Gartner

Licensed for Distribution

This research note is restricted to the personal use of Can Huynh (can.huynh@loto-quebec.com).

Selecting the Optimal Technical Architecture for Data Ingestion

Published 1 November 2019 - ID G00390192 - 40 min read

By Analysts Dk Mukherjee

Initiatives:Data Management Solutions for Technical Professionals

The demand to capture data and handle high-velocity message streams from heterogenous data sources is increasing. Data and analytics technical professionals must adopt a data ingestion framework that is extensible, automated and adaptable. This research details a modern approach to data ingestion.

Overview

Key Findings

- Existing batch mode data integration architectures are not meeting current data latency requirements, thus leading to late-arriving information.
- The need for in-flight analytics has made it necessary to marry event-driven raw data with historical and other application (CRM, ERP etc.) data prior to persistence.
- There is growing demand for a data ingestion framework that supports dynamic tagging of data by leveraging sophisticated ontologies to help fingerprint the data and identify sensitive data or anomalies.

Recommendations

Data and analytics technical professionals tasked with creating data management solutions must:

- Adopt a versatile ingestion framework that can seamlessly confront the volatility of source data availability, format inconsistency and the volatility of data flow management.
- Invoke ontology services to classify incoming nonobvious data.

- Apply data hygiene before data is persisted to facilitate reliable “in-flight analytics.”
- Leverage a diverse set of connectors to complement ingestion with continuous enrichment from other sources.
- Adopt batch ingestion models for large data volumes. Utilize a streaming ingestion model for streams of continually arriving data.

Analysis

In today’s digital era, the rate of data generation is growing exponentially. As data volumes rise with a concomitant uptick in velocity, existing data ingestion engines face unprecedented strain in terms of scalability, performance and storage efficiency. Furthermore, diversity in source systems has made it unwieldy to accurately classify and ingest data.

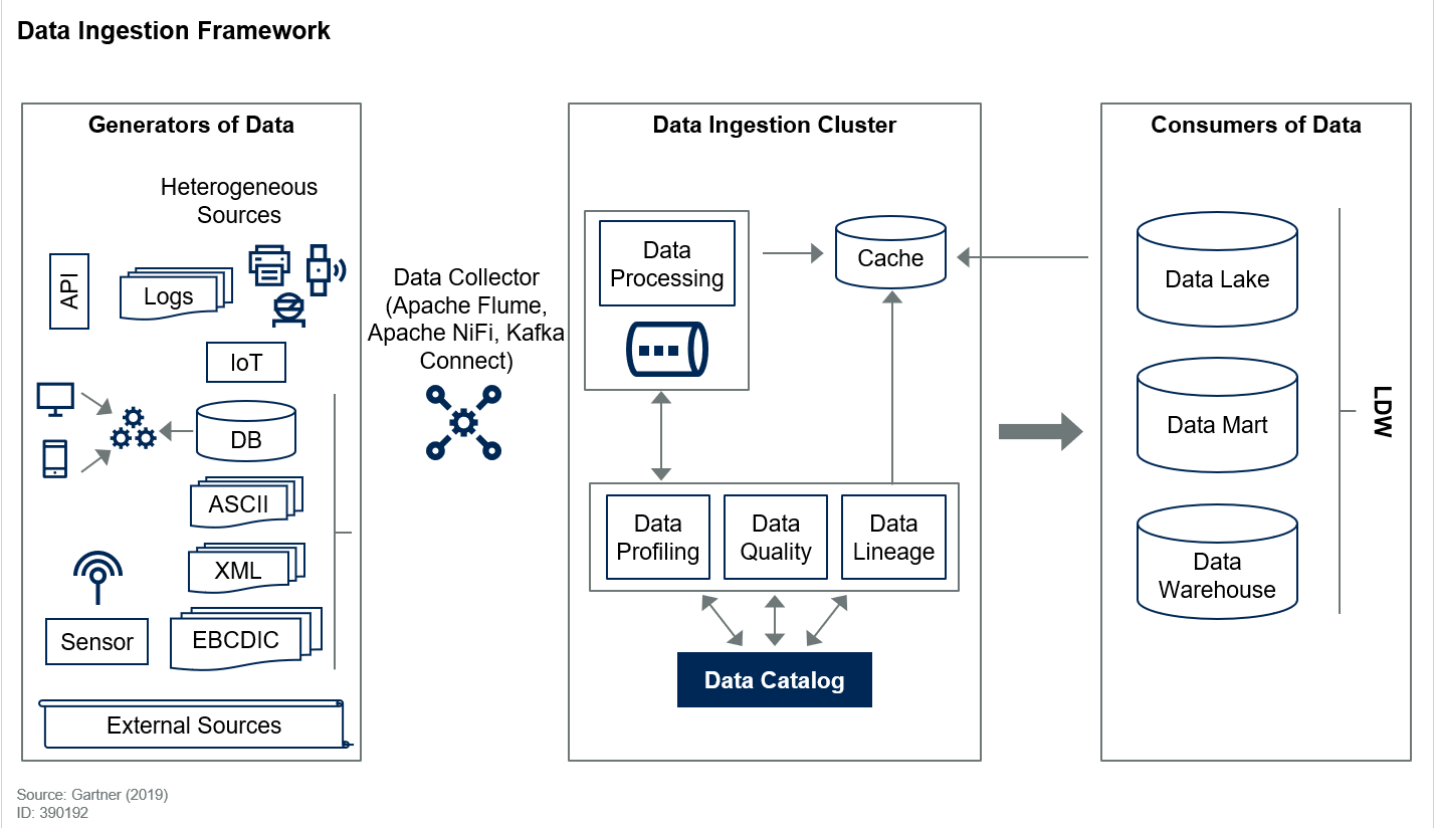
Business initiatives such as Internet of Things (IoT), cloud and analytics, which require near instant translation of data to information, are fundamentally changing data ingestion requirements. Traditional, batch-driven approaches to data ingestion may not adequately cater to modern data handling requirements owing to high latency and inability to handle schema drift.

Data and analytics technical professionals must understand and embrace a modern approach to data ingestion. This research note catalogs challenges, identifies features and requirements, and recommends an architectural approach to building a versatile data ingestion framework. It also presents a guidance framework that helps technical professionals to arrive at an optimal data ingestion approach.

Data Ingestion Defined

In its simplest form, data ingestion is the process of collecting data from data sources (generators of data) and moving it to target systems (consumers of data) where it can be stored and analyzed. Data, however, emanates from many different source systems and comes in many different formats. Data ingestion technologies must therefore connect to a panoply of data sources and collect, parse, cleanse, validate, calculate, organize, aggregate and transform data before it can be delivered to downstream target systems. Figure 1 illustrates a modern data ingestion framework.

Figure 1. Data Ingestion Framework



Data ingestion employs two processing models: batch ingestion and stream ingestion.

Batch Ingestion

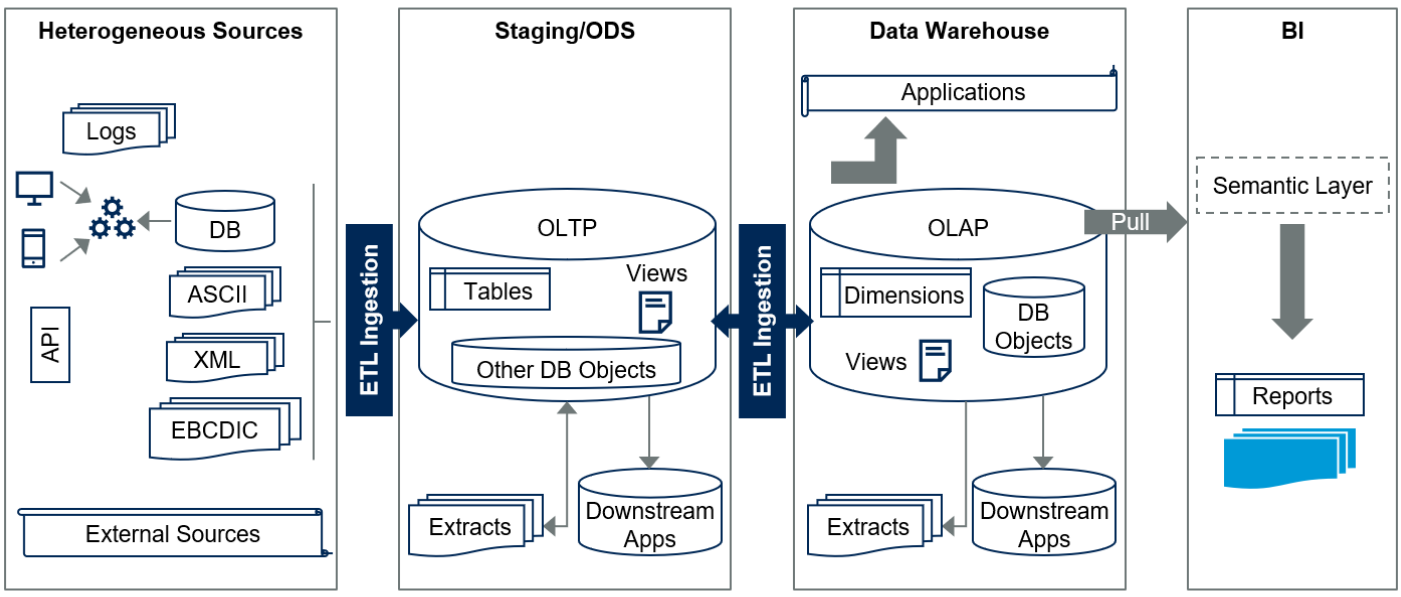
Batch processing has been the traditional approach for ingesting data for the last 40-odd years.

