Microsoft

Azure Data Explorer: a big data analytics cloud platform optimized for interactive, ad-hoc queries over structured, semi-structured and unstructured data

Introduction

The world of Big Data is growing steadily, and the number of technologies that process large amounts of data is growing along with it. Modern Big Data solutions are typically composed of multiple technologies, each addressing a particular set of requirements. The usual activities include data collection, ETL, enrichment, batch processing, real-time stream analytics, dashboards, interactive exploration, reporting, aggregation and summarization, as well as training and servicing ML models.

In this whitepaper we present the platform and technology that underlies the Azure Data Explorer (ADE) service offering (previously codenamed Kusto). ADE fits a specific set of needs within this landscape, and it's best illustrated with an example. Let's imagine a large-scale cloud service deployed on thousands of VM's, handling complex user transactions under strict SLA's and interacting with other external services. When properly instrumented, such a service emits an enormous number of signals: generic performance events from each VM (e.g. memory, disk and CPU usage), application-specific events (e.g. user authentication, start/progress/end of a transaction), internal health alerts and, last but not least, trace and debug statements added by the developers to aid in the service troubleshooting. Various subsets of these signals are routed to different systems: chosen performance and application events to the stream analytics solution for real-time alerting, generic performance counters to the time-series DB for operational dashboards, high-level application events to the data warehouse for further business analytics, etc. Our platform was designed to get all this data in its raw, unprocessed form, store it for a limited time interval (i.e. a sliding window to a subset of data ranging from a few days to a few months), and allow fast ad-hoc queries and analytics over this diverse data.

More formally, the main ADE features are:

1. Ability to work with any kind of data: structured, semi-structured (JSON and more) and unstructured (free text).
2. High velocity (millions of events per second), low-latency (seconds) and linear scale ingestion of raw data.
3. Industry-leading scalable query performance.
4. Rich and powerful query capabilities, supporting the whole complexity spectrum: from a simple keyword search to the complex time series and behavioral analytics.
5. Simple, yet powerful and productive query language and toolset.

The target data workload of the technology can be characterized as follows:

- Sliding window to a subset of chronological data
- Read-many
- Insert/append-many
- Delete-rarely (bulk delete).

- Update almost never

Service overview

A typical deployment of the platform for a given customer (tenant) consists of two distinct services working in tandem: the **Engine** service and the **Data Management** (DM) service. Both services are deployed as clusters of compute nodes (virtual machines) in Azure.

The Engine service is responsible for processing the incoming raw data and serving user queries. It exposes a JSON API endpoint through which users interact with the service by sending queries and control commands.

The DM service is responsible for connecting the Engine to the various data pipelines, orchestrating and maintaining continuous data ingestion process from these pipelines (including robust handling of failure and backpressure conditions) and invocation of the periodic data grooming tasks on the Engine cluster. In principle, DM service is optional, though in practice it is present in the vast majority of Kusto deployments.

The focus of this document is the Engine service, although we'll occasionally refer to the DM when discussing the relevant functionality.

Logical model

The engine service exposes a familiar relational data model:

- At the top (cluster) level there is a collection of databases;
- Each database contains a collection of tables and stored functions;
- Each table defines a schema (ordered list of typed fields);
- There are various policy objects that control authorization, data retention, data encoding and other aspects; these can be attached to a database, a table and sometimes to a table field.

Unlike a typical RDBMS, there are no primary/foreign key constraints in ADE (or any other constraints, such as key uniqueness), and the necessary relationships are established at the query time. There are at least two reasons for the lack of such formal constraints: first, they would be constantly violated with the kind of raw and noisy data that the system is intended to handle; second, enforcement of these constraints in a large distributed system would result in a substantial negative impact on the data ingestion rate.

All the user interaction with the service falls into one of the two broad categories: queries and control commands. Queries reference any number of tables (including tables in different databases or even clusters) and apply a variety of composable relational operators to express the desired computation. The result of the query is one or more tabular data streams. Queries are read-only, they never mutate the data or the logical model.

Control commands may inspect and mutate the metadata objects (e.g. create a new database or a new table, change a schema of an existing table). The most frequent control command is “ingest data”, appending a new batch of records to the table.

The actual data of the logical table is stored in a number of horizontal data shards and possibly in a few *row stores* (row stores are discussed later in the document). For the most part the user is not concerned with the data shards, and just treats a table as a single logical entity. However, in some cases exposing the notion of the data shard is useful, therefore they're not entirely hidden from the user.

The engine service implements an RBAC-style authorization model at the cluster, database and, with some limitations, table granularity.

The hierarchy of databases, tables, associated policies and data shard references form a *metadata tree* – a complete and consistent description of the Engine service instance.

Engine cluster architecture

The engine cluster is a collection of compute nodes (virtual machines) in Azure, connected to a VNET, with the gateway nodes accessible externally through the load-balancer. Once the cluster is provisioned, it can be scaled up or down (in terms of the number of machines), either automatically or manually, in response to the changing load and/or data volume. The minimum engine cluster size is two VM's, the maximum size tested in production is about 500 machines.

Each node in the cluster fulfills one or more of the following roles:

1. **Admin Node**: responsible for maintaining the cluster metadata and performing the metadata transactions.
2. **Query Head**: holds a read-only view of the metadata, responsible for accepting and processing the query (building the distributed query plan) and then initiating and orchestrating the distributed query execution.
3. **Data Node**: the most common role in the cluster. Data node contributes its memory and disk space to the caching of the data shards; executes fragments of the distributed query; creates and caches new data shards from the ingested batches of raw data. Data nodes expose internal RPC endpoint for communication with other nodes.
4. **Gateway**: accepts the external API calls, performs the client authentication and dispatches the request to the appropriate handler for further processing (e.g. control commands to the Admin node, queries to the one of the Query Head nodes).

