AgilData: An Agile Approach to Streaming Fast Data

Introduction

Big Data is growing fast, from a multitude of sources – in fact we are in the midst of a true Big Data *explosion*. In fact, as of 2014, every day 5 exabytes (5×1018) of data were created. With new Big Data sources coming online all the time, it is worldwide data growth could easily 10X this figure in the coming years and by 2018, half of all consumers will interact with services based on cognitive computing on a regular basis.

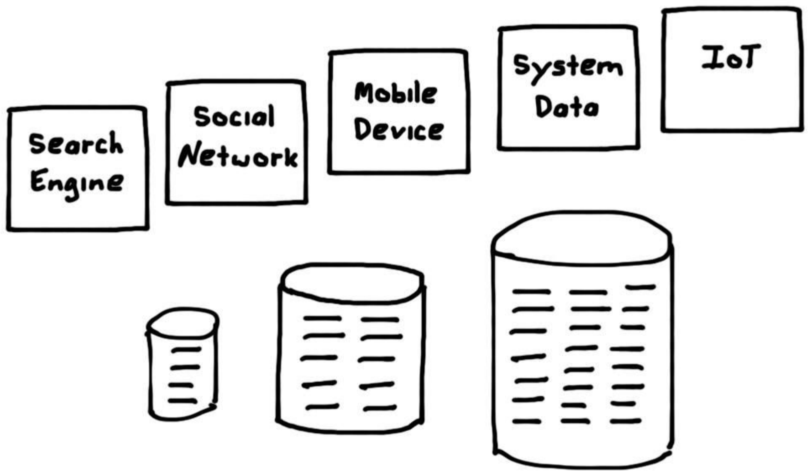

Figure 1: Big Data is growing from a multitude of sources

Big Data sources include Search Engines, Social Networks, Mobile Devices, System Data, and now the Internet of Things (IoT). Virtually every aspect of our lives will be impacted by data as ever-more capable devices and smart components are introduced to society. These components generate data – lots of it – often in ways we never imagined possible just a few short years ago.

We should not forget to also include the impact of growing transaction systems on Big Data. Many operational databases are growing rapidly as well, expanding in to the multi-terabyte range and beyond. These systems provide a rich and often vital source of Big Data, one that must not only provide answers to complex queries, but additionally support the traditional needs of transactionality, data integrity, high performance and reliability.

Recently another new development has emerged in the Big Data landscape: Fast Data or the need to know what is *happening now*, in real time. It is no longer adequate to understand what is contained in Big Data sources last month, last week or even yesterday – companies need to see a real time view of Big Data *as it happens*. With this type of visibility, organizations can respond immediately to changes in areas such as the market, user patterns, fraud detection, and many others.

Virtually all organizations and business will be affected by and need access to Big Data – even those that are not directly in the industries shown in Figure 1. Here are just a few simple examples:

- An online game application may generate its own Big Data for tracking user actions during gameplay, yet also needs to access Social Network feeds to interact with its users on a real time basis.
- A healthcare practitioner will need access to many Big Data stores, such as government-provided disease tracking databases and centralized electronic healthcare records.
- An aircraft mechanic must access up-to-date services bulletins for the specific aircraft being worked on at that moment, in order to perform potentially critical (not to mention life-saving) maintenance.
- Climatologists must integrate and study Big Data from several sources to gauge the impact of climate change on the environment, the population and our vital physical resources.
- A CEO or other executives in large organizations must have improved visibility into both operational data within their companies, as well as insights into the external Big Data sources that affect the business and its growth.

We have also seen an emergence of a whole new class of Big Data applications; applications that *aggregate* Big Data from a variety of sources, to extract new meaning and value by combining, interpreting and creating new views across these data sources.

In short, Big Data and its impact will affect us all, and in ways that have not yet been conceived.