The data transport mechanism shown below is referred to as “batch data movement” or “batch jobs” because data at rest is ingested and processed in batches. These jobs are executed at certain times of the day/night when computing resources are available. A well-checkpointed orchestration workflow retrieves a file/log (among a potpourri of heterogeneous sources) from a folder in an FTP server of the source system or a batch of records from an RDBMS as examples. An operational data store (ODS) serves as the first ingestion platform for that data. Sometimes, shell scripts (Python, etc.) are leveraged to parse the file to look for expected patterns to establish conformance to predetermined standards before any operation is conducted on the data.

Figure 2 demonstrates standard batch data movement processes.

Figure 2. Standard Batch Data Ingestion

Standard Batch Data Ingestion



Source: Gartner
ID: 390192

Typical incoming data is denormalized/flattened prior to ingestion into the ODS. Then, in the same pipeline, extensive transformations are conducted on the data to clean, standardize, link and master the data before it is loaded into the warehouse. The Inmon data warehouse (DW) often adheres to a third normal form to:

- Avoid redundancy
- Facilitate joins among disparate subject areas

The Kimball dimensional data marts that source data from the DW then serve as the foundational source for the business intelligence (BI) semantic layer.

Stream Ingestion

Batch processing has long been the de facto standard for ingesting data. Traditional data ingestion paradigms accepted the inherent latency of extract, transform, load (ETL)-based batch processing. However, modern use cases demand real-time data processing that promotes mission-critical business decisions.

The data sources layer (Figure 1) comprises an assortment of data types from disparate source systems and devices that arrive at nondeterministic rates, sizes and/or schemas. The data collector either “pulls” data from the sources or has data “pushed” to it. Common data collectors are Apache Flume, Apache NiFi and Kafka Connect.

Continuous data is also seen as a stream or river of events. Examples such as emissions from gas sensors, transactions from retail point of sale systems, stock data from the trading floor and actions

performed on a website to purchase/return goods abound.

A construct ("data collector") is needed that detects these events and pulls or encounters a push via a set of connectors.

The data collector typically executes data flow pipelines by connecting heterogeneous batch and streaming data sources with the processing layer. Data collector pipelines ought to be schema-agnostic and adapt to schema drift. Drift is seen when, for example the field name varies between two subsequent JSON records or when the number of fields varies between two subsequent JSON records.

The data collector ought to have the ability to transform flat files into streaming data. It needs to scale with the number of heterogeneous sources to avoid loss of information or posing any resistance to processing.

For example, Kafka Connect has its own dedicated cluster where on each machine Kafka Connect instantiates Java processes called "Workers." These Workers are responsible for transferring data to the Kafka topics (see "Building Data Lakes Successfully" for more details).

In Kafka, the incoming data constitutes key:value pairs with a byte array. The Connect API serves as an abstraction layer for downstream brokers to read the data. The schema of the incoming data is typically housed in an AVRO format to help validate extractions.

NiFi has the added advantage of accommodating messages with a wide range of sizes. The size of data is constrained only by the heap size allocated to a particular NiFi instance. In cases where the size of the incoming data is greater than the heap size, it helps to split the data into multiple AVRO files to avoid "OutOfMemory" errors.

However, it cannot replicate data like Kafka. Flume is constrained in a similar manner. Kafka can replicate data to partitions in slave nodes.

If a node fails, one has to wait for it to recover for the data to be retrieved. Although NiFi is recognized as a reliable ingestion engine for large-volume data flow configurations, it is not recommended for intricate computations and event processing (CEP). The presence of a "cache," typically a file or a database in the ingestion cluster layer, augments the flexibility of the ingestion framework. This intermediate data store, devoid of constraints, is used (among other things) to:

- Buffer data
- House a business rule engine (BRE)
- House reference data

At varying intervals, this data is persisted to the permanent DW.

Modern Use Cases Drive Change to Data Ingestion Requirements

The following use cases are driving change to the data ingestion discipline:

- **Analytics:** Adoption of analytics continues to increase as organizations seek to build the "intelligent enterprise." Demand for real-time analysis and intelligence requires that data be ingested from multiple sources that manifest themselves in different formats and at different frequencies.
- **IoT:** The number of things such as devices, sensors and cameras connected to the internet continues to grow — and at an unprecedented pace. As the number of connected IoT devices grows, so too does the amount of data generated by these devices. With data arriving on the order of seconds or less, the onus is on the ingestion engine to accept and route the data in a manner that makes downstream processing of the data seamless.
- **Customer data:** Understanding your customers and their buying demands is a key competitive differentiator for organizations. To attain a 360-degree view of the customer, it is imperative to accommodate and link data from disparate sources. Data ingestion systems must provide a versatile set of data connectors to access data in the source systems in real time.
- **Cloud:** In an age of cloud computing, enterprise data is dispersed across various cloud domains. There is now a need for "near instant" access to data from different domains and regions. After this data has been uploaded to the cloud, there must be a data access framework that has the bandwidth to ingest a large volume of data and then decompose it into microbatches for convenient consumption by downstream CRM/ERP/BI and other applications.
- **Applications:** In the world of CRM, for example, there is an increasing need to transcribe speech to text. The generated audio stream is fed to a data ingestion system that employs natural language processing (NLP) to:
  - Classify the incoming data
  - Determine the appropriate fields in the target data store that need to be updated to initiate an action

Data Ingestion Features

The modern use cases detailed in the previous section present challenges to data and analytics technical professionals, as highlighted in Table 1.

Table 1: Data Ingestion Features

Data Ingestion Features ↓	Description ↓
Data ingestion frequency	Orchestration and workflow handling of the ingestion should be tied to the frequency. Ingestion processes should be deployed with the right scale to ensure each ingestion batch is properly sized for throughput.
Data format	Understanding the type of data that is being ingested is important because not all tools can support all types of data formats. Some tools, for example, cannot support ingested binary data like video and audio, and some tools can work only with structured datasets. Therefore, responding to heterogeneity in a manner that does not ignore essential data is imperative.
Hardware/software failure	The ingestion pipeline is expected to revert to its original state in the event of a failure. For example, if failure occurs between a staging layer and the warehouse, the load should be able to reinstantiate from the point of failure.
Unexpected source data schema	When data from a hitherto nonspecific schema arrives, the ingestion engine will need to adapt and ingest the data.
In-flight analytics	In use cases where cogent response to adverse events can only materialize if the ingestion framework is agile enough to furnish information before data is persisted in downstream systems, it is important for the ingestion engine to make its pipeline queryable.
Data change rate	The ingestion approach needs to display agility to identify and accommodate changes in source system data. The downstream layer must be nimble enough to employ change data capture (CDC) algorithms on the data to capture the changes in data and populate appropriate data structures on time.
Data quality	The veracity of the data will need to be validated for completeness, conformance to standards, accuracy, validity and uniqueness. An intermediate data structure to house the business rule engine (BRE) can be leveraged to institute checks on incoming data before it is persisted.
Data privacy	The ingestion engine must make accommodations to either anonymize or pseudonymize the sensitive data (or personally identifiable information [PII]) as it arrives.

Data Ingestion Features	Description
Data locations	Determine whether the data ingest will happen on-premises, from on-premises to cloud, from cloud to cloud, or from cloud to on-premises. Understanding the origin and destination for the data to be ingested is important for choosing the right toolsets, designing the right network bandwidth and architecting failure-handling mechanisms.
Throughput	Understanding how data is generated, transported or accessed offers clues to how to set up, configure and plan capacity of the ingestion framework. Cluster and scalability of the ingestion tool should also be guided by the above requirements. For example, to accommodate spikes in data, the ingestion cluster must launch new nodes/brokers to ingest data and relieve stress on the existing nodes.
Velocity	When faced with fast-arriving data, the ingestion engine will need to respond with agility to transfer the data to an available node/broker.
Volume	Sometimes, the size of the payload can test the response of an ingestion engine. The engine will need to have the ability to buffer the data adequately and perhaps have the downstream processor parse and create minibatches.
Data security	Data needs to be encrypted in transit and at rest. Secure transmission, either via SSH or SSL, needs to be assured in the network.