Cluster state and metadata transactions

ADE relies heavily on the Azure Blob service as a durable, highly-available storage both for the data shards and for the cluster metadata.

Each data shard has a unique ID and is represented by a number of blobs in the storage (we describe the shard format in a later section). A database maintains a list of available *storage containers* (storage account URI + container name), configured by the database administrator. Each data shard resides in one of those containers. A table definition holds a list of references to its data shards (i.e. list of shard ID's), along with the corresponding storage containers. Therefore, data shards are spread across multiple storage accounts and containers, balancing the load on the Blob service and allowing for very efficient manipulations on a group of shards at the container level.

The latest snapshot of the metadata (along with the previous versions) is always kept in the Azure blob. The Admin node also loads it in memory and maintains it as immutable data structure (with efficient cloning and copy-on-write support). A metadata transaction clones and modifies the in-memory object tree, then commits a new snapshot to the blob and makes the new metadata tree visible to the rest of the cluster. Depending on the kind of the transaction, the snapshot written to the blob could be either a full one or a delta snapshot (to save on bandwidth and speed up the commit flow).

If the current Admin instance becomes unavailable, a new Admin is immediately elected. The new Admin node loads the latest metadata snapshot from the blob, surveys the cluster nodes to collect their runtime state and continues with the normal operation.

With the exception of the Row Store (a complementary storage technology within the platform, discussed later in this document), all of the data transactions happen at the shard granularity. The most frequent transactions are related to the data ingestion (creates a new shard and appends a shard reference to the table metadata) and retention (deletes a shard when it falls out of the chronological "sliding window"). Following are the detailed steps of the data ingestion transaction:

1. An ingestion command arrives at the Admin node; it specifies the target table and the list of data sources (e.g. list of CSV file URL's).

2. Admin finds an available Data node to perform the processing and forwards the ingestion command to this node.
3. Data node fetches and processes the data sources, creates a new shard, writes its artifacts to the blob storage, and returns the description of the new (uncommitted) shard to the Admin
4. The Admin adds the new shard reference to the table metadata and commits the delta snapshot of the database metadata.

When the engine starts processing a new query, it takes a snapshot of the current metadata and attaches it to the query until it is completed.

When a data shard is deleted (either due to the retention, manual table drop command or for any other reason), its storage artifacts (blobs) are not immediately deleted from the Blob Storage. Instead, they are garbage collected at the container level later, when that container reaches the hard delete threshold. This scheme enables two features:

1. In-progress queries that started before the metadata change can complete successfully (they continue to execute with the stale snapshot)
2. It's possible to revert the metadata to a previous version in case of a mistake.

(the only exception here is the "data purge" command executed due to privacy reasons and used for the GDPR compliance, in which case all of the relevant storage artifacts are deleted immediately).

Shard distribution

Each data shard is assigned to one of the cluster nodes by a consistent hash function. This is a "soft" assignment: the node does not own or manage the shard lifetime in any way. Instead, the assigned node is the preferred one to cache the shard locally (in memory and on disk) and, consequently, is also the preferred one to execute the part of the distributed query relevant to this shard.

Whenever there is a change in a set of participating cluster nodes (e.g. a node goes down, a cluster is scaled-out), consistent hash redistributes the proportional number of data shards across the remaining nodes.

Query Execution

We'll start with an overview of the Kusto Query Language (KQL). While the engine can accept both KQL and SQL queries (supporting a subset of T-SQL), KQL is the primary means of interaction with the service.

Query language overview

KQL was designed with the "simple things must be simple" philosophy in mind. The main design goals were: low barrier of entry, easy to read syntax, ability to "scale" the language and provide a smooth transition from simple one-liners to complex data processing "scripts". KQL draws inspiration from the *nix shells for its syntax and from the C# LINQ framework for its semantics.

At the top level, a query is a sequence of one or more statements, with at least one of these statements producing a tabular result set (the tabular expression statement) as the result of the query. This statement is modeled as a flow of data from one tabular query operator to another, starting with some tabular data reference and concatenated by the pipe (`|`) character. This pipe is just a convenient mechanism for operator composition: conceptually, it takes the output of the previous operator (to the left of the pipe) and makes it an input of the next operator (to the right of the pipe). Most tabular operators are parameterized by scalar expressions that refer to the data fields of the incoming record stream and operate in the context of the "current record".

For example, the following query starts with a table called `StormEvents` (the cluster and database that host this table are implicit here), picks up just the records for which the `StartTime` column is within the month of November 2007, from that set it keeps only the records for which the value of the `State` column is the string *FLORIDA*, and then returns the number of the records in the "surviving" set.

```
StormEvents
| where StartTime >= datetime(2007-11-01) and StartTime < datetime(2007-12-01)
| where State == "FLORIDA"
| count
```

Some common query operators:

- **where** - filters the source by applying the specified predicate
- **extend** – computes and adds a new field to the source records
- **summarize** – computes a number of aggregate functions for each distinct group of records that belong to the same key
- **join** - merges the rows of two tables to form a new table by matching values of the specified column(s) from each table. Supports common join flavors (inner-, outer-, semi-, etc.)
- **union** - takes two or more tables and returns the rows of all of them

Here is a more involved example, adapted from one of our typical troubleshooting queries:

```
let RelevantLogs =
Logs
| where TraceTimeStamp between (datetime(2018-01-01 14:00) .. 1d);
RelevantLogs
| where EventText has "Event: NotifyUserAuthenticated (token=<User="
| extend UserID = extract(@'User=(\w+)', 1, EventText)
| join kind=inner (
    RelevantLogs
    | where Level == "Error”
    | summarize by ClientActivityId
    ) on ClientActivityId
| summarize ErrorCount = count() by UserID
| top 20 by ErrorCount desc
```

