BigR.io
EMPOWERING DATA

Big Data Architecture Patterns

Repeatable Approaches to Big Data Challenges for Optimal Decision Making

By Gregory Harman
Managing Partner
BigR.io, LLC

Abstract

A number of architectural patterns are identified and applied to a case study involving the ingest, storage, and analysis of a number of disparate data feeds. Each of these patterns are explored to determine the target problem space for the pattern and pros and cons of the pattern. The purpose is to facilitate and optimize future Big Data architecture decision making.

The patterns explored are:

- *Lambda*
- *Data Lake*
- *Metadata Transform*
- *Data Lineage*
- *Feedback*
- *Cross-Referencing*

About BigR.io

BigR.io is a technology consulting firm empowering data to drive analytics for revenue growth and operational efficiencies. Our teams deliver software solutions, data science strategies, enterprise infrastructure, and management consulting to the world's largest companies. We are an elite group with MIT roots, shining when tasked with complex missions: assembling mounds of data from a variety of sources, building high-volume, highly-available systems, and orchestrating analytics to transform technology into perceivable business value.

With extensive domain knowledge, BigR.io has teams of architects and engineers that deliver best-in-class solutions across a variety of verticals. This diverse industry exposure and our constant run-in with the cutting edge, empowers us with invaluable tools, tricks, and techniques. We bring knowledge and horsepower that consistently delivers innovative, cost-conscious, and extensible results to complex software and data challenges. Learn more at www.bigr.io

Introduction

Modern business problems require ever-increasing amounts of data, and ever-increasing variety in the data that they ingest. Aphorisms such as the "three V's[1]" have evolved to describe some of the high-level challenges that "Big Data" solutions are intended to solve. An introductory article on the subject may conclude with a recommendation to consider a high-level technology stack such as Hadoop and its associated ecosystem.

While this sort of recommendation may be a good starting point, the business will inevitably find that there are complex data architecture challenges both with designing the new "Big Data" stack as well as with integrating it with existing transactional and warehousing technologies.

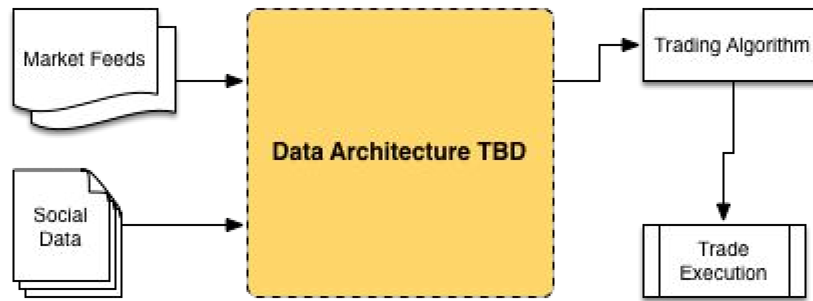
This paper will examine a number of architectural patterns that can help solve common challenges within this space. These patterns do not rely on specific technology choices, though examples are given where they may help clarify the pattern, and are intended to act as templates that can be applied to actual scenarios that a data architect may encounter.

The following **case study** will be used throughout this paper as context and motivation for application of these patterns:

Alpha Trading, Inc. (ATI) *is planning to launch a new quantitative fund. Their fund will be based on a proprietary trading strategy that combines real-time market feed data with sentiment data gleaned from social media and blogs. They expect that the specific blogs and social media channels that will be most influential, and therefore most relevant, may change over time. ATI's other funds are run by pen, paper, and phone, and so for this new fund they start building their data processing infrastructure greenfield.*

[1] Volume, Velocity and Variety

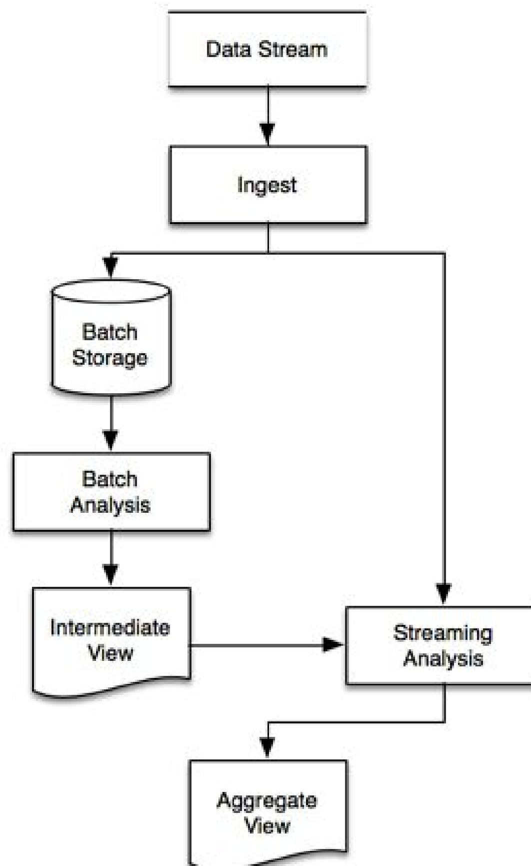
Diagram 1: ATI Architecture Before Patterns