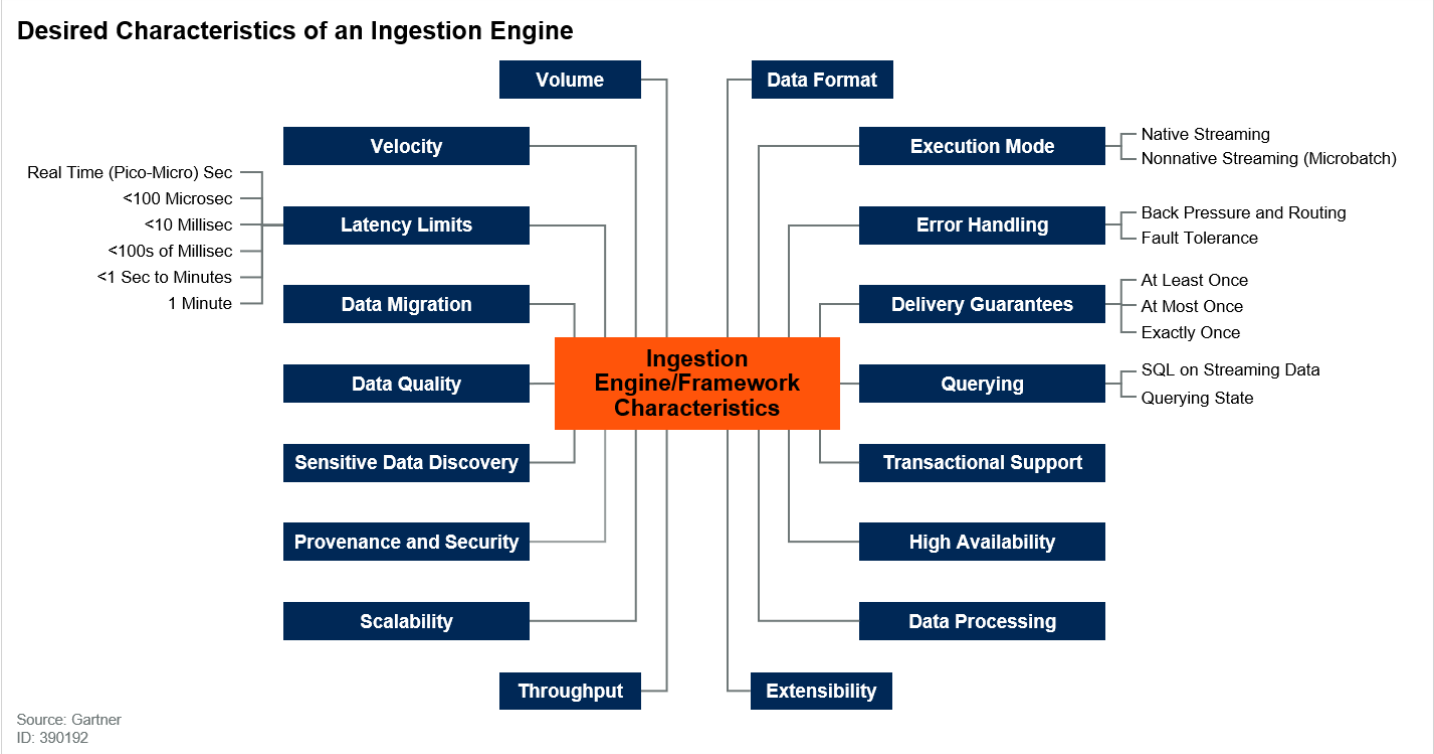
Source: Gartner

Data Ingestion Framework Requirements

The features detailed above are driving the requirements for a modern-day data ingestion engine.

The attributes of an ingestion engine needed to meet the challenges posed in modern use cases mentioned earlier are encapsulated in Figure 3. These attributes serve to equip an architect with factors that she/he would need to consider while deciding on an optimal approach for an ingestion framework.

Figure 3. Desired Characteristics of an Ingestion Engine



The Data Format

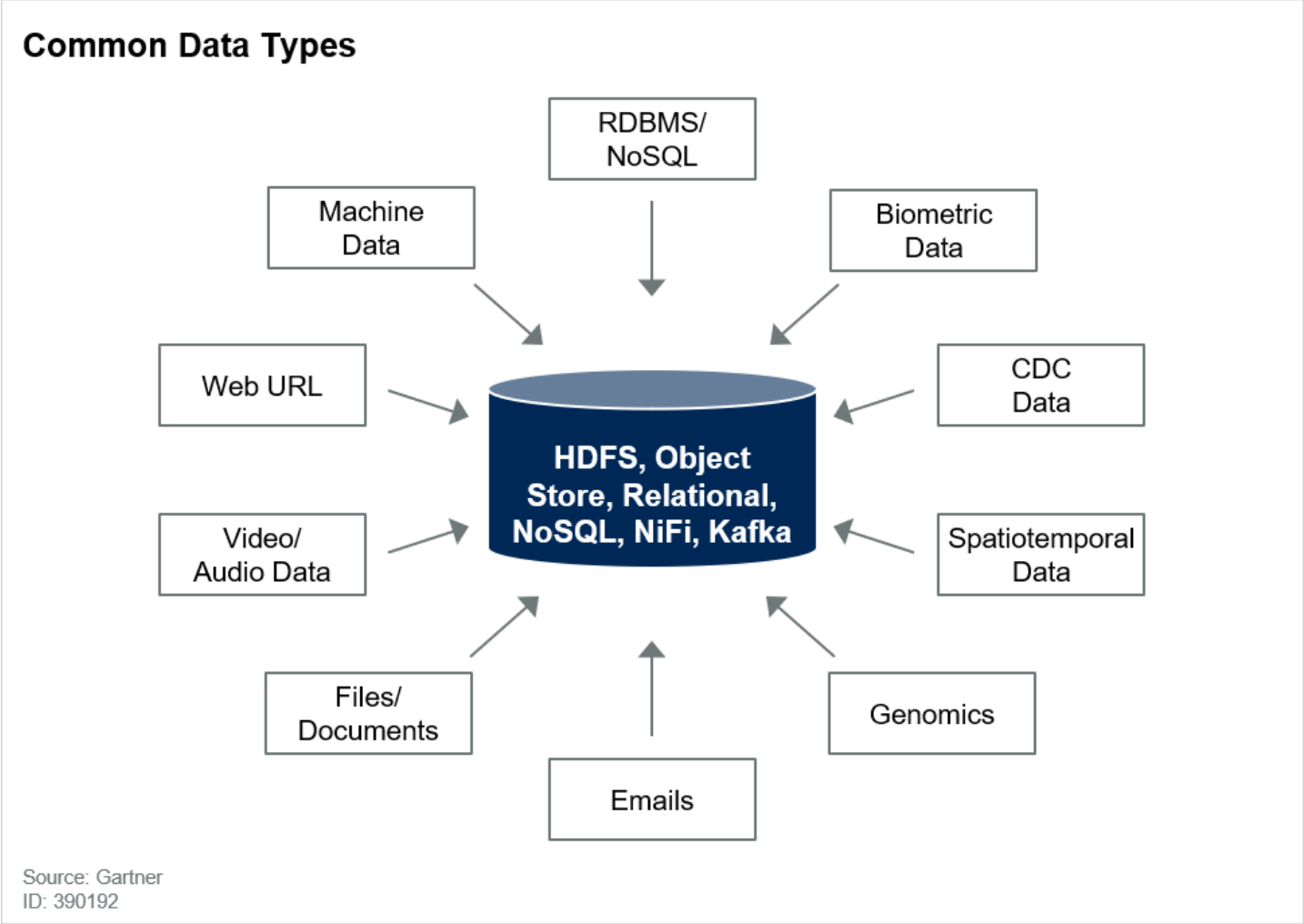
The variable that merits extensive study is the format of the data required to be ingested.

Data is generated and arrives in one of the following formats:

- **Structured:** Structured data is formatted such that relationships between attributes can be inferred easily. Moreover, data is provided with reliable metadata.
- **Unstructured:** Unstructured data does not conform to any predetermined format or schema and requires extensive processing to extract meaningful information from.
- **Semistructured data:** Semistructured data appears in formats where the schema is open to evolution. However, the presence of metadata allows for an interrogation that can serve up useful information in a short period of time.

Figure 4 shows some commonly ingested types of data.

Figure 4. Common Data Types



Structured Data

Data in flat files, especially in .csv and .txt formats, is quite simply represented by a cell that resides at the intersection of a field and a record. Flat files are easily portable and can be compressed easily, and their contents can be viewed with any free editor.

There are two distinct types of text files:

- **Delimited:** Each field is separated by a character or tab. Common forms are .csv and .tsv files.
- **Fixed width:** Each field has a designated field length.

For both types of files, file encoding needs to be selected. UTF-8 is more pervasive than the others such as US-ASCII or Windows-1252. An ingestion pipeline would need to conduct tasks such as:

- Read file name
- Show the content of the file to conduct a cursory check (with UNIX Shell commands) on structure of the file
- Institute error handling such that records seen to violate constraints are returned to the data owner

- Incorporate filtering such that only the most recent records are ingested

Time-Stamped Data

Records bearing a time stamp help identify either the time that an event transpired or the time when the event was entered in the file or database.

Open Data

Open data can be republished without restrictions from copyright, patents or other mechanisms of oversight. For example, the open data portal ("Socrata Homepage," Socrata) houses criminal activity data for numerous cities in the U.S. The goal is to promote data accessibility and data literacy.

Virtualized Data

Data virtualization (DV) attempts to perform data cleansing, data transformation and data correlation as data moves from production source systems while avoiding any intermediate storage. Virtualized data presents a unified view of data from multiple sources, thus obviating the need for laborious ETL tasks.