Here, the `Logs` table continuously receives the raw traces coming from one of our production systems that handles user requests. Each user request generates many traces, spread both across time and across several processing nodes. `ClientActivityId` field is a correlation ID common to all traces from the same user activity. **`Let`** statement defines a reusable part of the query: `Logs` table narrowed down to a specific 1-day interval. The first part of the query finds all trace records emitted when the user is authenticated (this happens early on in the request processing, and these traces have a characteristic “Event: …” pattern in their text) and invokes `extract()` function to parse the hashed UserID value from the free text of the trace message using a regular expression. **`Extend`** operator adds this parsed UserID to the output record stream. The second part of the query looks for all appearances of the error-level traces in the same interval of the Logs table (**`where`** `Level == “Error”`). **`Join`** operator then correlates both datasets on the `ClientActivityID`. Another way to interpret the above: we discover all errors in the traces, then “go back” in the story leading to the error (for each distinct ClientActivityID) up to a point where the UserID is traced and extract this UserID, “attaching” it to the later error trace record. **`Summarize`** operator then counts the number of errors experienced by each user. Finally, the **`Top`** operator retains the top 20 users with the highest error count.

There is much more to the query language than this short introduction, but it will suffice for the query engine design discussion. We recommend the following materials to study the query language and get the broader view of its features:

- KQL from Scratch course on Pluralsight (https://www.pluralsight.com/courses/kusto-query-language-kql-from-scratch)
- KQL reference documentation (https://aka.ms/kql)

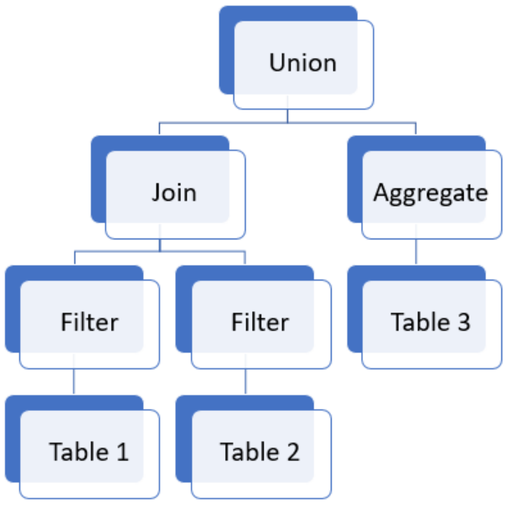
Query analysis

Upon receiving the query text, the engine parses it into an Abstract Syntax Tree (AST). KQL has a few constructs that serve as syntactic convenience (aka "syntactic sugar"), these constructs are lowered to the more primitive AST nodes at this stage.

Next, the engine performs the so-called *semantic pass* over the AST. The objectives of this pass are:

- Name resolution: find out which schema entities (databases, tables, fields, functions) are referenced by all the non-keyword names in the query
- While the entities are resolved, verify that the user has the necessary permissions to access them
- Type check and inference: ensure that the expected and the actual types of all expressions and operator arguments are consistent

Finally, the query analyzer builds an initial *Relational Operators' Tree* (RelOp tree) from the simplified and verified AST. Nodes of this tree represent relational operators (such as Table Access, Selection, Projection, Join, Union, etc.), where each operator consumes zero or more tabular inputs (usually these are the outputs of the child operator nodes) and produces a single tabular output:



This operator tree is the main representation of the query in the engine; most of the analysis and the optimizations are performed on this RelOp tree.

SQL query undergoes a similar processing, the end resulting being the same abstract RelOp tree.

Query optimization

Once the initial RelOp tree is built, the goal of the query optimizer is to transform it into an efficient distributed query plan. Note that this initial RelOp has no notion of query distribution, it is just a tree of abstract operators where the leaves are table accesses.

Query optimizer defines a large set of *rewrite rules*. Each rewrite rule can match a certain pattern of operators in the tree and apply some local transformation to these operators. Query optimization is achieved by iteratively matching and applying many rewrite rules to the RelOp tree. Most of the rewrite rules are applied unconditionally, whenever they match a target pattern of operators in the tree and these operators satisfy some additional constraints. However, some rules are applied based on the cost estimation (such as reordering of the *Join* inputs).

Subsets of related rewrite rules are grouped into *passes*, where each pass performs a certain optimization task. For instance, one of the rewrites in the predicate push-down pass recognizes *Extend* operator followed by a *Filter*. If the filter predicate expression does not depend on the values computed by the preceding *Extend* operator, the application of the rewrite rule reorders the two operators (thus pushing the Filter down, towards the leaf Table access).

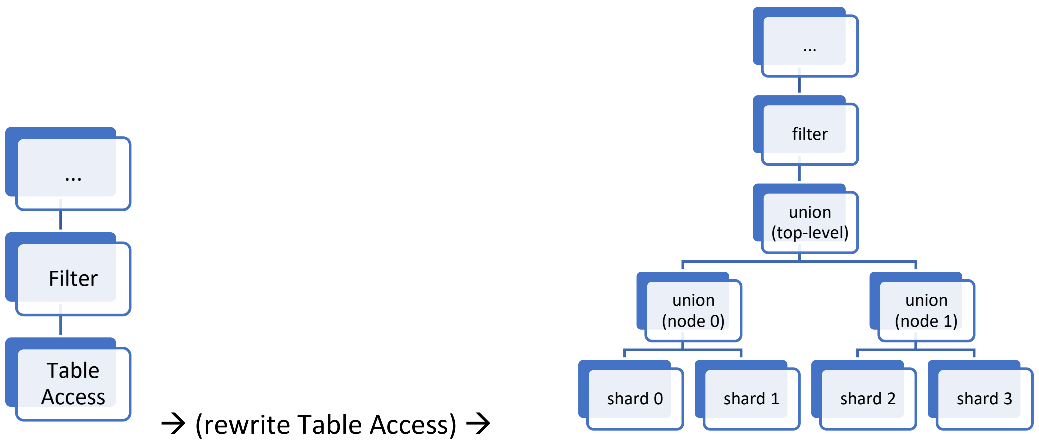
Most of the optimizations performed at this stage are not specific to ADE and are common across many relational databases:

- Propagation of projections
- Predicate push-down
- Constant folding
- Elimination of unused expressions
- etc.