Pattern 1: Lambda

The first challenge that ATI faces is the timely processing of their real-time (per-tick) market feed data. While the most recent ticks are the most important, their strategy relies on a continual analysis of not just the most recent ticks, but of all historical ticks in their system. They accumulate approximately 5GB of tick data per day. Performing a batch analysis (e.g. with Hadoop) will take them an hour[2]. This batch process gives them very good accuracy – great for predicting the past, but problematic for executing near-real-time trades. Conversely, a streaming solution (e.g. Storm, Druid, Spark) can only accommodate the most recent data, and often uses approximating algorithms to keep up with the data flow. This loss of accuracy may generate false trading signals within ATI's algorithm.

Diagram 2: Lambda Architecture



[2] The actual processing time would vary by any number of factors; an hour is chosen here for simplicity of explanation.

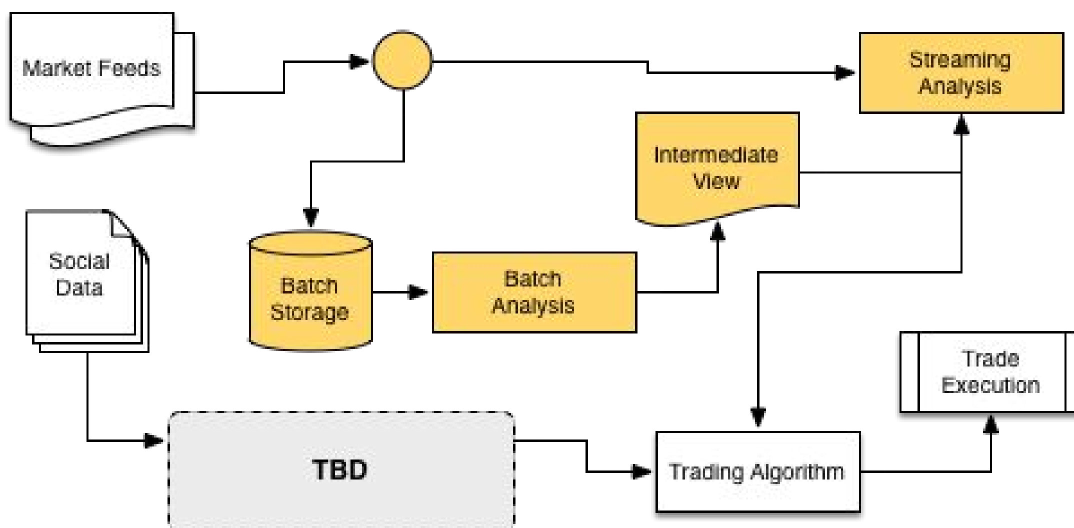
In order to combat this, the *Lambda Pattern* will be applied. Characteristics of this pattern are:

- The data stream is fed by the ingest system to both the batch and streaming analytics systems.
- The batch analytics system runs continually to update intermediate views that summarize all data up to the last cycle time — one hour in this example. These views are considered to be very accurate, but stale.
- The streaming analytics system combines the most recent intermediate view with the data stream from the last batch cycle time (one hour) to produce the final view.

While a small amount of accuracy is lost over the most recent data, this pattern provides a good compromise when recent data is important, but calculations must also take into account a larger historical data set. Thought must be given to the intermediate views in order to fit them naturally into the aggregated analysis with the streaming data.

With this pattern applied, ATI can utilize the full backlog of historical tick data; their updated architecture is as such:

Diagram 3: ATI Architecture with Lambda



The Lambda Pattern described here is a subset and simplification of the Lambda Architecture described in Marz/Warren[3]. For more detailed considerations and examples of applying specific technologies, this book is recommended.

Pattern 2: Data Lake

ATI suspects that sentiment data analyzed from a number of blog and social media feeds will be important to their trading strategy. However, they aren't sure which specific blogs and feeds will be immediately useful, and they may change the active set of feeds over time. In order to determine the

---

[3] Marz, Nathan, and James Warren. *Big Data: Principles and Best Practices of Scalable Real-time Data Systems*. Shelter Island, NY: Manning, 2015.