Database-Based Extraction: Relational Database Management System (RDBMS)

Relational databases such as Oracle and SQL Server have long served as faithful data repositories for relational data that consists of intersections of rows and columns such as with the parts being cataloged in a supply store in Figure 5.

Figure 5. Sample Relational Data

Sample Relational Data

Results | Messages

	SN	ITEM_NUM	ITEM_DESC	MFR_MODEL_NUM	UNSPSC_CD
1	SS37899B-F	5ECA6	1/4 Hex Bit Holder, 1/4 Hex Shank, Overall Bit Length: 2-3/8	48-32-4502	27112815
2	GS37778B-F	4BCA6	1/2 Hex Bit Holder, 1/2 Hex Shank, Overall Bit Length: 2-	49-32-4502	27112816
3	SS37898B-F	5ECA6	1/4 Hex Bit Holder, 1/4 Hex Shank, Overall Bit Length: 2-3/8	48-32-4500	27112815

Source: Gartner
ID: 390192

ERP and CRM Systems: ERP Challenges

ERP systems such as Microsoft Dynamics 365, SAP, Abas and JD Edwards EnterpriseOne house an enormous number of tables and adopt an unwieldy level of normalization. In most cases, those tables have nonintuitive names, thus making reliance on the vendor-furnished abstraction layer (such as a GUI) unavoidable.

CDC Data

With source data residing in voluminous databases, there is considerable duress on ETL processes to extract and store this data in the staging layer. This is exacerbated with the need to extract this data on subsequent days, in which case increased storage and delayed downstream processing often results in a suboptimal data journey.

However, if source records that underwent a change can be identified, extracted and loaded into the staging layer, the time for processing and loading can be shortened considerably. This “change” would need to be categorized as a “new record or insert” or a “changed/modified record” or a record that was “deleted/removed” from the source system.

There are essentially two distinct methods of extracting information from changes: intrusive and nonintrusive. The first leads to an impact on the source system that encounters some semblance of parsing the source data to identify changes.

Source-Data-Based CDC

The ETL tool needs to recognize the attributes in the source system whose changes over time must be reflected in the downstream data warehouse:

- **Direct read based on time stamps:** If the source data records date-time values — an update time stamp and an insert time stamp — it is tractable to deduce “what” changed and “when” it changed.
- **Database sequences:** Tables employ an autoincremented surrogate key to identify records to make it easier to identify “new” versus “old” records.

Both instances require extra work on the database end in terms of maintaining extra data structures.

What are the challenges seen with the aforementioned approaches?

- If the source system record has only an “insert time stamp,” can the date/time for the change in event be deduced? What if there is an additional “update time stamp”?
- How can we infer that a record in the source system has been deleted? Would it make it easier for the ETL tool to implement CDC logic if the source record displayed a logical delete?
- Is there a way to extract how many times a record has been updated between two load dates?

A history of changes made to the source tables is often desired. One option to save the changes (inserts/updates/deletes) to a table is triggers. Another is to write complex stored procedures.

The triggers place the changed records in intermediate tables that the ETL tool can read from and transfer to the DW, thus capturing all the changes. However, considering their noticeable impact on the performance of source systems, triggers are usually discouraged. Although triggers provide the

functionality of capturing near-real-time changes, the stress (the inherent intrusiveness) they impose on source systems renders them unusable.

An alternative is replication. The changes in the source systems are merely replicated to stage tables in the warehouse. This requires minimal intrusion because the changes are extracted from the source system log files.

Snapshot-Based CDC

When source data does not expose usable time stamps or triggers, and replication is not an option, then snapshots of tables are acquired at various points in time, and a comparison is conducted to capture the changes.

Some of the standard approaches to detecting differences are:

- A full outer join and tag of the resulting records for insert, update or delete.
- Standard ETL tools conduct a “merge” and flag records for inserts, updates and deletes.
- The snapshot approach imposes a cost related to storage. However, its advantages over the approaches discussed thus far cannot be ignored.

One area that merits some thought is the performance degradation when the tables that are being compared are large. This is where a database engine performs better than a middle-tier ETL engine.

Log-Based CDC

The least intrusive approach in extracting changes to source data is to query the transaction logs that house all inserts, updates and deletes. There is an attendant cost involved in translating the binary log into something that downstream systems can comprehend and act on. Each database has its proprietary tool that helps make the log queryable.

Table 2 provides a succinct compilation of the approaches discussed thus far and the attendant strengths/weaknesses.

Table 2: CDC Options

↓	CDC Options ↓			
Benefits	DB Time Stamps	Triggers	Snapshot	Transaction Log
Insert/Update Distinction	No	Yes	Yes	Yes

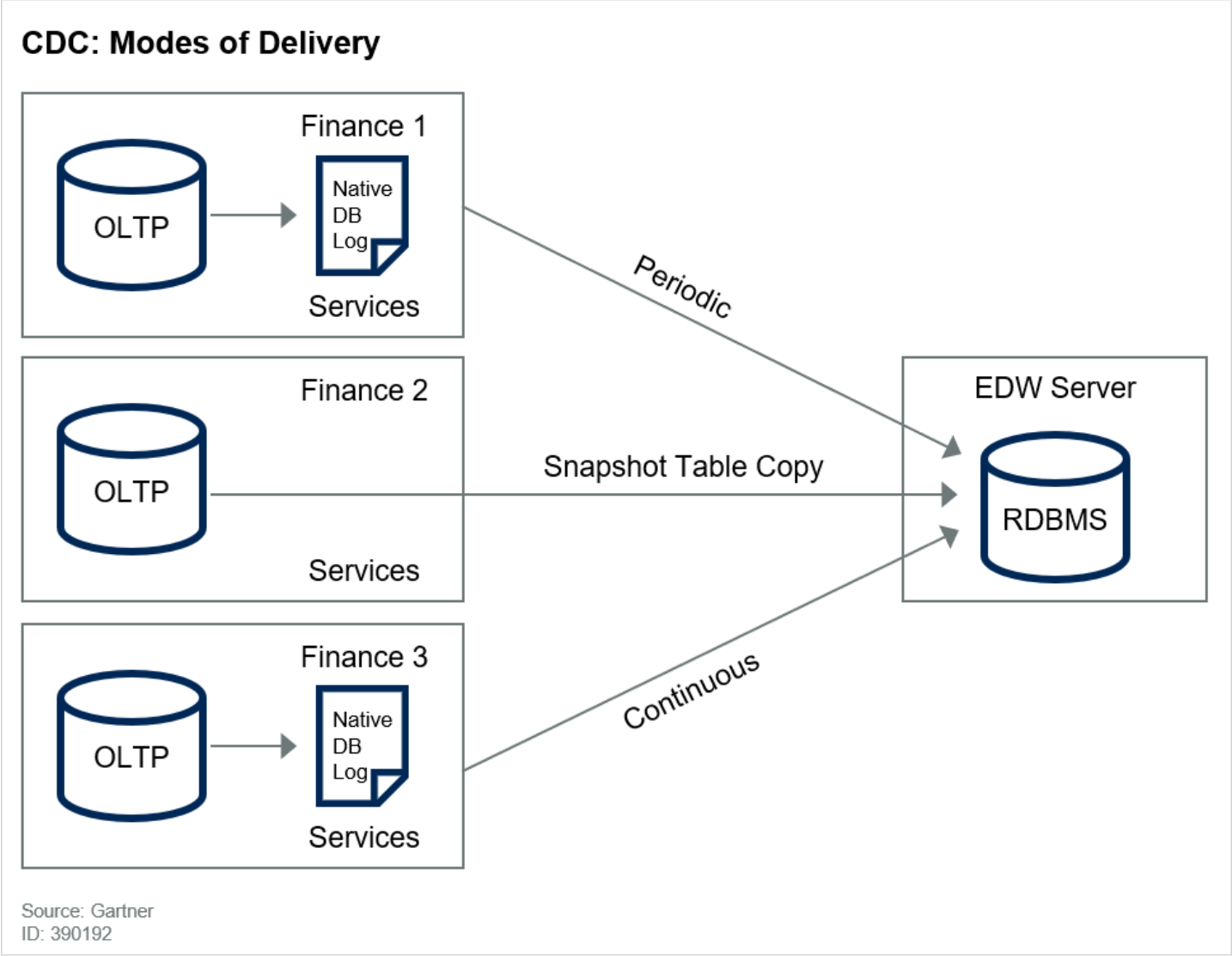
	CDC Options			
Multiple Updates Detected	No	Yes	No	Yes
Deletes Identified	No	Yes	Yes	Yes
Real-Time Rendition	No	Yes	No	Yes
DBMS-Independent	Yes	No	Yes	No

Source: Gartner

Figure 6 shows a view of the CDC process that emphasizes how “changed” data is transferred to the target enterprise data warehouse (EDW). In this case, the EDW server merges financial data from multiple branches into a single corporate instance.

Three common modes dictate the replication frequency. They generally vary from continuous (near-real time) to snapshot to periodic. Table-level refresh (snapshot) or copy can be used in addition to log-based CDC.