Distributed query

The next phase after the "generic" optimizations is to transform an abstract query into the distributed one, considering the available compute nodes of the Engine cluster and the tables' data shards assigned to these nodes.

The general approach is to represent an abstract *Table Access* operator as the union of the data shards that comprise the table. Taking this one step further, even more practical approach is to employ a hierarchy of such unions: first, consider the union of the data shards (for the table in question) on the specific cluster node, then take the union of all the nodes:



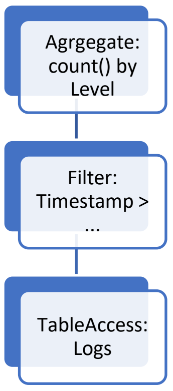
The query optimizer replaces all *Table Access* operators in the RelOp tree with the two-level shard union structure (as illustrated in the diagram). Then the optimizer runs a dedicated distributed optimization pass, relying on the same framework of rewrite rules. These rules attempt to propagate relational operators through the node/shard unions, down to the data shard level, and these are mostly the same transformations as the ones applicable for the regular *Union* operator. In particular:

- Filters are pushed down through the unions;
- Aggregation operators are split into the "leaf" aggregation (below the Union node) and "merger" aggregation (above the Union node), adjusting the aggregate functions accordingly;
- Top-N operator is replicated under the Union;
- etc.

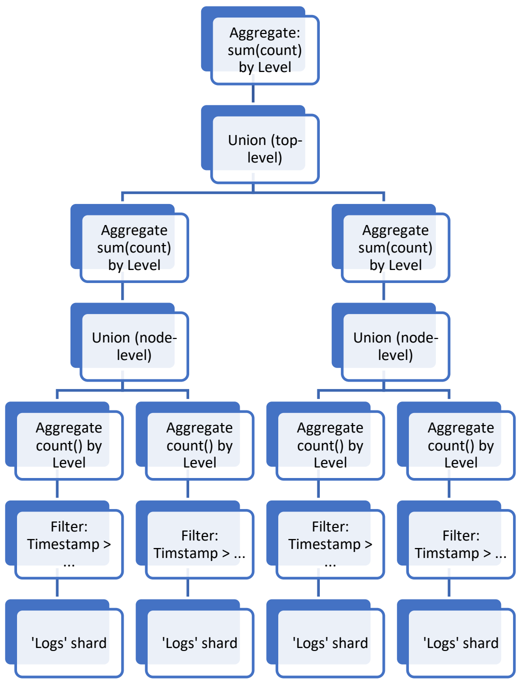
As an example, consider the following query:

```
Logs | where Timestamp > ago(1h) | summarize count() by Level
```

The initial RelOp tree for this query looks like this:



Distributed RelOp tree:


Aggregate: sum(count) by Level
Union (top-level)
Aggregate sum(count) by Level
Aggregate sum(count) by Level
Union (node-level)
Union (node-level)
Aggregate count() by Level
Aggregate count() by Level
Aggregate count() by Level
Aggregate count() by Level
Filter: Timestamp > ...
Filter: Timstamp > ...
Filter: Timestamp > ...
Filter: Timstamp > ...
'Logs' shard
'Logs' shard
'Logs' shard
'Logs' shard

Since the vast majority of queries deal with the uniformly distributed data shards, the query plan above usually has a lot of symmetry. In order to reduce the analysis time, the optimizer always keeps these RelOp trees in a compact, “collapsed” form, where each union has a single source operator, and maintains a separate list of relevant shards or nodes that parameterize the sub-tree. The rewrite rules, in turn, are designed to work with this compact form.

Once we have a detailed distributed query plan, what remains is to decide where each operator is about to execute (on what DataNode), and what sub-trees of the query plan may run in parallel. A dedicated optimizer pass, guided by constraints and heuristics, makes this decision for each operator in the query plan.