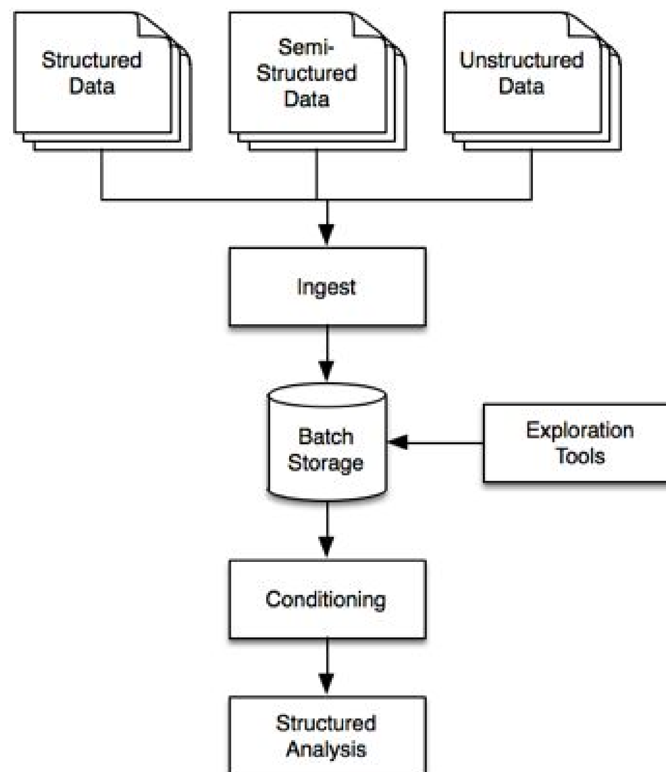
active set, they will want to analyze the feeds' historical content. Not knowing which feeds might turn out to be useful, they have elected to ingest as many as they can find.

They quickly realize that this mass ingest causes them difficulties in two areas:

1. Their production trading server is built with very robust (and therefore relatively expensive) hardware, and disk space is at a premium. It can handle those feeds that are being actively used, but all the speculative feeds consume copious amounts of storage space.
2. Each feed has its own semantics; most are semi-structured or unstructured, and all are different. Each requires a normalization process (e.g. an ETL workflow) before it can be brought into the structured storage on the trading server. These normalization processes are labor-intensive to build, and become a bottleneck to adding new feeds.

These challenges can be addressed using a *Data Lake Pattern*. In this pattern, all potentially useful data sources are brought into a landing area that is designed to be cost-effective for general storage. Technologies such as HDFS serve this purpose well. The landing area serves as a platform for initial exploration of the data, but notably does not incur the overhead of conditioning the data to fit the primary data warehouse or other analytics platform. This conditioning is conducted only after a data source has been identified of immediate use for the mainline analytics.