Figure 6. CDC: Modes of Delivery



Semistructured Data

Examples are XML and JSON.

XML is an open standard for presenting structure and content of data in plain text. XML is a meta language that forms the basis of formats like RSS or Atom. Typically, systems exchange data in an XML format.

Following are ways an XML file needs to be processed:

- Verify that the file is well-formed: Check if the nesting structure is balanced.
- Validate the content with a data type definition (DTD) or XML Schema Definition (XSD) file.

NoSQL: Increasingly, data stores such as Apache Hadoop, Couchbase and PostGreSQL are storing data in a document format that makes it conducive for search and extraction, such as in the JSON format below.

```
{
  "product_id":"72008788",
  "name":"Apple Watch",
  "price":"372.99",
  "attributes":{          // object begins
    "color":"black",
    "size":"44mm",
    "type":"wrist watch",
    "ounces":"172"
  }                       // object ends
}
```

Here, an object is a set of unordered “name:value” pairs. There are query tools that allow usage of SQL to query JSON data.

Web-Based Extraction

- **Text-based web extraction:** Download a file placed on a website. In many instances, reference data is made available as tab delimited or comma separated files.
- **HTTP client (XML and JSON data):** Utilize UNIX scripts to “scrape” websites.

Using SOAP

SOAP forms the foundation layer of a web services protocol stack. It uses XML as its message format. The principal challenge in ingesting an XML file is in deciphering the nested structure of the file. The workload that this process entails compels an ingestion engine to process the data in batches. Typically, scripting (using Python, etc.) is used to parse the data and decompose it into microbatches so that latency is not affected adversely.

Unstructured Data

Examples are emails, a doctor’s dictation and audio/video files.

Every industry has seen the need to parse the unstructured data encountered in documents, emails, images and so forth. The data embedded in documents has long been sought with a view to uncover

information that either adds to existing knowledge or serves to amend existing knowledge.

To extract important insights from documents, a subset of the methods typically used is:

- **Tokenization:** Split text into multiple sentences to aid processing one sentence at a time. Then split a sentence into words.
- **Part-of-speech tagging:** Part-of-speech tagging serves to categorize words as nouns, pronouns, verbs adjectives and so on.
- **Ontology look-up:** Existence of open-source ontologies and taxonomies allows a look-up to help contextualize the data.
- **Entity extraction:** To faithfully model data in unstructured documents, entity extraction is employed to help categorize data into dimensions.

Genomics Data

The data profile is typically a compendium of profiles of thousands of patients. Genomics data is categorized in four distinct ways:

- **Sequence:** For example, the nucleotide sequence of a chromosome
- Annotations: Descriptions of features — such as genes and transcripts — that appear in genomes or transcripts
- **Quantitative data:** Any kind of numerical value associated with a chromosomal position
- Read alignments: A record matching a short sequence of DNA to a region of identical or similar sequence in a genome

Biometric Data

Measurements related to human characteristics allow a person to be identified based on a set of verifiably unique data that are specific to them.

Data Spanning Multiple Structures

Certain data types are a conglomeration of various strains of the data fabric.

Machine Data

Data often associated with:

- Personal computers, smartphones and other devices
- Servers and networks

- Websites: Data could be semistructured
- Applications and programs
- Security information and event management (SIEM) logs
- Financial transaction records

Spatiotemporal Data

This can describe point locations or more complex lines such as vehicle trajectories or polygons (plane figures) that make up geographic objects like countries, roads, lakes or building footprints. Temporal sequences of position records (x, y, t) are increasingly being collected in rapidly growing amounts due to the development of tracking technologies such as GPS, RFID, Wi-Fi, mobile phones, banking transactions and sensor networks.

Volume

With an uptick in the volume of source data, it is increasingly becoming important for the ingestion engine to accommodate large datasets. The processing layer is expected to decompose the generated data into microbatches to adhere to throughput requirements.

Velocity

With messages/data being delivered at an ever-increasing velocity, the ingestion engine will need to adopt a subscription approach that allows for timely dissemination of the data.

Following are some desirable traits that will allow the ingestion engine to accomplish its goals:

- Low-latency message delivery
- Batch data and compression
- Horizontal scaling

Querying

A SQL engine compatible with the ingestion pipeline needs to be incorporated to enable real-time data processing. It must be scalable, elastic and fault-tolerant, and it must support operations such as filtering, transformations, aggregations, joins and windowing.

Latency Limits

With data arriving on the order of seconds or less, the onus is on the ingestion engine to accept and route the data in a manner that makes downstream processing of the data seamless. For example, data from gas sensors needs to be ingested and processed in a matter of milliseconds. Otherwise,

the stale data is as good as nonactionable data. On the other hand, large payload data such as banking XML data can be ingested in batches and routed as microbatches.

Failures and out-of-order sequence issues arising from ingestion can add to the latency.

Data Migration

Given a mosaic of data sources and both on-premises and cloud targets, the ingestion engine will need to ensure seamless data flow. The migration could entail:

- Cloud to cloud (e.g., AWS to Microsoft Azure)
- On-premises to cloud (e.g., on-premises Oracle to Amazon Redshift)
- Files or RDBMS to Hadoop Distributed File System (HDFS)
- CRM (e.g., Salesforce) to Azure SQL Database or Apache Kafka to SQL Server

There would be an added requirement to ingest incremental changes in source data in addition to the initial load.

Data Quality

Sometimes, the incoming data does not conform to agreed-upon standards and schema. The ingestion engine will need to conduct rudimentary but necessary checks to filter out corrupt/redundant data so that only actionable data is persisted in a downstream data store.

An AVRO/JSON file can house the data/schema drift rules and associated alerts. The following alerts are most commonly seen:

- Field name varies between two subsequent JSON records
- Number of fields varies between two subsequent JSON records
- Data type of specified field changes or specified field is missing
- Order of fields varies between two subsequent JSON records
- String is empty

Figure 7 shows how UNIX commands, Extensible Stylesheet Language (XSL) or Python can also be leveraged to parse XML documents to validate structure and content. Usage of convenient scripting languages becomes critical when low-latency ingestion performance is desired.

The figure shows sample XML input data and the UNIX commands used to parse the data to extract meaningful information.

Figure 7. Sample XML Data

Sample XML Data

```
<?xml version="1.0"?>
- <flureport title="" LegendLabel6="Widespread" Legend6="B39874" LegendLabel5="Regional" Legend5="F0BB37" LegendLabel4="Local"
Legend4="EBE236" LegendLabel3="Sporadic" Legend3="Sporadic" LegendLabel2="No Activity" Legend2="No Activity" LegendLabel1="No Report"
Legend1="No Report" timePeriod="Week" defaultColor="FEE391" subtitle="Week Ending April 20, 2019- Week 16">
  - <timeperiod subtitle="Week Ending January 05, 2019- Week 1" year="2019" number="1">
    - <state>
        <abbrev>AL</abbrev>
        <color>Widespread</color>
        <label>Widespread</label>
      </state>
    + <state>
    + <state>
    + <state>
    + <state>
    + <state>
    + <state>
    + <state>
    - <state>
        <abbrev>DC</abbrev>
        <color>Sporadic</color>
        <label>Sporadic</label>
      </state>
    - <state>
        <abbrev>FL</abbrev>
        <color>Widespread</color>
        <label>Widespread</label>
      </state>
    - <state>
        <abbrev>GA</abbrev>
        <color>Regional</color>
        <label>Regional</label>
      </state>
    - <state>
        <abbrev>HI</abbrev>
        <color>Local Activity</color>
        <label>Local Activity</label>
      </state>
```

Source: "Weekly Flu Report," Centers for Disease Control and Prevention, http://www.cdc.gov/flu/weekly/flureport.xml.
ID: 390192

To parse out existing reporting levels, technical professionals can examine the contents of the "<label>" element for the entire XML document.

In UNIX:

1. Use **curl** and then **grep** for lines containing "**label**"

1. **Cut** to extract content between ">" and "<"

2. **Sort** with the "-u" flag to remove duplicate results

The expected output ought to be:

- Local activity
- No activity

- No report
- Regional
- Sporadic
- Widespread

Similarly, XSL can be applied to streaming data.

Additionally, Python modules can be leveraged to parse XML and JSON data.

Data Discovery/Profiling

To establish the veracity of the data and promote data literacy for downstream systems, the stream processor is required to characterize the data. This entails compiling essential statistics on the nature of the data that resides in the source systems. The acquired information and associated inferences have a direct bearing on the design of downstream DWs and marts. Furthermore, profiling exercises enable technical professionals to infer business rules that can complement those furnished by data stewards of source systems in the form of data dictionaries.

For example, a study of a financial transaction might:

- Unearth anomalies such as an Anglicized name that appears in some other language.
- Expose a high percentage of NULL records.
- Expose a date in an unexpected format: “yyyy-mm-dd” instead of “yyyymmdd.”

Fundamental types of profiling activities are as follows:

- **Column profile:** Reports on statistics per field in the dataset.
- **Dependency profile:** Reports on dependencies among fields in the dataset.
- **Join profile**: Reports on dependencies among different tables.