Thus as this trend toward ubiquitous Big Data rapidly evolves, the need for organizations to *tap into* and *leverage* these Big Data sources, for strategic advantage, competitive positioning, and to meet the emerging needs of users becomes a critical concern. Organizations that do not capitalize on the Big Data explosion will be left behind, with competitive offerings outstripping their capabilities.

As the volume, number of sources, complexity and demands of Big Data infrastructures increase, organizations are struggling to manage, interpret and extract real meaning from all this data. In other words, what does all this data mean and how can we make it valuable to the organizations that collect or access this data?

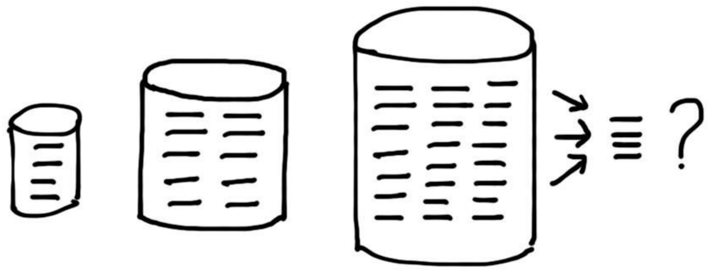

Figure 2: What does all this Big Data mean?

AGILDATA

After spending considerable time looking at this problem, it is apparent that we need a different *approach to Fast* Big Data, one that is flexible and *agile* in nature. We need to be able to access fast data in a way that more naturally conforms to our constantly changing applications and needs.

This is emphasized by the fact that most development organizations today utilize A*gile* software development methodologies, an iterative approach to building features and functionality. In some cases the entire company or organization also aspires to function in an *Agile* manner, allowing fast and nimble response to changes in the marketplace and various competitive threats.

If being *agile* is our goal when it comes to software development processes and organizational management, why is it that data infrastructures continue to be a barrier to application change? It can be simply stated this way:

> *Your process is* Agile*…Why isn't your data?*

Considering this set of emerging problems, it is time to reconsider how we have been looking at data, and take an entirely new view of the problem.

Have we been Looking at Databases Wrong all this Time?

Today's databases are typically used as *static* repositories, with this simple flow:

- Create an initial data structure (schema)
- Put data into the database via this schema
- Retrieve data from the database via this schema

This is a totally understandable and natural way to look at a data infrastructure, after all we normally consider that a database is a persistent repository for our data.

When an application starts out, the schema is simple, and things tend to work very well. Developers and data architects generally start with a data modeling process, defining a schema that will meet these early requirements. The queries are straightforward, as the initial schema often addresses – and more importantly mirrors – the first set of features of the application. These features are generally easier to implement in an early phase, particularly given the simple nature. This of course ties to the iterative nature of an *agile* process, building a small set of features initially and then adding complexity and new functionality in smaller chunks as the application moves forward.

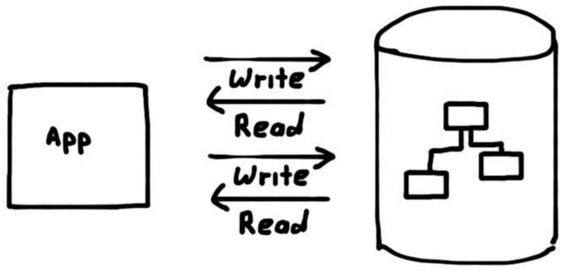

Figure 3: A simple application and database schema

New Requirements Drive Application Complexity

As requirements evolve and new features are added, the data structure becomes more complex. At the same time application data is growing, this also adds to the Big Data challenge over time. Along with the new requirements and schema complexity, the application code typically gets *far more complex*. The application is required to do the following types of tasks:

- Read data using complex queries
- Process the data into potentially multiple formats
- Write the data back to the database, sometimes performing complicated batch processing

As a result, not only does the application get more complex, the database typically experiences significant performance degradation, given that the schema and query complexity results in more interaction and lower efficiency when interacting with the database.