Diagram 4: Data Lake

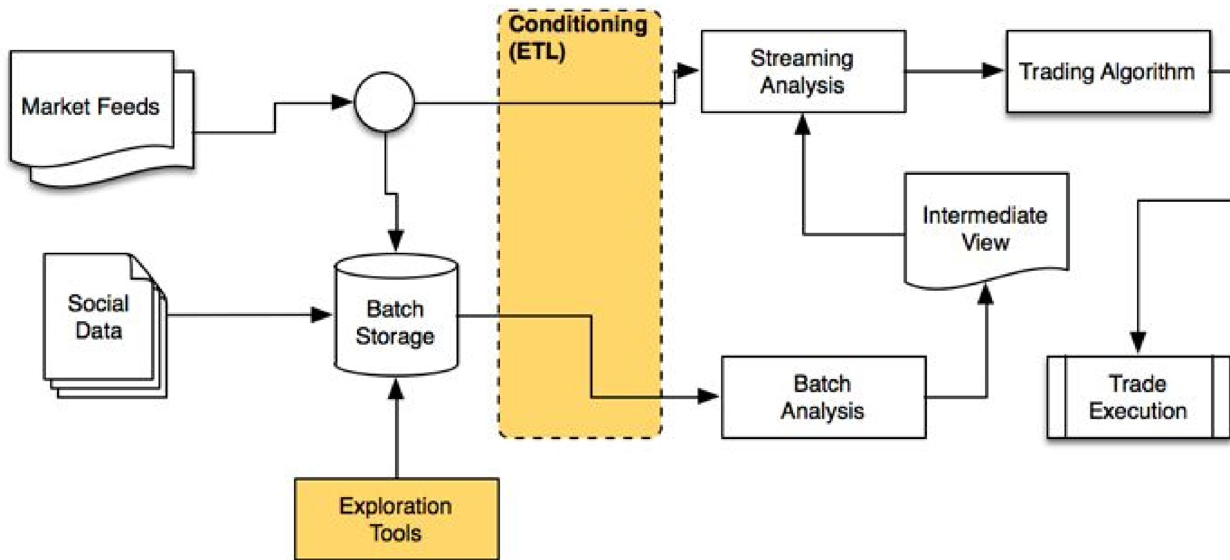


Data Lakes provide a means for capturing and exploring potentially useful data without incurring the storage costs of transactional systems or the conditioning effort necessary to bring speculative sources into those transactional systems. Often all data may be brought into the Data Lake as an initial landing platform. However, this extra latency may result in potentially useful data becoming stale

if it is time sensitive, as with ATI's per-tick market data feed. In this situation, it makes sense to create a second pathway for this data directly into the streaming or transactional system. It is often a good practice to also retain that data in the Data Lake as a complete archive and in case that data stream is removed from the transactional analysis in the future.

Incorporating the Data Lake pattern into the ATI architecture results in the following:

Diagram 5: ATI Architecture with Data Lake



Pattern 3: Metadata Transform

By this time, ATI has a number of data feeds incorporated into their analysis, but these feeds carry different formats, structures, and semantics. Even discounting the modeling and analysis of unstructured blog data, there are differences between well-structured tick data feeds. For example, consider the following two feeds[4] showing stock prices from NASDAQ and the Tokyo Stock Exchange:

Field	NASDAQ	Tokyo Stock Exchange	Normalized?
Date	01/11/2010	10/01/2008	✓
Time	10:00:00.930	08:20:04.490	✓[5]
Control Number	N/A	NULL	Field mismatch
Price	210.81	172.0	Currency
Volume	200	7000	✓

[4] These feed formats and samples are taken from the quote file format definitions published by Tick Data, Inc. as of 6/28/15. Note that even coming from the same organizational source, these feeds have significant representational differences.

[5] Only true for data after October 1, 2008. Previously millisecond data was not available, which may be considered a normalization difference.

Field	NASDAQ	Tokyo Stock Exchange	Normalized?
Exchange Code	Q	11	Different encoding systems
Sales Condition	@F	0	Different encoding systems
Volume Flag	N/A	NULL	Field mismatch
Correction Indicator	00	N/A	Field mismatch
Sequence Number	155401	N/A	Field mismatch
Trade Stop Indicator	NULL	N/A	Field mismatch
Source of Trade	N	N/A	Field mismatch
Trade Reporting Facility	NULL	N/A	Field mismatch
Exclude Record Flag	NULL	NULL	Encoding, usage (Nasdaq = exclusion flag; TSE = closing flag)
Filtered Price	NULL	NULL	Currency

The diagram above reveals a number of formatting and semantic conflicts that may affect data analysis. Further, consider that the ordering of these fields in each file is different:

NASDAQ: *01/11/2010,10:00:00.930,210.81,100,Q,@F,00,155401,,N,,,*
TSE: *10/01/2008,09:00:13.772,,0,172.0,7000,,11,,*

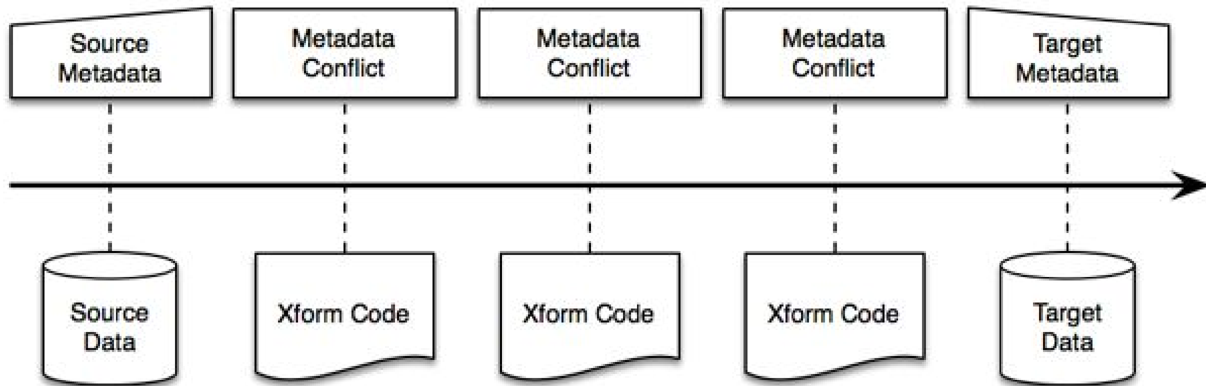
Typically, these normalization problems are solved with a fair amount of manual analysis of source and target formats implemented via scripting languages or ETL platforms. This becomes one of the most labor-intensive (and therefore expensive and slow) steps within the data analysis lifecycle. Specific concerns include:

- Combination of knowledge needed: in order to perform this normalization, a developer must have or acquire, in addition to development skills: knowledge of the domain (e.g. trading data), specific knowledge of the source data format, and specific knowledge of the target data format.
- Fragility: any change (or intermittent errors or dirtiness!) in either the source or target data can break the normalization, requiring a complete rework.
- Redundancy: many sub-patterns are implemented repeatedly for each instance – this is low-value (re-implementing very similar logic) and duplicates the labor for each instance.

Intuitively the planning and analysis for this sort of work is done at the metadata level (i.e. working with a schema and data definition) while frequently validating definitions against actual sample data. Identified conflicts in representation are then manually coded into the transformation (the “T” in an ETL process, or the bulk of most scripts).

Instead, the Metadata Transform Pattern proposes defining simple transformative building blocks. These blocks are defined in terms of metadata – for example: “perform a currency conversion between USD and JPY.” Each block definition has attached runtime code – a subroutine in the ETL/script – but at data integration time, they are defined and manipulated solely within the metadata domain.

Diagram 6: Metadata Domain Transform

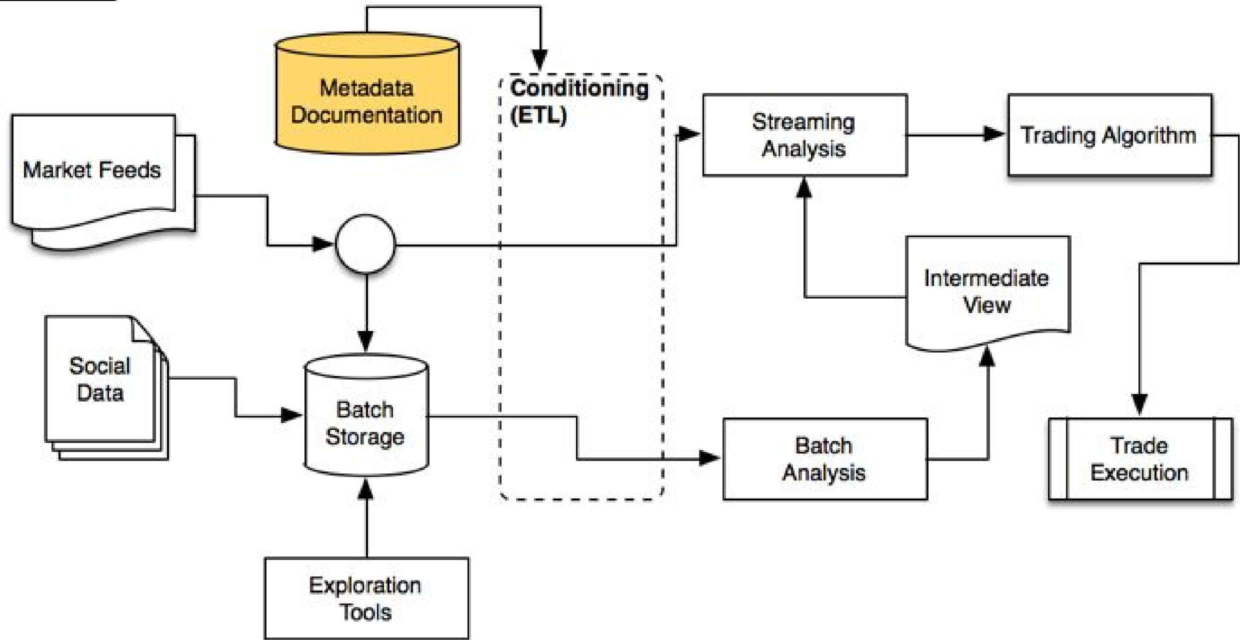


This approach allows a number of benefits at the cost of additional infrastructure complexity:

- Separation of expertise: Developers can code the blocks without specific knowledge of source or target data systems, while data owners/stewards on both the source and target side can define their particular formats without considering transformation logic.
- Code generation: defining transformations in terms of abstract building blocks provides opportunities for code generation infrastructure that can automate the creation of complex transformation logic by assembling these pre-defined blocks.
- Robustness: These characteristics serve to increase the robustness of any transform. As long as the metadata definitions are kept current, transformations will also be maintained. The response time to changes in metadata definitions is greatly reduced.
- Documentation: This metadata mapping serves as intuitive documentation of the logical functionality of the underlying code.