The column profile comprises:

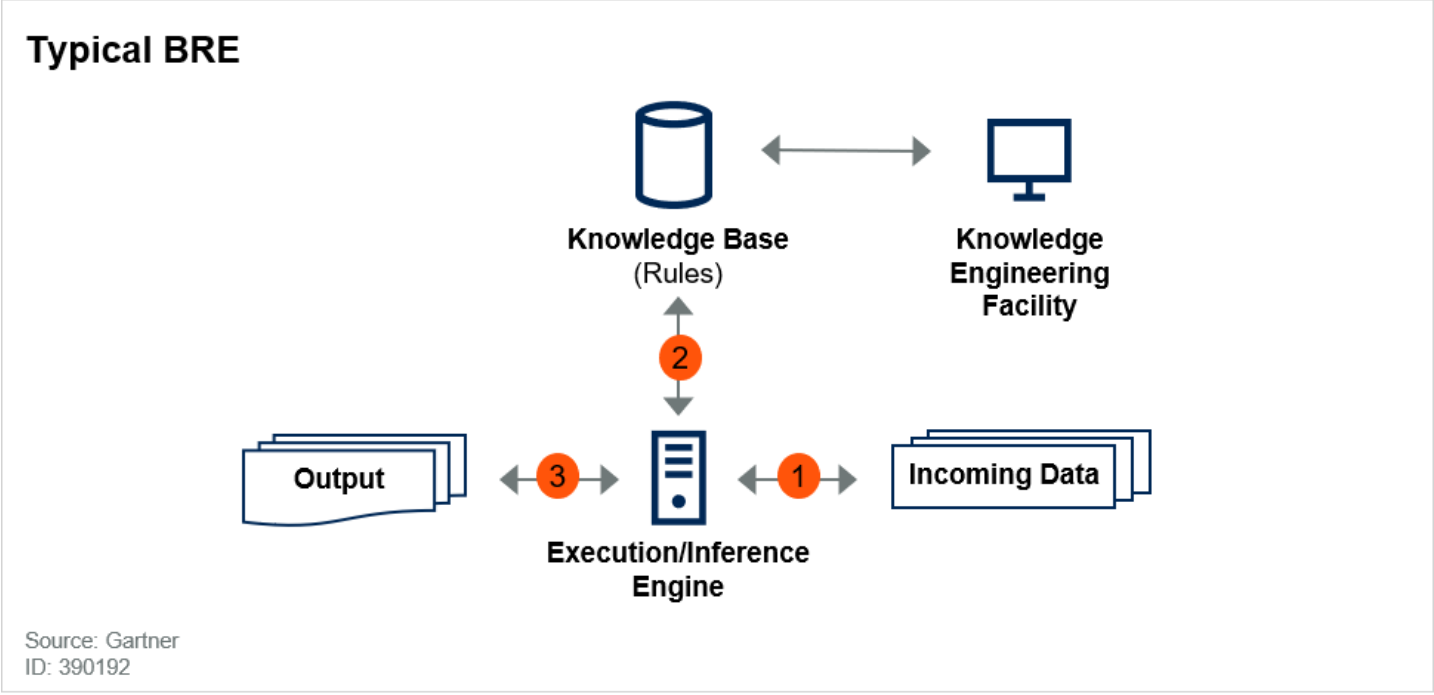
- **Number of distinct values:** Reports on the count of unique values in a field.
- **Number of NULL values:** Reports on count of records where the cell value is NULL.
- **Highest and lowest values:** Reports on range of data.

- **String pattern and length:** Reports on expected string values (e.g., Social Security numbers [SSNs] are expected to be nine characters in length).
- **Frequency counts:** Reports on top and bottom "N" items in a field.
- **Pattern finder:** Reports on a cryptically named field to see if it conforms to patterns such as ZIP Codes or phone numbers.
- **Dictionary matcher:** Reports on comparison between the field under study and an external dataset such as Dun and Bradstreet data.
- **Value distribution:** Reports on frequency of each value that could be in the top or bottom "N" values.

Business Rule Engine

Data profiling is a necessary precursor to the design of a reliable and scalable business rule engine (BRE). At its core, a BRE encapsulates a recipe for ensuring that the fundamental meaning of the data is not susceptible to "drift." It does so by subjecting the source data to a series of interrogations before it can be considered ready for ingestion by the DW (see Figure 8).

Figure 8. Typical BRE



Any BRE that serves to preserve the sanctity of the source data, per the data dictionary that circumscribes the source system, needs to be equipped with the following pillars:

- **Attribute constraints:** Rules that dictate unique values of each field/attribute

- **Relational integrity rules:** Rules that serve to link different subject areas in a dataset and are extracted from data models
- **Historical data rules:** Rules that guide an ETL/data architect to make necessary accommodations for domain-specific legacy data in so far as how it conflicts with current domain values (e.g., a diagnosis dimension table that replaces International Classification of Diseases [ICD] 9 diagnosis codes with ICD10 diagnosis codes)
- **Dependency rules:** Rules that serve to ensure consistency across attributes of an entity (e.g., a rule that ensures that a ZIP Code is consistent with a county or city).

Finally, a library of rules that govern life cycle of processes needs to be included as well. For example, for a product to be considered "ready to be shipped," it needs to be tagged with a purchase order and an invoice. This also helps to catalog a product's lineage.

Conforming Data

To ensure that a construct can read data from an assortment of heterogeneous sources and then unify the data to conform to the standards consumers expect, a look-up against a repository of reference values needs to be conducted.

For example, a conformed product dimension is the enterprise's agreed-upon master list of products, including all attributes:

- Processes of conforming:
  - Standardizing: Process of transforming data taken from different sources and various formats into a consistent format
  - Matching and deduplication: Process to identify, relate and merge records corresponding to the same entity stored in same or different databases
  - Surviving: Process to select the best elements from multiple records to survive consolidation (ideal for the creation of golden records for a single product and/or customer view)

How can we design a system that minimizes/precludes manual labor when it comes to tagging data to encourage convenient data exploration?

In other words, can we design an approach that "harvests" metadata instead of manually creating it?

Imagine if the RDFS/XML serialization (ontology service) in Figure 9 was exposed as a web service.

Figure 9. Sample Ontology service

Sample Ontology Service

```
<rdf:RDF xml:lang="en"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

<!-- Note: this RDF schema would typically be used in RDF instance data
     by referencing it with an XML namespace declaration, for example
     xmlns:xyz="http://www.w3.org/2000/03/example/vehicles#".  This allows
     us to use abbreviations such as xyz:MotorVehicle to refer
     unambiguously to the RDF class 'MotorVehicle'. -->

<rdf:Description ID="MotorVehicle">
 <rdf:type resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
 <rdfs:subClassOf
  rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
</rdf:Description>

<rdf:Description ID="PassengerVehicle">
 <rdf:type resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
 <rdfs:subClassOf rdf:resource="#MotorVehicle"/>
</rdf:Description>

<rdf:Description ID="Truck">
 <rdf:type resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
 <rdfs:subClassOf rdf:resource="#MotorVehicle"/>
</rdf:Description>

<rdf:Description ID="Van">
 <rdf:type resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
 <rdfs:subClassOf rdf:resource="#MotorVehicle"/>
</rdf:Description>

<rdf:Description ID="MiniVan">
 <rdf:type resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
 <rdfs:subClassOf rdf:resource="#Van"/>
 <rdfs:subClassOf rdf:resource="#PassengerVehicle"/>
</rdf:Description>

</rdf:RDF>
```

Source: Gartner
ID: 390192

The ingestion tool (Kafka/NiFi) in the case of ~100,000 files per day, could make a call to this ontology service for a detected key:value pair of “1002:Van” (as an example) and have a classification of “MotorVehicle” returned.

This tag could then be saved in the downstream systems for search and reporting; this approach assumes significance especially when schema of incoming data is not known in advance.

A robust and versatile approach that satisfies the aforementioned requirements has the potential of answering unanticipated questions. For example, smartphone data can help map the movements of the owner, which can help infer their residence and consumption behavior.

Sensitive Data Discovery

Data profiling as explained above will help delineate attributes that need special care while propagating to downstream systems. The ingestion engine will need to compare the arriving data with a locally housed business rule engine to ascertain the level of sensitivity of the data.

Provenance and Security

For the purposes of capturing lineage of records in the ingestion system as data traverses across the mosaic of data artifacts (including in-motion and at-rest data repositories, Kafka topics, applications, reports and dashboards, and ML models), a versatile metadata harvesting instrumentation approach is required.

For example, the following JSON output serves to expose the data source of a particular dashboard.


```
{
  "annotations": {
    "list": [
      {
        "$$hashKey": "object:84",
        "builtIn": 1,
        "datasource": "-- Grafana --",
        "enable": true,
        "hide": true,
        "iconColor": "rgba(0, 211, 255, 1)",
        "name": "Annotations & Alerts",
        "type": "dashboard"
      }
    ]
  }
}
```


If the data that is drawn from sources is accompanied with metadata that suggests appropriate security needs to be imposed, then data transferred over the wire needs to be encrypted automatically. Sensitive data should be masked or obfuscated before being ported to a target. Access to services across networks is governed by an authentication protocol such as Kerberos. Kerberos provides an avenue to verify identities of users and/or servers using cryptography.