Data movement strategies

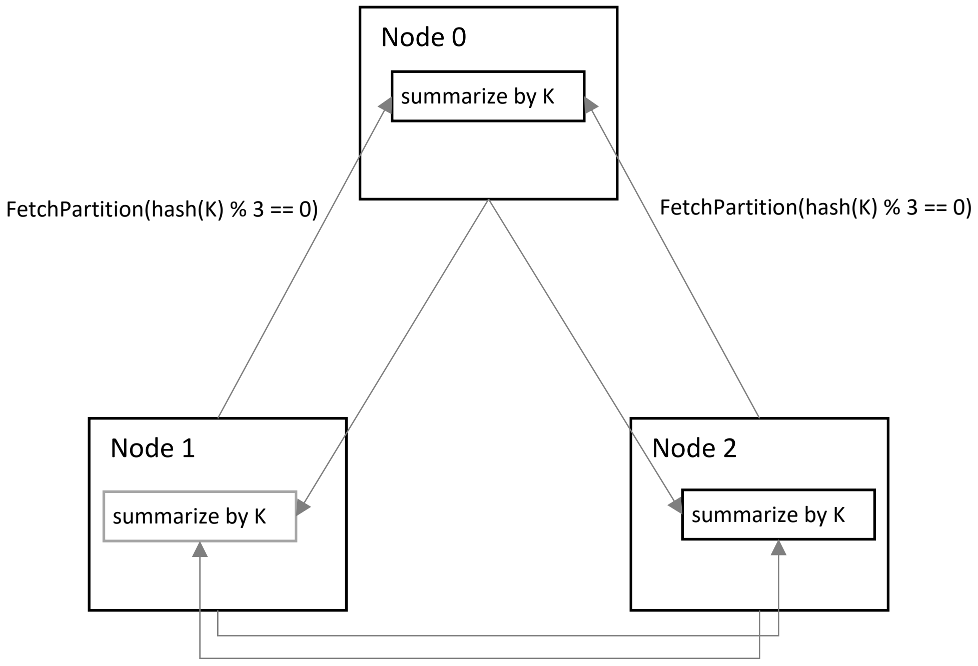
Up until now we’ve focused on the scatter-gather pattern of the distributed query, where we try to run as much of the query logic as close to the data shard as possible, then aggregate the shard-level results at the node level, followed by a top-level aggregation. This strategy works best for the queries that can perform most of the data reduction at the shard level, e.g.:

```
Logs | where Message contains “error” | summarize count() by dayofmonth(Timestamp)
```

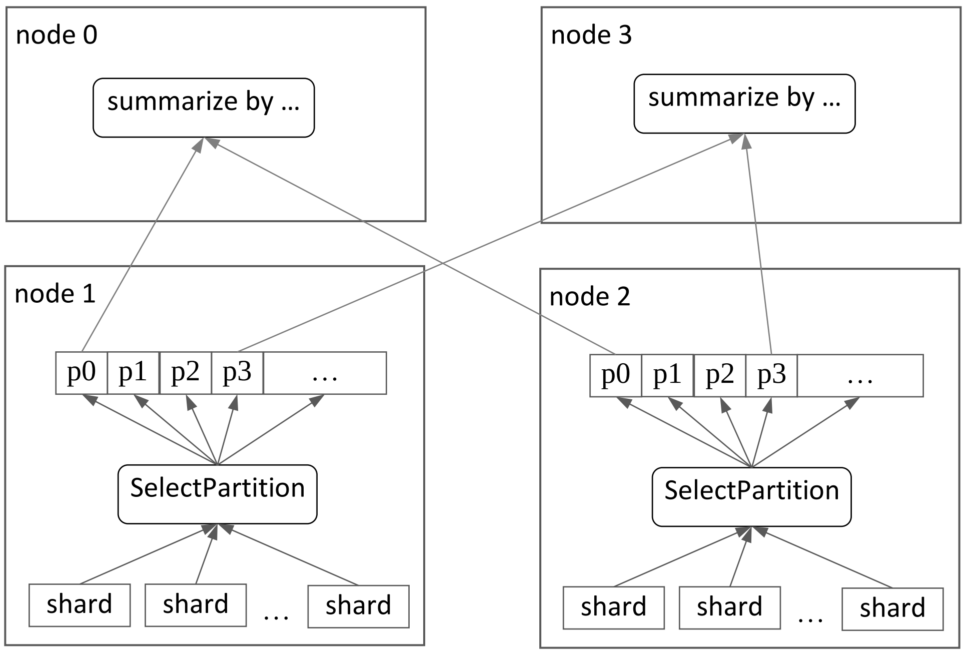
Here, the cardinality of the **summarize** group key is low (day of month), thus most of the computation happens at the shard level (heavy filtering and reduction of potentially billions of records into 31 groups), followed by a very lightweight summarization at the node and the top levels of the query.

However, this is not the best strategy for **join** and **summarize** operators with very high cardinality of the join or group-by keys, respectively. First, the resulting hash tables might easily exceed the per-node query memory budget; second, the node-level and the top-level aggregations become significant sequential bottlenecks in the query execution. In order to overcome these limitations, ADE query planner and execution engine implement the *data shuffling* mechanism, which, combined with the cost-based optimization and the relevant rewrite rules, enables additional execution strategies for the **join** and the **summarize** operators that are more resilient to high-cardinality keys.

The essence of the data shuffling in the Engine service is to partition the source data records into a number of disjoint sets based on the hash of the join/summarize key and ensure that the entire partition ends up on one of the cluster nodes, where it can be processed by the local join/summarize operator implementation, independent of any other partition:


Node 0
summarize by K
FetchPartition(hash(K) % 3 == 0)
FetchPartition(hash(K) % 3 == 0)
Node 1
summarize by K
Node 2
summarize by K

There is no dedicated "shuffle" operator in the query engine. Instead, the engine has a more primitive yet very flexible **SelectPartition** operator. Logically, this operator scans the source record stream and retains only the records with the key that belongs to the requested partition P out of total N partitions (i.e. `hash(K) mod N == P`). The crucial difference between a regular filter operator and **SelectPartition** is that the latter computes all the partitions at once, performing a single pass over the source data and buffering the partitioned records using a FIFO-like cache structure. When one of the nodes X requests its own partition from the node Y, node Y locates the in-progress SelectPartition operator and pumps the buffered partition X from the cache. This process happens in parallel on all cluster nodes:



Note: SelectPartition is also running on node 0 and node 3; we've omitted it from the diagram for clarity.

With the data shuffling mechanism in place, we can now discuss the implementation strategies for the join and summarize operators.