Applying the Metadata Transform to the ATI architecture streamlines the normalization concerns between the market data feeds illustrated above and additionally plays a significant role within the Data Lake. Given the extreme variety that is expected among Data Lake sources, normalization issues will arise whenever a new source is brought into the mainline analysis. Further, some preliminary normalization may be necessary simply to explore the Data Lake to identify currently useful data. Incorporating the Metadata Transform pattern into the ATI architecture results in the following:

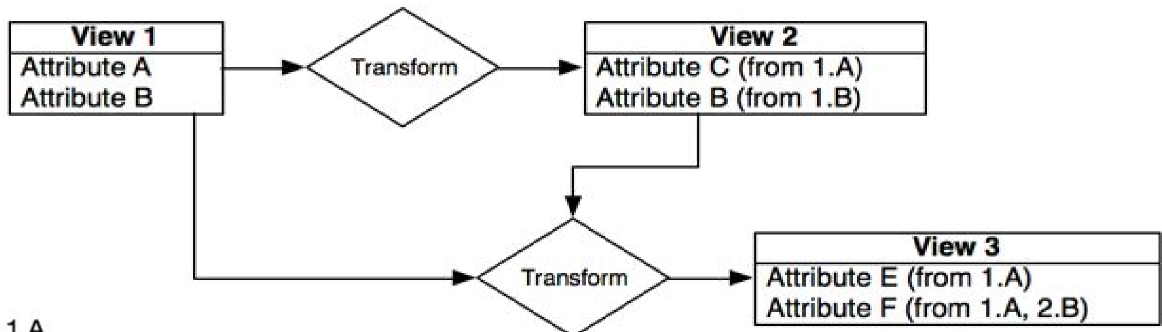
Diagram 7: ATI Architecture with Metadata Transform



Pattern 4: Data Lineage

Not all of ATI's trades succeed as expected. These are carefully analyzed to determine whether the cause is simple bad luck, or an error in the strategy, the implementation of the strategy, or the data infrastructure. During this analysis process not only will the strategy's logic be examined, but also its assumptions: the data fed into that strategy logic. This data may be direct (via the normalization/ETL process) from the source, or may be take from intermediate computations. In both cases, it is essential to understand exactly where each input to the strategy logic came from – what data source supplied the raw inputs.

The Data Lineage pattern is an application of metadata to all data items to track any “upstream” source data that contributed to that data's current value. Every data field and every transformative system (including both normalization/ETL processes as well as any analysis systems that have produced an output) has a globally unique identifier associated with it as metadata. In addition, the data field will carry a list of its contributing data and systems. For example, consider the following diagram:



Note that the choice is left open whether each data item's metadata contains a complete system history back to original source data, or whether it contains only its direct ancestors. In the latter case, storage and network overhead is reduced at the cost of additional complexity when a complete lineage needs to be computed.

This pattern may be implemented in a separate metadata documentation store to the effect of less impact on the mainline data processing systems, however this runs the risk of a divergence between documented metadata and actual data if extremely strict development processes are not adhered to. Alternately, a data structure that includes this metadata may be utilized at "runtime" in order to guarantee accurate lineage. For example, the following JSON structure contains this metadata while still retaining all original[6] feed data:

```
{
  "data" : {
"field1" : "value1",
"field2" : "value2"
},
"metadata" : {
        "document_id" : "DEF456",
        "parent_document_id" : ["ABC123", "EFG789"]
        }
}
```

In this JSON structure the decision has been made to track lineage at the document level, but the same principal may be applied on an individual field level. In the latter case, it is generally worth tracking both the document lineage and the specific field(s) that sourced the field in question.

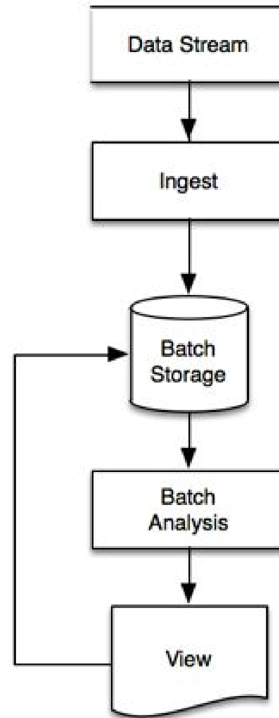
In the case of ATI, all systems that consume and produce data will be required to provide this metadata, and with no additional components or pathways, the logical architecture diagram will not need to be altered.