Scalability

The solution needs to:

- Expand its resources to accommodate multiple producers with varying data arrival rates

- Accommodate varying sizes of input data
- Demonstrate elasticity via scaling in/out/up/down

Throughput

In response to an in increase in demand, the solution needs to parallelize the ingestion task among multiple threads while executing the data flow. There will also be a requirement to compress the data in order to relieve stress on the network.

Extensibility

The solution needs to:

- Demonstrate the ability to work with a variety of data sources
- Incorporate the “plug-and-play model” for the data pipelines to add/remove consumers seamlessly

Execution Mode

The ingestion engine needs to have the ability to route incoming stream of events as they arrive while adhering to throughput requirements. In addition, the engine needs to decompose large datasets into microbatches in order to relieve stress on the consumers.

Error Handling

Back Pressure and Routing

With fast-arriving data from the data generators, the onus is on the consumers to digest the data at the same rate. However, the subscriber/consumer is often left to handle the mismatch when consumption is seen to not keep up with arrival of data.

When back pressure is applied on a stream:

- Data can be cached until remaining data is consumed
- Redundant data can be discarded
- The producers can be notified when to create and push the new data

This approach is also designed to minimize data loss.

Fault Tolerance/Failover Support

Some vendors offer a clustered architecture. The platform checks for node failures and can detect application or service failures. Services that deploy applications typically run on a node. Once that is down, the service needs to restart somewhere else in the cluster.

The distributed architecture with automated failover saves major effort by avoiding manual configurations and provides reliable software for 24/7 applications.

To minimize data loss, a checkpointing mechanism is needed to keep track of all the events that have been processed. In the event of a failure, the last known good state is retrieved. A temporary data store (shown as “Cache” in Figure 1), or Kafka as the persisted messaging layer, can be used to access data streams over an immediate past. ETL architects can then rewind to a specific point and resume ingestion.

Check Ingestion Status

The ingestion framework will need to include the ability to query the “cache” (as shown in Figure 1) to monitor ingestion activity. For every event message, the time stamp (among others) will need to be recorded to ascertain whether there is a delay or failure in the ingestion process. Any error messages will also need to be noted.

This file load metadata in the cache will need to be tracked during the ingestion process to ensure duplicate data is not loaded.

Delivery Guarantee

The ingestion engine will need to furnish a guarantee that:

- Messages will be read in the order they were generated.
- Messages will not be lost.

Transaction Support

Transaction support for various file formats as targets or conduits to targets such as RDBMS (AVRO for most databases and Apache Parquet for those with a columnar data storage design such as Amazon Redshift) will be an integral feature of the ingestion engine.

High Availability

The ingestion engine will need to support clustering for availability in the event of failure. The ability to have alternate nodes available for execution without manual intervention is key.

Data Processing

The ingestion framework will need to:

- Design for failure: That is, test a connection, check for empty folders, compare files/folders, set timeout on FTP/SSH connections and so on
- Detect changed data (CDC)

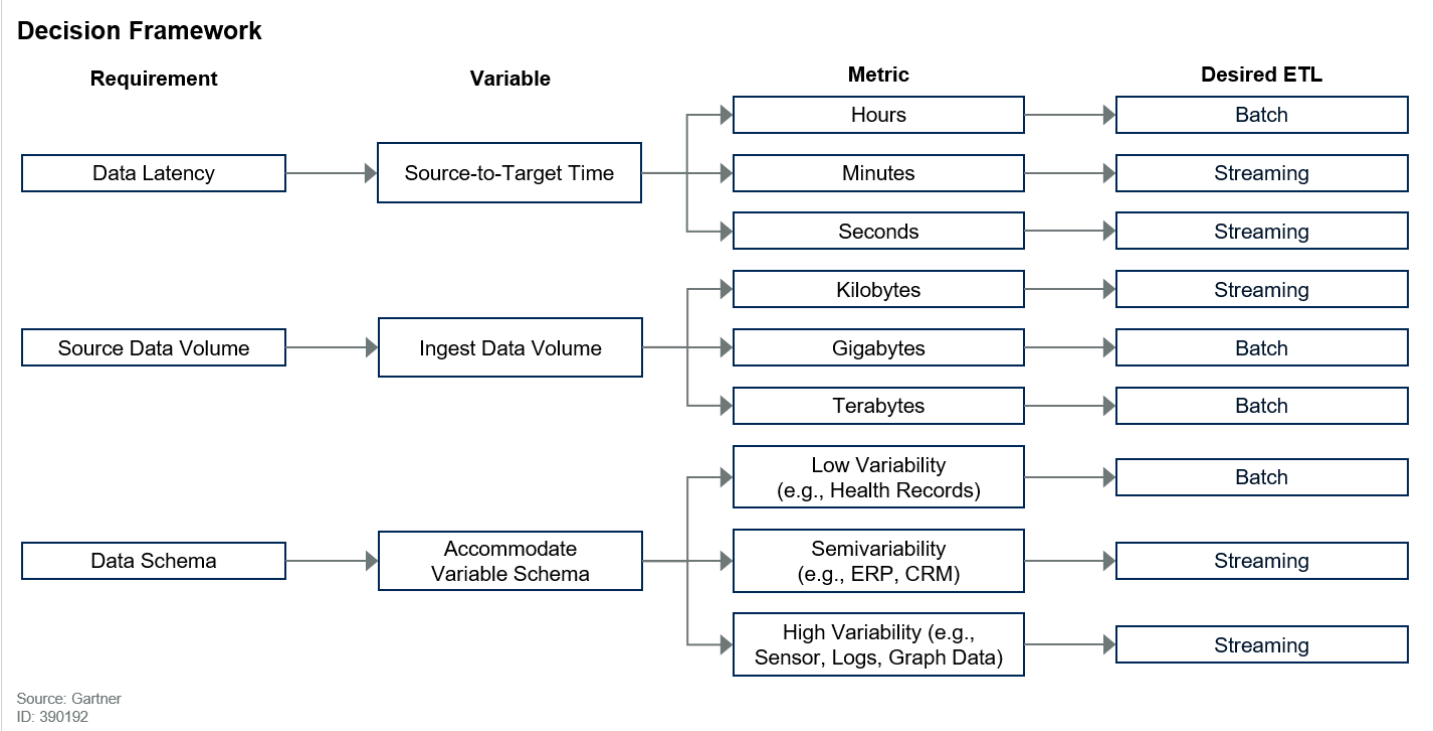
- Check for data quality
- Profile data during ingestion, especially for an obscure source schema
- Validate characteristics of data to ensure conformance to requirements/standards
- Employ scalability (parallelism, partitioning and clustering)
- Conduct “in-flight” transformations on the data
- Implement logging and auditing. Will it need to have “SQL-like” support to query the data as it makes its way across the streaming, execution and processing layers?
- Design orchestration: That is, the framework must allow technical professionals to build a data pipeline that can be used to access data, transform and deliver it at scale, expeditiously transfer the results to cloud repositories, and (conversely) move and process data that was previously locked up in on-premises data silos. The design is expected to handle resource availability, manage intertask dependencies, retry transient failures or timeouts in individual tasks, and create a failure notification system.
- Incorporate data flow monitoring, debugging support, source control, GUI ease of use and so on
- Establish compatibility with common programming languages such as Python, shell script and so on

Decision Framework

When faced with the decision to choose an ingestion approach that caters to specific environments, it would help to consider 3 principal requirements and associated variables — data latency, source data volume and variable data schema.

The guidance framework in Figure 10 serves to bring to fore the various variables, their ramifications, and finally the suggested approach.

Figure 10. Decision Framework



When considering accommodations to cater to low latency and variable source data schemas, streaming ingestion surfaces as the obvious choice. However, when one is studying options for source data size, one needs to decompose the variable further:

- Ingest data volume — Batch ingestion is designed to handle large data volumes while streaming is optimal for small data volumes.
- Increase throughput — Streaming ingestion tools have the ability to increase concurrent threads dynamically to increase throughput if data is seen to buffer beyond a certain threshold for a particular task/thread. For batch tools, additional parallel workflows would need to be built; however dynamic resource allocation is a not a feature that batch ingestion is seen to support.

Furthermore, it is critical to analyze the ramifications when one encounters conflicting metrics. For example, if the latency is seen to be on the order of hours and the data source schema is displaying instances of variability, ingestion engines that can recognize and adapt will need to be selected.

Strengths

Streaming ingestion approaches have introduced competencies in areas that were long seen to create “friction” in data interfaces.

Following are some of the benefits that large data deployments are seen to be accruing:

- Low-latency ingestion with an accompanying ability to conduct and share analytics before data is persisted: The ability to blend incoming data with data from remote CRM/ERP and other applications with a view to enrich the data is a significant advantage in the streaming paradigm.

- Dynamic data flow management is provided for both structured and unstructured data: This provides the ability to institute fault management in order to minimize data loss.
- Dynamic resource allocation to ingestion and dissemination components in order to relieve “back pressure”: This is especially true in cases where the volume and/or velocity exceed certain thresholds.
- In the absence of schema knowledge, message brokers are equipped to make remote calls to services to help categorize incoming “schemaless” data.
- Ingestion can be suspended if the quality of incoming data is seen to not conform to preagreed thresholds.