For each appearance of the join operator in the RelOp tree, query optimizer estimates the size (number of records) and the cardinality (number of distinct keys) for the left and the right sides of the join. Based on this estimation, the optimizer decides on one of the three implementation strategies:

- If both join sides are of similar size and the key cardinality is on the order of 1M or less, employ the regular (naïve) join implementation: bring both sides to the same node and perform a non-distributed hash-based join there;
- If one of the join sides has up to 100K records and is significantly smaller than the other side, employ the broadcast join strategy: evaluate the smaller join side first and embed it as a table in the query plan, then distribute (broadcast) this table to each node and perform the hash join as close to the data shard as possible, in parallel on all the relevant cluster nodes and data shards;
- Otherwise, employ a shuffled join: apply the same partitioning scheme on both join sides, bring each partition to its dedicated node and perform a hash join within the partition on that node. By default, the number of partitions is chosen as the number of nodes in the cluster.

Similar considerations apply to the **summarize** operator as well (with the exception of the broadcast strategy, irrelevant for summarize).

Once the data is shuffled, the optimizer will try to build a query plan that keeps as much of the remaining query operators as possible within the newly formed partitions, in order to avoid the unnecessary additional data movement as well as to benefit from these partitions in subsequent operators (e.g. if a join is followed by a summarize with the same or more granular key, that summarize may continue on the same partition formed by the join).

Distributed query execution

The final query plan has all the necessary information to execute the distributed query. The actual query execution is initiated by walking the operator tree and instantiating an *Iterator* for each operator (on demand).

The engine implements a pull-mode iterator model, where each iterator delivers a columnar batch at a time. A typical iterator implementation will instantiate the iterators for its source operators, invoke their Next() method to pull the batches of data, process these batches according to the operator's specification, and deliver the results to the consumer iterator. For instance:

- Filter's iterator Next() implementation will invoke Next() method on its source, apply the predicate to the columnar batch, and return the reduced columnar batch to the consumer;
- Aggregation's iterator Next() will consume the entire source iterator upon its first invocation, while updating the "grid" with the aggregated results, then return the aggregated grid as a single batch to the consumer;
- HashJoin iterator (the underlying implementation of the Join operator) will first instantiate its Build source iterator and prepare the hash table with the join keys. Subsequent invocations of the HashJoin's Next() method will invoke Next() on its Probe source iterator, locate the matching entries for the current columnar batch in the hash table, build the resulting (joined) batch and return it to the consumer.

There are two special types of operators in the query plan to tackle the parallel and the distributed query execution aspects. Both are pass-through from the functionality point of view, but they affect the way that subsequent operators are executed:

- *ParallelSubquery*: executes the query plan sub-trees in parallel;
- *RemoteSubquery*: executes the query plan sub-tree on another DataNode (parameterized by the identity of the remote DataNode).

These operators are injected by the query planner at the appropriate junctions of the query plan tree during the final stage of the distribution pass.

Shard storage

Data shard in ADE represents a horizontal partition of a table, and each shard is implemented as an independent compressed column store.

There are no hard constraints on the size of a single shard: neither on the number of logical records it represents nor on its physical size (size of the shard's encoded data in the Blob store). Nevertheless, the preferred shard size is 100K-10M records, since a large number of small shards results in significant management and query planning overhead, while small number of very large shards inhibits query execution parallelism.

The most common way to create a new data shard is through the *Data Ingest* control command, which encodes and commits a new data shard from the supplied batch of source artifacts.

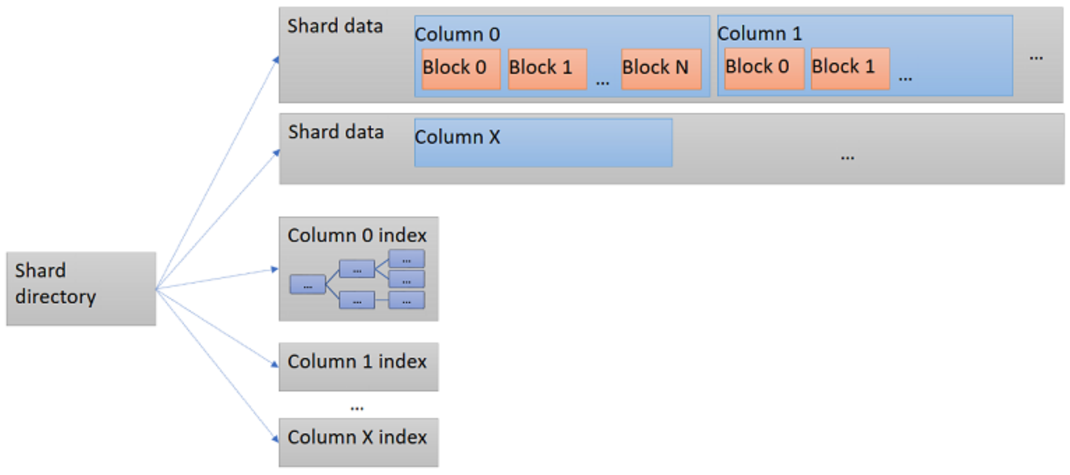
Ingestion and shard format

Data ingestion process maintains a (sliding) buffer of values for each field in the target table. It parses the source record stream (e.g. a CSV file) according to the schema of the target table and appends the parsed field values to the corresponding value buffers. Once the accumulated value buffer for a given field has sufficient size (for instance, 100-1000 values for the fixed-size data types, or 10K-50K bytes for variable-sized data types), the ingestion process attempts to find the most suitable encoding for the accumulated sequence of values.

At the end of this process, each field (column) of the source data is represented by a sequence of encoded (compressed) blocks, along with a *block map* data structure that records both logical and storage ranges for each block. All encoded columns are written to a blob, one after another.

In addition to the data encoding, the ingestion process also maintains an *index builder* component for each data field and records each processed value of a given field with the corresponding index builder. Once the ingestion process has consumed all the data sources and is about to seal the new shard, each index builder spills the encoded index data structures to a dedicated blob.