Pattern 5: Feedback

Frequently, data is not analyzed in one monolithic step. Intermediate views and results are necessary – in fact the Lambda Pattern depends on this, and the Lineage Pattern is designed to add accountability and transparency to these intermediate data sets. While these could be discarded or treated as special cases, additional value can be obtained by feeding these data sets back into the ingest system (e.g. for storage in the Data Lake). This gives the overall architecture a symmetry that ensures equal treatment of internally-generated data. Furthermore, these intermediate data sets become available to those doing discovery and exploration within the Data Lake and may become valuable components to new analyses beyond their original intent. As higher order intermediate data sets are introduced into the Data Lake, it's role as data marketplace is enhanced increasing the value of that resource as well.

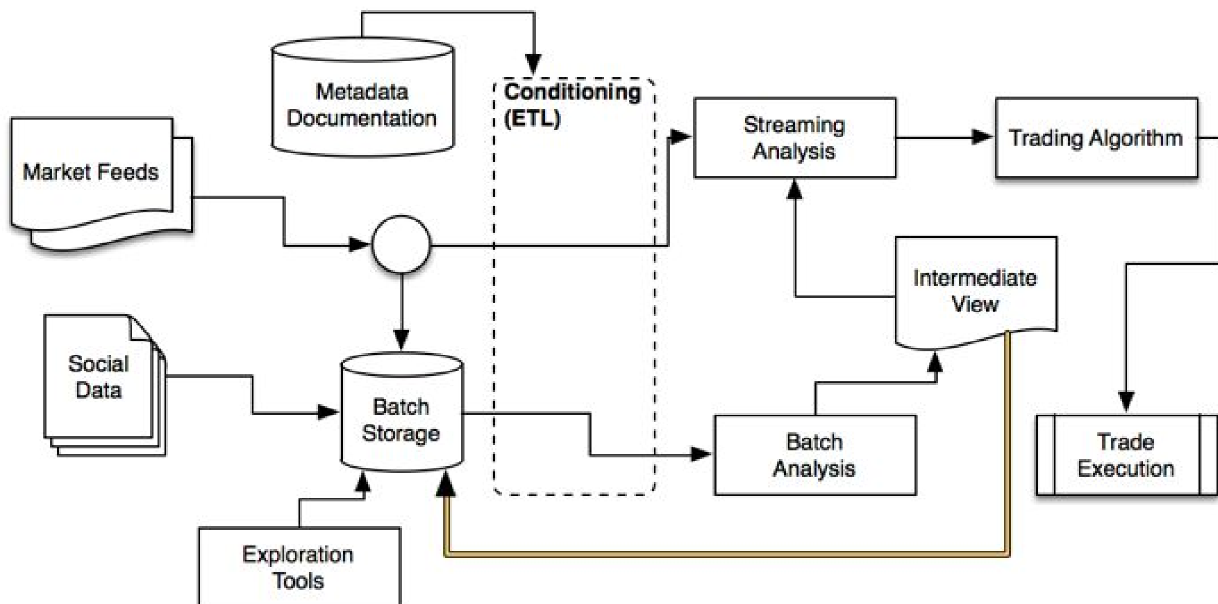
[6] A careful reader will note that fewer fields are represented here than in the original feed example given earlier in this paper. This reflects that the feed data has been transformed since its original ingest, and fields deemed unnecessary to the final analysis have been dropped.

Diagram 9: Feedback Pattern



In addition to incremental storage and bandwidth costs, the Feedback Pattern increases the risk of increased *data consanguinity,* in which multiple, apparently different data fields are all derivatives of the same original data item[7]. Judicious application of the Lineage pattern may help to alleviate this risk.

Diagram 10: ATI Architecture with Feedback



[7] As an aside, this is perhaps similar to the risk models applied to sub-prime mortgage derivatives. As new derivatives were created from aggregates of other derivatives, the *consanguinity* in which the seemingly uncorrelated risk/return models were actually all derived from the same underlying high-risk mortgages caused a divergence of perceived risk from actual risk.

ATI will capture some of their intermediate results in the Data Lake, creating a new pathway in their data architecture.