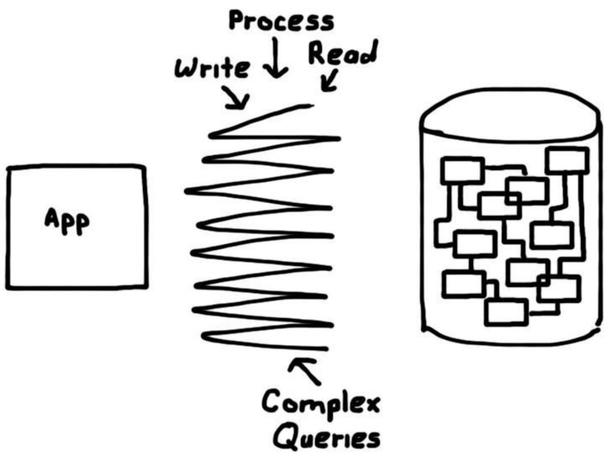

Figure 4: Changing requirements drive schema complexity, performance degradation

Resorting to Multiple Data Engines to Improve Performance

Today developers have a large array of database engines to choose from. To address the problem of performance degradation and complexity, organizations often resort to having *multiple* database engines for a single application, using one or more new engines that support performance and data requirements more closely. This can work effectively to improve performance, but now the application becomes *far more complex* – the developer must now interact with multiple data structures in multiple database engines. Often each engine has its own API, and the developer is responsible for writing data in a coordinated way across the various engines. Given than many NoSQL and NewSQL data engines do not support strong transactional semantics, this can lead to data inconsistency, and relies on application code to manage inter-database operations.

AGILDATA

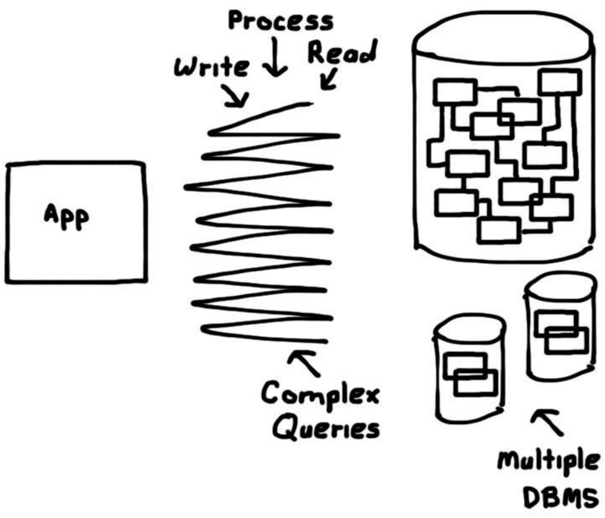

Figure 5: Complex Queries

The result is a complex graph of application-to-database interaction – the application must go back-and-forth to the database performing many complex steps, making code difficult to develop and maintain.

By looking at databases as a static repository with relatively fixed schemas, the end result is a data structure (schema) that does not mirror application needs, perpetuating the cycle of development and database complexity over time.

Working with a Distributed Database Cluster

This problem is significant, but there is yet another entire dimension to the problem. Today, applications and data sources are driving the Big Data explosion as described earlier. This necessitates a *distributed database cluster* of one form or another to handle the load at scale. Implementation of a database cluster requires *horizontal partitioning* of the data across multiple servers. Such partitioning, whether in a relational, NoSQL or NewSQL database engine, segments the data by a *key* across the various nodes (servers) in the cluster. This is typically done using a hash algorithm, and is a technique called *database sharding*.

Database sharding can be extremely effective for horizontal scaling, as data read and write operations can be spread across many servers. In an ideal world, all or most read/write operations in a distributed cluster can be performed on a *single shard* (server). However, when data is partitioned by a key it is an inevitable occurrence that many operations cannot be performed on a single shard, and *distributed* *operations* are required. These distributed operations must access (read or write) data to and from *multiple shards* in a coordinated manner, and result in significant network traffic between nodes and the application. As a result, latency goes up, and performance suffers; distributed operations are often much slower when compared with a single monolithic database instance.