Because streaming ingestion is designed for rivulets or streams of data, batch ingestion continues to be the choice when it comes to ingesting large batches that are of the order of gigabytes to terabytes of data.

Weaknesses

- When required to design a workflow that calls for JOIN of datasets and associated data transformations, streaming ingestion processors have not matured to the same level as existing ETL tools.
- Most large deployments have identified the absence of a versatile GUI-powered orchestration for data interrogation and cleansing as sources of “data flow friction.”

Guidance

This research has focused on alternate approaches to current ETL frameworks that accommodate batch data. Because existing data ingestion approaches are not agile enough to respond to fast-arriving data in disparate formats, an alternate approach has been suggested.

Organizations that are tasked with handling data transfer from source to target in a manner that caters to low data latency, high fault tolerance and reliable in-flight analytics should adopt a streaming data ingestion framework.

When considering accommodations to cater to low-latency and variable-source data schemas, streaming ingestion surfaces as the obvious choice. However, when studying options for source data size, technical professionals need to decompose the variable further:

- Ingest data volume: Batch ingestion is designed to handle large data volumes, whereas streaming is optimal for small data volumes.

- Increase throughput: Streaming ingestion tools have the ability to increase concurrent threads dynamically to increase throughput if data is seen to buffer beyond a certain threshold for a particular task/thread. For batch tools, additional parallel workflows would need to be built. However, dynamic resource allocation is a not a feature that batch ingestion generally supports.

For example, if the latency is seen to be on the order of hours and the data source schema is displaying instances of variability, ingestion engines that can recognize and adapt will need to be selected.

The Details

In a world of burgeoning interconnectedness, there is a growing need to exchange operational data and extract actionable analytics in the order of seconds. Operational data from sources such as application logs and metrics, event data, and information from microservices applications and third parties constitute data that is typically inserted and updated frequently. Some examples are as follows:

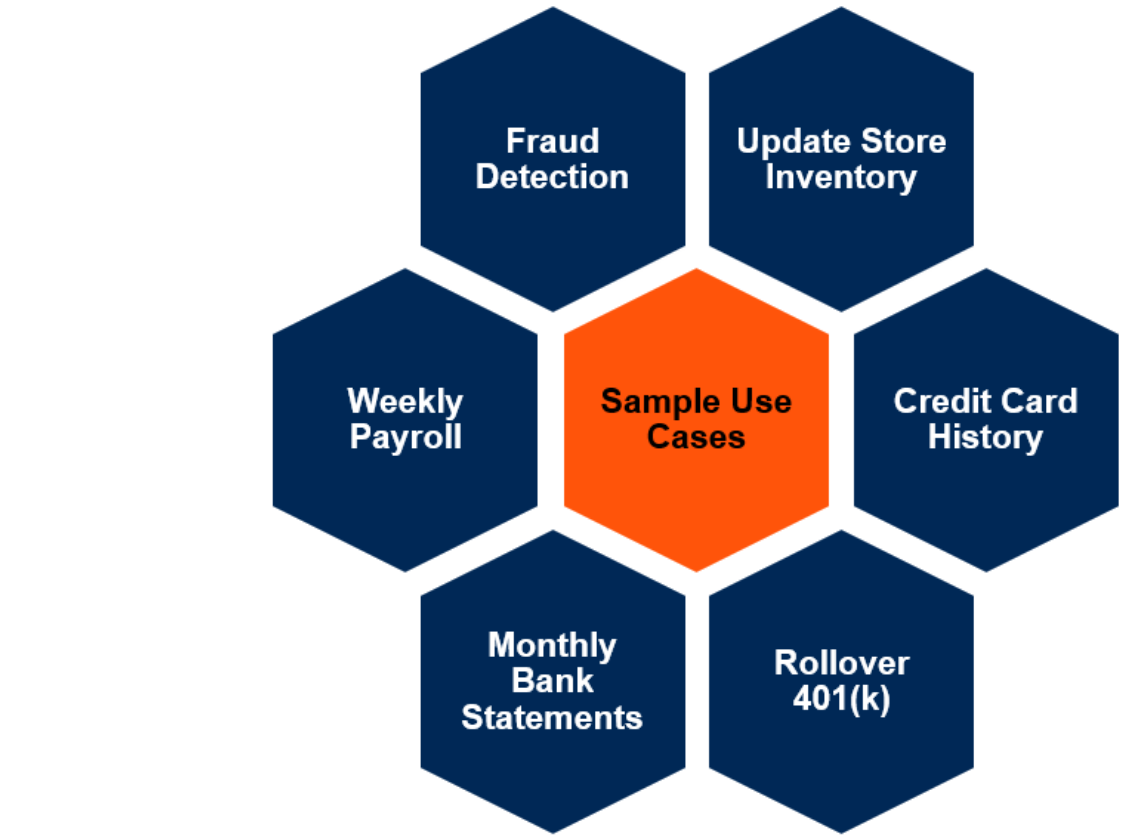
- Instrument data from a network of air quality sensors distributed throughout a locality
- Machine data from a network of solar panels collected throughout the day
- Weather data collected from dispersed weather stations
- Heart monitors that transmit signals to urgent care centers to ensure prompt care in times of distress
- CRM applications that capture accounts, contacts, opportunities, log cases and so on and are validated with data from a master data management (MDM) hub
- Tracking of actions on a website
- Tracking of transactions in banks

Consider the following use cases of data movement where an *agile* and *responsive* ingestion engine plays a role:

- Stream data from CRM application such as Salesforce and a sales online transaction processing (OLTP) database into a data lake.
- Stream data from an IoT sensor attached to a patient's heart monitor to an ER facility.
- Stream data from large healthcare plans to a centralized data lake. Then use a search tool like Splunk to detect anomalies/fraud.

A few more examples, shown in Figure 11 will dictate requirements of an ingestion framework.

Figure 11. Sample Use Cases



For example, a company's "weekly payroll" process is likely to run once a week with large batch files with minimal demands on throughput and latency.

On the other hand, store inventory would need to be updated within a few milliseconds of a sale of a product.

Can Batch ETLs Avert a Catastrophe?

Consider a typical interconnected system that provides heating and cooling to individual floors or other areas. For example, a heat pump extracts heat from air or water for heating. In a *water source heat pump,* a pipe carries water through the structure to supply the heat pump.

Real-time monitoring solutions might use sensor data to detect high temperature spikes. This data could be used to dynamically alter temperature of water to avoid catastrophic failure.

Real-time data can help with everything from deploying emergency resources in a road crash to helping traffic flow more smoothly during a citywide event. Real-time data can also provide a better link between consumers and brands to allow the most relevant offers to be delivered at precise moments based on location and preferences.

Real-time data flow entails an unbounded stream of input data. Latency is typically of milliseconds or seconds. This data typically assumes an unstructured or semistructured format, such as JSON. To support near-real-time consumption, the time between ingestion of data and storage in the analytics data store is expected to be of the order of milliseconds.

The principal objective of streaming data ingestion is to reduce latency of data that is arriving at higher-than-normal frequency followed by provision of insights before any inserts. In concert with its multitenancy approach, this approach meets challenges related to incremental loads, scalability and fault tolerance.

Although batch data processing restricts one to querying distinct blocks of data analogous to the “snapshot” concept visited earlier in the document, streams are “unbounded” by nature.

Streams, which are a sequence of records, hold valuable information on events that may have transpired at a source (e.g., a signal from a gas sensor).

The stream processing application furnishes insight into the contents of the stream. For example, one may wish to know the maximum strength of the signal from the gas sensor. The application ingests data from sources via connectors or message brokers.

Sinks are then used to pass the transformed data to storage repositories. The streaming application is tasked with routing the data to multiple sinks.

Recommended by the Author

Use Data Integration Patterns to Build Optimal Architecture

Stream Processing: The New Data Processing Paradigm

Recommended For You

Selecting SQL Engines for Modern Data Workloads

Operationalizing Big Data Workloads

Leveraging Data Virtualization in Modern Data Architectures

Selecting SQL Engines for Big Data Workloads

Data Management Solutions for Technical Professionals Primer for 2020

© 2020 Gartner, Inc. and/or its affiliates. All rights reserved. Gartner is a registered trademark of Gartner, Inc. and its affiliates. This publication may not be reproduced or distributed in any form without Gartner's prior written permission. It consists of the opinions of Gartner's research organization, which should not be construed as statements of fact. While the information contained in this publication has been obtained from sources believed to be reliable, Gartner disclaims all warranties as to the accuracy, completeness or adequacy of such information. Although Gartner research may address legal and financial issues, Gartner does not provide legal or investment advice and its research should not be construed or used as such. Your access and use of this publication are governed by Gartner's Usage Policy. Gartner prides itself on its reputation for independence and objectivity. Its research is produced independently by its research organization without input or influence from any third party. For further information, see "Guiding Principles on Independence and Objectivity."

About Gartner Careers Newsroom Policies Privacy Policy Contact Us Site Index Help Get the App

© 2020 Gartner, Inc. and/or its Affiliates. All rights reserved.