A sealed data shard is represented by a top-level *shard directory* blob, that references all the artifacts comprising an encoded shard (data blobs and index blobs), as well as stores offsets to major data structures within those blobs:



In general, the shard storage format is designed to avoid any non-trivial parsing and minimize the number of I/O operations when accessing the relevant data objects at query time.

Dynamic data type

Dynamic is a special data type supported by the (shard) storage layer and the query engine, intended for efficient handling of JSON values. While it's perfectly valid to store JSON values as plain *string* data type (and use the dedicated JSON parse/extract functions to access the relevant properties within the query), *dynamic* provides a fine-tuned representation (speeding up property access by an order of magnitude), improved indexing support and a convenient query syntax.

The essence of the dynamic data type is a special binary encoding of the nested values (primitives, arrays and property bags) that retains enough information for efficient access to the property bag values or the array elements. In particular:

- All object encodings start with the type tag
- Strings are length-prefixed
- Property names are dictionary-encoded and replaced by their dictionary token
- Array starts with the list of offsets to each of its elements' encodings
    - There are dedicated encoding schemes for homogeneous numeric and string arrays
- Property bag starts with the list of (name token, offset) pairs for each of its elements' encodings

Thus, deeply nested elements within a dynamic value can be accessed in time that is proportional to the size of the access path, as opposed to the size of the entire object.

Data indexing

By default, every field is indexed during the data ingestion (low-level encoding policy allows one to fine-tune or disable the index for the specific fields). The scope of the index is a single data shard. The implementation of the index depends on the type of the field.

String column index

The engine builds an inverted term index for the string column values. Each string value is analyzed and split into normalized terms, and an ordered list of logical positions (i.e. containing record ordinals) is recorded for each term. The resulting sorted list of terms and their associated positions is stored as an immutable B-tree.

Numeric column index

For the numeric columns (these also include DateTime and TimeSpan columns), the engine builds a simple range-based forward index. The index records the min/max values for each block, for a group of blocks and for the entire column within the data shard.

Dynamic column index

To index a dynamic field, the ingestion process enumerates all "atomic" elements within the dynamic value (property names, values, array elements) and forwards them to the index builder. Other than that, dynamic fields have the same inverted term index as the string fields.

Shard merge and rebuild

As we mentioned earlier, the initial data shards created by the Engine service correspond directly to the ingested batches of raw data. This is not always optimal, since small batches of ingested data result in small data shards, and over time the Engine cluster might end up with a very large number of very small shards. This impacts negatively both the shard maintenance processes within the Engine cluster as well as the query planning time and the query execution performance.

The Engine service implements two operations to reduce the number of data shards: *shard rebuild* and *shard merge*, while the DM service monitors the state of the Engine cluster and orchestrates the background execution of these operations.

Shard rebuild command takes a few small data shards, fully decodes them, creates a single concatenated stream of records and re-encodes it as a new, larger data shard. The motivation for the rebuild is to eliminate very small data shards.

Shard merge command creates a larger data shard by referencing the data artifacts of the smaller shards and rebuilding only the indices (e.g. performing the N-way merge of the inverted indices for the string columns). The motivation for the merge is to increase the scope of the index.

Shard query

As we mentioned in the earlier *Query Analysis* section, the leaves of the query plan tree are the *Shard Access* operators. From the query engine point of view, "shard query" is just a chain of operators adjacent to the bottom-level *Shard Access* in the query plan tree. Usually these operators will be able to take advantage of the data shard format and its features (e.g. the specifics of the data encoding and the indices), beyond just processing a generic stream of columnar blocks.

As an example, let's look at the Filter operator adjacent to a shard access. This operator will execute in two stages:

1. Traverse the predicate expression and collect all simple conditions on the shard's columns (e.g. a string column matching a constant pattern, a numeric column compared with a constant). Use the appropriate index on the column to get the list of candidate positions or blocks. Combine the index results (union/intersection/negation) according to the logical operators in the original expression.

2. Iterate over the candidate position ranges obtained in (1), verify and evaluate the filtering conditions against the actual data where needed, lazily fetch additional column slices.

A more involved example is the Top-N operator that follows a Filter, at the data shard level, e.g.:

```
Logs | where Level == "Error" | top 100 by Timestamp
```

Note that both the Filter and the Top-N in this case are propagated down to the individual shards by the distributed query optimization rules, the results from multiple shards are then merged and the final Top-N operator computes the correct global result. Thus, at the bottom of the distributed query plan there will be a *ShardAccess* operator, followed by the *Filter* operator on the Level column, followed by the *Top*-100 operator on the Timestamp column. The implementation of Top-N in this case will recognize that it's executing against the *ShardAccess*, and will proceed with the dedicated two-pass implementation as follows:

1. Instead of fetching all the shard's columns, access only the Level and the Timestamp.
2. Use the index on the Level column to get the list of candidate positions. Evaluate the filter expression on the Level at the candidate positions and feed the Timestamp's that passed the filter *along with the associated shard positions* to the Top-100 evaluator (which implements a variation of the min-heap data structure).
3. At the end of the first pass, the Top evaluator will accumulate the list of 100 positions within the shard that satisfy the filter and correspond to the top 100 records by Timestamp.
4. Now the Top-N evaluator will access the remaining columns and efficiently fetch only the records from the top 100 shard positions determined earlier.

Row Store and Streaming Ingestion

Shard storage allows for very efficient query but, as explained above, requires data to be ingested in relatively large chunks. Row Store provides an intermediate storage mechanism which allows efficient ingestion of data in small portions (up to several megabytes in size), while making this data immediately available for query. This scenario is called *Streaming Ingestion*. Row Store mechanism is not in effect by default; first, the Engine cluster must be enabled for *Streaming Ingestion* and then specific tables or databases must be assigned a *Streaming Ingestion Policy.* Enabled cluster provides fixed number of Row Stores. Each enabled table is automatically assigned one or more Row Stores along with the unique set of keys.