This problem is commonly understood, but what is not as obvious is the effect on database structure for accessing data as horizontal partitioning is implemented. When a given *key* is selected to shard a database, it will support a certain subset of application requirements well – but make other requirements extremely cumbersome and reduce performance. In effect, sharding the data in a particular manner is good for certain use cases and bad for many others. This factor exacerbates the need for complex distributed operations in the cluster.

To illustrate the problem, here is a simple example based on a game application. A game is made of *Players* and *Games*. Let's assume that we are designing a multi-player game, where several *Players* can compete in a given instance of a *Game* (a *PlayerGame*).

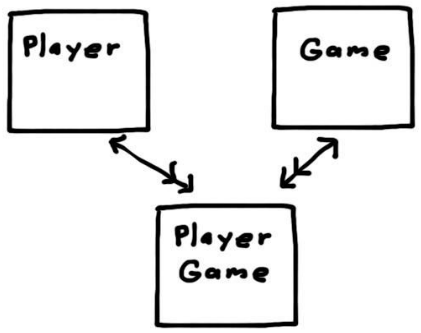

Figure 6: Example Game Data Model

The game works well, and becomes extremely popular, growing to millions of users per day in just a few months. Thus we decide to implement a database cluster for horizontal partitioning to accommodate the load.

A key requirement is for a *Player* to view the *PlayerGames* he or she has played. Therefore, a logical *key* for partitioning the database horizontally is *PlayerID*. This ensures that all information for any given *Player* will reside on a single shard (node or server). Therefore, obtaining a list of *Games* by *Player* is can be performed on a single shard and performs well.

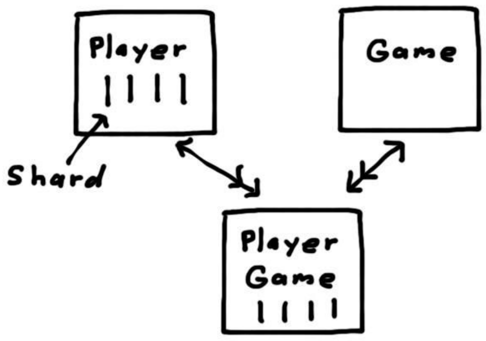

Figure 7: Game Data Model Sharded by Player

Now we get a new requirement from users of our game: they request the ability to see the competitor *Players* for a given instance of a *Game*. Unfortunately, because our model is partitioned by *PlayerID* (and thus the *Player* object), the only way to satisfy this requirement is to perform a *distributed operation*. In this case there is no choice other than searching *all shards* for other *Players* that competed in the *PlayerGame* instance. In some databases, this requires a scan of the entire database, often using a *MapReduce* algorithm. No matter the implementation, such an operation is slow, complex, and certainly cannot meet the performance needs of a high-volume interactive application.

To solve this, we can take our data and create an additional data structure, but sharded to meet this new requirement. In this case we shard by *Game* to meet the application need for efficiently accessing a list of all competitor *Players* who played a particular *Game* instance. This is called *data de-normalization*, creating multiple structures and copies of portions of the data in a database to improve efficiency and convenience.

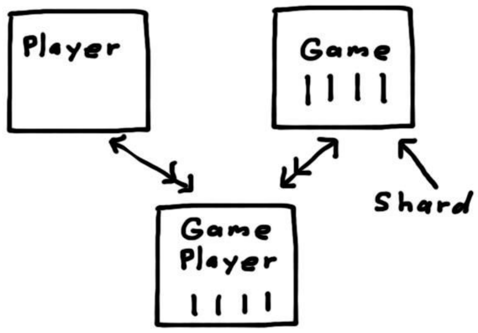

Figure 8: Game Data Model Sharded by Game

Now we can have efficient access of this data from a single shard (node or server), obtaining a list of all *GamePlayer* objects for a given *Game*.

