In this paper, we presented AWS Glue, Amazon's serverless data integration cloud service. We launched Glue in August 2017, and in the intervening six years, it has grown into a full-fledged data integration service that is used for a staggering variety of use cases, from interactive data discovery and exploration to large scale data transformation. Throughout this process, we have learned a variety of lessons about building and running cloud data services at scale. The first is that getting adoption for a general purpose platform is challenging early on, and we found the most success by focusing on a narrow set of customers and use cases – in our case developers loading data warehouses and data lakes. Once we gained traction with them, we were able to quickly expand to new users by leveraging the underlying platform, for example by building Glue Studio to serve less technical users.

We also found that we got disproportionate benefit from investing in our underlying serverless compute platform. Decoupling storage from compute through features like the cloud shuffle plugin make the system more flexible and enabled features like autoscaling. Enabling fast startup time was also extremely powerful. By making very short jobs feasible, it enabled a whole new class of applications such as interactive notebooks and data wrangling. Not all of these applications were obvious when we started working on startup time, but our customers have consistently shown that they will find novel ways to leverage platform-level improvements.

Finally, data integration services live or die by their ecosystem of connectors. Early on, Glue found success by helping customers discover, catalog, and transform the many types of semi-structured data they had in Amazon S3. As data formats and data stores have continued to evolve, we have continued to extend Glue, for example by adding support for transactional data formats like Apache Iceberg and Apache Hudi, and by adding Glue connectors to the AWS Marketplace to let customers and partners write their own.

The AWS Glue story does not end here. Guided by our tenets, we continue working with customers to drive innovations in ease of use, serverless execution, extensibility, and performance to reduce the overall costs of data integration.

ACKNOWLEDGMENTS

AWS Glue has greatly benefited from thousands of Glue customers whose continuous feedback helped the team to innovate on their behalf. We thank Anurag Gupta for setting the original direction for Glue. Most importantly, we thank the amazing past and present Glue team members for their impactful contributions and the moments we had during the course of this evolution.

REFERENCES

[1] Apache Arrow - a cross language development platform for in-memory analytics. Last accessed: 02-27-2023. https://arrow.apache.org/.

[2] Apache Hudi - Hadoop Upserts Deletes and Incrementals. Last accessed: 02-27-2023. https://github.com/apache/hudi.

[3] Apache Iceberg - open table format for analytics datasets. Last accessed: 02-27-2023. https://github.com/apache/iceberg.

[4] Apache Parquet. Last accessed 02-27-2023. https://parquet.apache.org/.

[5] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Luszczak, Michael Switakowski, Michael Szafranski, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[6] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, TJ Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift re-invented. In *SIGMOD/PODS 2022*. https://www.amazon.science/publications/amazon-redshift-re-invented

[7] AWS Glue - Glue Studio documentation. Last accessed: 02-27-2023. https://docs.aws.amazon.com/glue/latest/ug/what-is-glue-studio.html.

[8] AWS Glue - Interactive Sessions API. Last accessed: 02-27-2023. https://docs.aws.amazon.com/glue/latest/dg/aws-glue-api-interactive-sessions.html.

[9] AWS Glue - Jobs API Documentation. Last accessed 02-27-2023. https://docs.aws.amazon.com/glue/latest/dg/aws-glue-api-jobs.html.

[10] AWS Glue - Overview of using notebooks. Last accessed: 02-27-2023. https://docs.aws.amazon.com/glue/latest/ug/notebook-getting-started.html.

[11] AWS Glue Libraries. Last accessed: 07-09-2023. https://github.com/awslabs/aws-glue-libs.

[12] AWS re:Invent 2017: Building Serverless ETL Pipelines with AWS Glue (ABD315). Last accessed: 07-09-2023. https://www.youtube.com/watch?v=eQBHIINW8VY.

[13] AWS re:Invent 2020: Serverless data preparation with AWS Glue. Last accessed: 07-09-2023. https://www.youtube.com/watch?v=pT5lAYTCYJ4.

[14] Cosco: An Efficient Facebook-Scale Shuffle Service Databricks. Last accessed: 02-27-2023. https://databricks.com/session/cosco-an-efficient-facebook-scale-shuffle-service.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[16] Developing, testing, and deploying custom connectors for your data stores with AWS Glue. Last accessed: 02-27-2023. https://aws.amazon.com/blogs/big-data/developing-testing-and-deploying-custom-connectors-for-your-data-stores-with-aws-glue.

[17] AWS Glue Job Insights Documentation. Last accessed: 02-27-2023. https://docs.aws.amazon.com/glue/latest/dg/monitor-job-insights.html.

[18] Amazon EventBridge. Last accessed: 02-27-2023. https://aws.amazon.com/eventbridge/.

[19] Deequ Unit Tests for Data. Last accessed: 02-27-2023. https://github.com/awslabs/deequ.

[20] GHArchive. 2022. GH Archive. https://www.gharchive.org/

[21] Pat Helland. 2019. Extract, Shoehorn, and Load: Data Doesn't Always Fit Nicely into a New Home. *Queue* 17, 2 (apr 2019), 13–17. https://doi.org/10.1145/3329781.3339880

[22] Introducing the Cloud Shuffle Storage Plugin for Apache Spark. Last accessed: 02-27-2023. https://aws.amazon.com/blogs/big-data/introducing-the-cloud-shuffle-storage-plugin-for-apache-spark.

[23] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 330–339. https://doi.org/10.14778/1920841.1920886

[24] Apache Hive Metastore. Last accessed: 02-27-2023. "https://cwiki.apache.org/confluence/display/hive/design#Design-Motivation".

[25] Bob Metcalfe. 2013. Metcalfe's Law after 40 Years of Ethernet. *Computer* 46, 12 (2013), 26–31. https://doi.org/10.1109/MC.2013.374

[26] "Zeus: Uber's Highly Scalable and Distributed Shuffle as a Service". Last accessed: 02-27-2023. "https://www.databricks.com/session_na20/zeus-ubers-highly-scalable-and-distributed-shuffle-as-a-service".

[27] Spark Job Scheduling. Last accessed: 02-27-2023. https://spark.apache.org/docs/latest/job-scheduling.html.

[28] Min Shen, Ye Zhou, and Chandni Singh. 2020. Magnet: Push-Based Shuffle Service for Large-Scale Data Processing. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3382–3395. https://doi.org/10.14778/3415478.3415558

[29] Spark Dynamic Allocation Shuffle Tracking. Last accessed: 02-27-2023. "https://spark.apache.org/docs/latest/configuration.html#dynamic-allocation".

[30] TPC-DS 2.13 (3 TB) Dataset. Last accessed: 02-27-2023. "https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.13.0.pdf".

[31] Dimitris Tsirogiannis, Nathan A. Binkert, Stavros Harizopoulos, Mehul A. Shah, Benjamin A. Sowell, Bryan D. Kaplan, and Kevin R. Meyer. 2019. Scalable analysis platform for semi-structured data. Patent No. US10275475B2, Filed Mar. 14th., 2014, Issued Apr. 30th., 2019.

[32] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD 2017*. https://www.amazon.science/publications/amazon-aurora-design-considerations-for-high-throughput-cloud-native-relational-databases

[33] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing *(NSDI'12)*. USENIX Association, USA.