Pattern 6: Cross-Referencing

By this point, the ATI data architecture is fairly robust in terms of its internal data transformations and analyses. However, it is still dependent on the validity of the source data. While it is expected that validation rules will be implemented either as a part of ETL processes or as an additional step (e.g. via a commercial data quality solution), ATI has data from a large number of sources and has an opportunity to leverage any conceptual overlaps in these data sources to validate the incoming data.

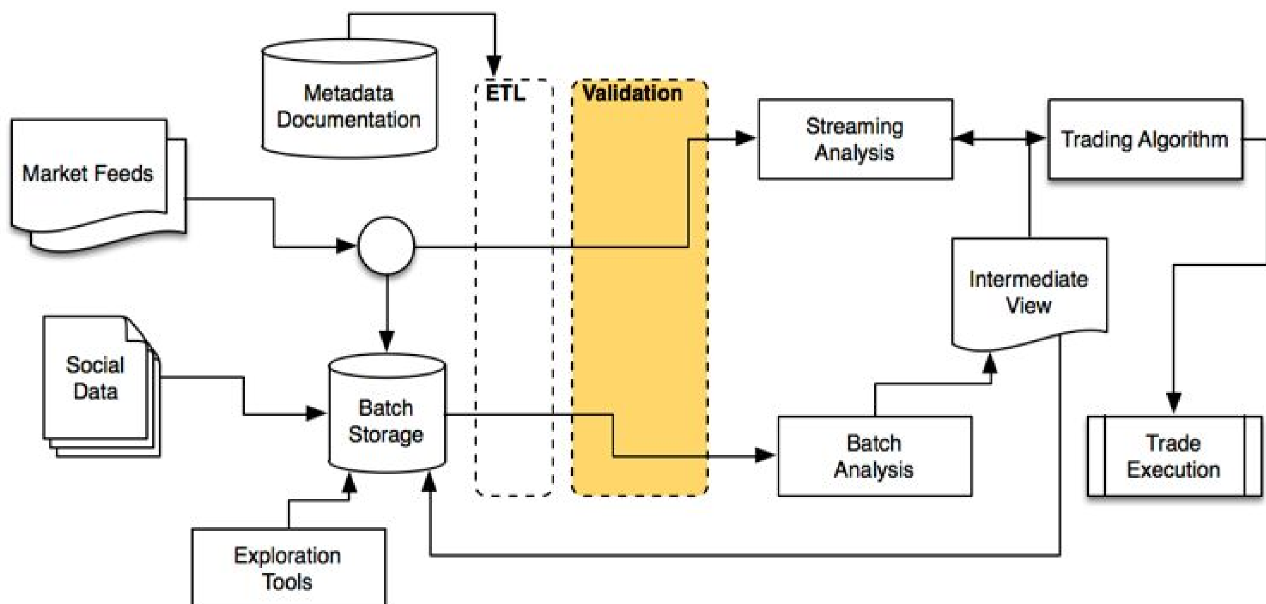
The same conceptual data may be available from multiple sources. For example, the opening price of SPY shares on 6/26/15 is likely to be available from numerous market data feeds, and should hold an identical value across all feeds (after normalization). If these values are ever detected to diverge, then that fact becomes a flag to indicate that there is a problem either with one of the data sources or with the ingest and conditioning logic.

In order to take advantage of cross-referencing validation, those semantic concepts must be identified which will serve as common reference points. This may imply a metadata modeling approach such as a Master Data Management solution, but this is beyond the scope of this paper.

As with the Feedback Pattern, the Cross-Referencing Pattern benefits from the inclusion of the Lineage Pattern. When relying on an agreement between multiple data sources as to the value of a particular field, it is important that the sources being cross-referenced are sourced (directly or indirectly) from independent sources that do not carry correlation created by internal modeling.

ATI will utilize a semantic dictionary as a part of the Metadata Transform Pattern described above. This dictionary, along with lineage data, will be utilized by a validation step introduced into the conditioning processes in the data architecture. Adding this cross-referencing validation reveals the final-state architecture:

Diagram 11: ATI Architecture Validation



Conclusion

This paper has examined a number patterns that can be applied to data architectures. These patterns should be viewed as templates for specific problem spaces of the overall data architecture, and can (and often should) be modified to fit the needs of specific projects. They do not require use of any particular commercial or open source technologies, though some common choices may seem like apparent fits to many implementations of a specific pattern.

If you would like additional insight into designing a new data architecture or assessing, expanding, and improving an existing architecture, please contact BigR.io to explore how we might be able to add lift to your initiative. Learn more at www.bigr.io.

Authored by:

Gregory Harman,
Managing Partner
BigR.io, LLC