However, we need to consider the burden this places on the application developer, making the application code even more complex. Now the application must write the data to *two places* to accommodate the dual sharding schema that has been implemented. This is not only additional work, it is also challenging to maintain data consistency given that many data engines do not support *distributed transactions*. Even when distributed writes are supported, the result is complex code.

Use of an In-Memory Cache

Often to get around such situations, architects and developers will implement an *in-memory caching* solution for these type of situations. This again can improve performance, but maintaining *cache consistency* is similar in nature to the de-normalization technique described above. In essence a cache is simply another copy of the data – one that must be maintained by the developer. Maintaining a cache involves complexity such as *cache invalidation*: knowing when to remove something from the cache if the underlying data is changed. This is often described as one of the "two hard problems" in computer science.

An in-memory cache is also a form of de-normalization, and again something that must be maintained by the application code, effectively adding to complexity and maintainability.

Aggregate Data Drives Operational Intelligence

The final area contributing database and application complexity is the need for *aggregate data*. As stated earlier, a key factor in the Big Data realm is to extract meaning from data to support *actionable decisions* through *operational intelligence*. If the right information can be derived from data sets, then informed users can arrive at the best possible business decisions – a vital factor ensuring the success and sustainability of any given organization or enterprise. Simply put, when the right data is not available, or is not provided in a meaningful form that can be acted upon in the decision-making process, then the value of the data is diminished.

Aggregate (or summarized) data gives *meaning* to Big Data, from a variety of sources (including transactional systems). This data can be used for a many purposes, including dashboards, static reports and feeds to other systems and processes.

Typically aggregate data is generated through the process of scanning large amounts of data in a given database. This is a compute-intensive process and can put a large burden on source transactional systems. Thus the generation of aggregate data in this manner must be strictly controlled, often running report or analysis queries in off hours or on redundant copies of the data.

Many organizations utilize specialized data warehouse engines for aggregate data analysis; such engines provide optimized support for large analytics queries. This is advantageous in that such queries are off-loaded from core transactional systems. Populating a data warehouse requires some sort of *extract/transform/load* process, pulling data from transactional or other sources and loading it into the specialized database in the proper format.

Alternatively, there has been significant adoption of MapReduce engines (such as Hadoop) as data stores for large Big Data sets. Using MapReduce it is feasible to compute complex aggregate summary data from a base data set. MapReduce is effectively a parallel batch process, and can be helpful for many use cases that are not time critical. The drawback again is that this requires specialized approaches and development to accomplish.

The primary drawback to each of these approaches for aggregate data analysis is that they are fundamentally *historical* in nature, generating meaningful data *after the fact*. The time delay for operational intelligence visibility can be hours, days or even weeks as such processing can only be performed on a periodic basis. In today's fast-moving economy and markets this is no longer acceptable – organizations need to see data in *real time*, getting the answers they need *now*. Further, each of these methods for extracting meaning from Big Data are complex, requiring significant development or system administration resources to establish; such resources can be better utilized driving *forward innovation* for the organizations and enterprises they serve.

The Wrong Way to Look at Databases

When addressing these problems, we need to take a step back. Is it possible that these critical issues can be attributed to looking at databases the *wrong way*? And more importantly, if there is something wrong with how we look at our data infrastructure, are there alternatives that can make a meaningful improvement?

The current view of the data infrastructure can be summarized by these points:

- Considering a database as a *static* repository;
- Relying on *fixed schemas* that do not directly reflect application needs;
- Continuing the vicious cycle of *adding complexity* to the data architecture, placing increasing burden on application functionality and developers.

It should be noted that a large driver of the current method of looking at and working with databases is dependent on traditional data modeling techniques. The most prevalent of these initially evolved to support relational database management systems (RDBMS): Entity-Relationship Modeling. A key tenet of this data modeling technique is *database normalization*, avoiding duplication of data in a given data model. The purpose of normalization was to ensure the integrity of a relational model, and was also highly driven to *conserve disk space*. When RDBMS systems were first widely adopted, database storage was a costly resource. This is no longer true; storage is plentiful, increasing in performance and becoming more cost-effective over time as technology and competition evolve.