Row Store is a persistent key-value store. Each value contains one or several rows of the table. Each Row Store will contain data from one or several tables identified by a key. When inserted, a value is assigned an *ordinal* (monotonously increasing number), committed to the *Write-Ahead Log* and then inserted into a key map from where it can be queried.

When a table enabled for *streaming ingestion* is queried*,* the query plan tree will have *the Rowstore Access* operator leaves (possibly in combination with the *Shard Access* operators). *Rowstore Access* operator will simply iterate over the table rows from the Row store, forcing subsequent filters to perform a full scan over those rows.

When the amount of data for the table in Row Store(s) passes an automatically maintained threshold, it is transferred to the *Shard Storage* thus balancing streaming ingestion and query efficiency.

Advanced features

In this section we discuss the implementation of several useful features provided by the Engine service that go beyond the basic data ingestion and querying functionality. All of these rely on the core mechanisms described earlier.

Cross-cluster queries

Each query operates in the context of the current cluster and the current database, but it may also reference databases and tables in external clusters (provided the user has the necessary access permissions on the external clusters).

When a query contains references to external databases/tables, the Query Head node will identify such references during the semantic pass and fetch the minimum necessary metadata from the corresponding remote clusters to analyze the query (e.g. fetch an external table schema). These references become *Remote Table Access* operators in the RelOp tree, along with the local *Table Access* operators.

Unlike the local Table Access operators, remote tables are not expanded into a hierarchy of data shards during the query distribution pass, since the query planner has no knowledge of the remote cluster topology and shard distribution. Other than that, remote tables are subject to the same optimization rules as the local ones (predicate push down, projection propagation, aggregation distribution and many others).

Once the generic optimizations are applied to the RelOp tree, the query planner needs to decide how to execute the query that contains remote table references. Functionally correct but naïve strategy would be to bring the entire remote table data to the local cluster and perform all the computations locally. However, in most cases this is far from optimal, since it would miss the opportunity to offload the computation to the remote cluster (and benefit from the indexes, cache and query parallelism there) as well as to reduce the data transfer costs over the network (possibly across Azure regions). Instead, the query planner identifies the maximum sub-trees of the query plan containing the remote table references that should be executed on the remote clusters. E.g. if the remote table access is followed by *Where* and *Summarize* operators, these should be executed on the remote cluster as well. In general, the decision as to what sub-queries should be delegated to the remote cluster is based both on the feasibility and the cost estimation heuristics.

Each of these RelOp sub-trees is then rendered back into a standalone KQL query and is collapsed into a *Cross-Cluster Query* operator that encapsulates the partial query to be executed on the remote cluster.

Update policy

Update policy is a policy set on a table that instructs ADE to automatically append data to that table (a target) whenever new data is inserted into another (source) table, by running the update policy's query on the data being inserted into the source table. This allows, for example, the creation of a one table as the filtered view of another table, possibly with a different schema, retention policy, etc. In other words, an update policy allows one to set up a simple and efficient ETL-like process within our service, without the need to establish more complex data processing flows with external dependencies.

The implementation extends the regular *Data Ingest* flow described earlier:

1. When a new data batch arrives to the source table, the engine creates a new data shard; this shard is written to the Blob storage and cached locally on one of the cluster nodes, but it's not committed yet to the metadata.
2. Before committing the new shard to the metadata, the Admin iterates over the update policies that depend on the source table and creates a query plan for each update policy query. Instead of running against the entire source table, the scope of these query plans is reduced to a single data shard created in (1).
3. For each query plan built in (2), the Admin initiates an "ingest data from query" flow, that executes the query plan, pumps the results to a separate ingestion process and creates a data shard for the target table.
4. Once the derived data shards are ready, the Admin commits all the data shards (the original one for the source table and the ones derived by the update policies for the target tables) to the metadata.

This feature also supports cascading updates.

Follower clusters

ADE supports the notion of a follower cluster. A follower cluster can follow one, several, or all of another cluster's databases. "Following" a database means that the follower cluster attaches to that database in read-only mode, making it possible to use the follower cluster to run queries on that database without utilizing the resources of the cluster that writes to the database (called *leader cluster*). The follower cluster periodically synchronizes to changes in the followed databases, so it has some data lag with respect to the leader cluster. The lag could vary between a few seconds to a few minutes, depending on the overall size of the followed databases' metadata.

The main motivation for the follower cluster feature is workload isolation. For instance, a lead ADE cluster may continuously ingest a stream of near real-time production events, powering queries from the operational dashboards and the troubleshooting scenarios. A team of data scientists that wishes to run occasional experiments on the production data may decide to stand up a follower cluster (on demand and possibly for a limited time) and run the heavier resource-intensive analytical queries on the read-only view of the same data, without interfering with the activity on the leader cluster.

The implementation of the follower cluster mechanism takes advantage of the metadata and shard data storage design, that allows an independent Engine service instance to load the latest snapshot of the relevant database and re-create full and consistent view of that database within its own cluster, and then update it periodically from the latest snapshot.

ADE in production

As of September 2018, our technology is used extensively by hundreds of teams within Microsoft, providing them a comprehensive telemetry analytics solution.

It is also used as an analytics data platform in several external products.

ADE is deployed in 41 Azure regions as 2800 Engine+DM cluster pairs, on a total of about 23000 VMs.

The overall data size stored in Kusto and available for query is 210 petabytes, with 6 petabytes ingested daily.

The engine clusters process around 10 billion queries per month.