In today's world, there are many types of database engines that offer new approaches to the traditional RDBMS, many directed at allowing more flexibility to the application developer. These data engines are included in the NoSQL and NewSQL category, and achieved rapid popularity due to their simplicity and flexibility of data structure. However, with this flexibility available to the application developer, the same drawbacks of application complexity required for maintenance of multiple copies of all or portions of the data in one or more engines still adds an undue burden for the application developer. These newer data engines provide an important step forward, yet many of the pitfalls described earlier still remain.

If we continue along the current path, Big Data systems and the applications that interact with them will continue to become more complex, more unmanageable, and most importantly difficult to extract meaning from. This trend toward undue complexity will accelerate due to the fast-changing demands of the market, the addition of new Big Data sources, and the rapid expansion of the size of Big Data.

In fact, you could say that there is a risk of Big Data stores evolving into repositories for “dead” data – data that provides little or no value to the organization.

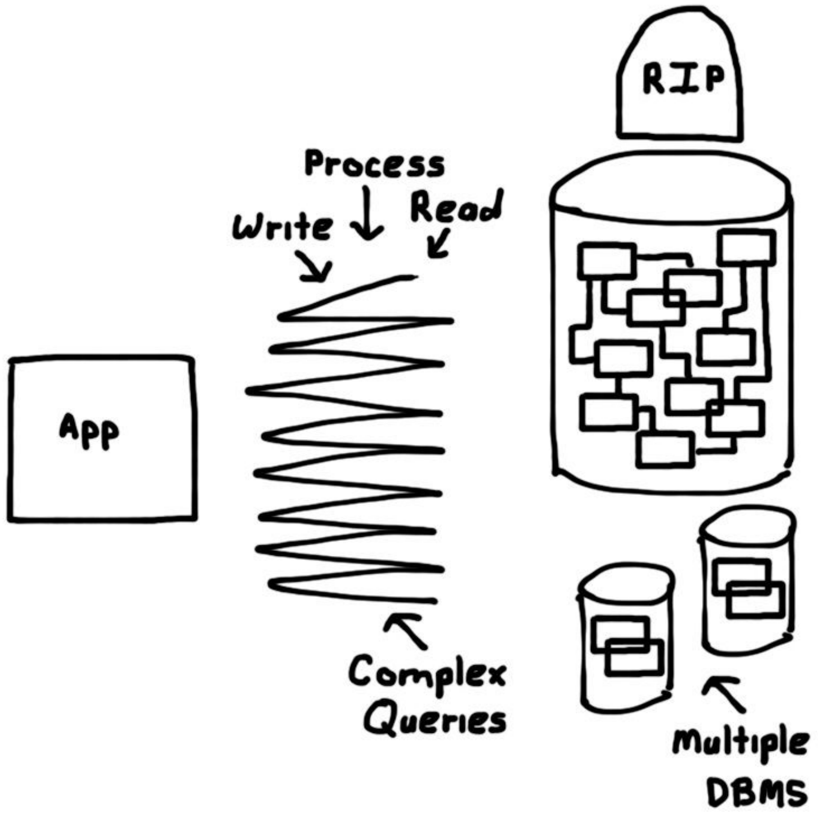

Figure 8 Big Data stores may evolve into repositories for "dead" data

An Agile View of Big Data

We have covered the drawbacks of the current way we think about databases; not only does this paradigm no longer meet our needs, it does not reflect the *way we work* today. Developers have long since adopted the popular A*gile* development approach with incremental milestones for rapid progress in accommodating fast-changing requirements. Further, organizations themselves are becoming more agile as well, moving at an increasingly fast pace to keep up with competitive demands.

So how do we solve the problem and transform the data infrastructure into a modern paradigm to fulfill these needs?

AGILDATA

What is needed is an *agile* view of Big Data. Instead of looking at a database as a *static repository*, in an agile Big Data infrastructure data is considered as a *dynamic stream*.

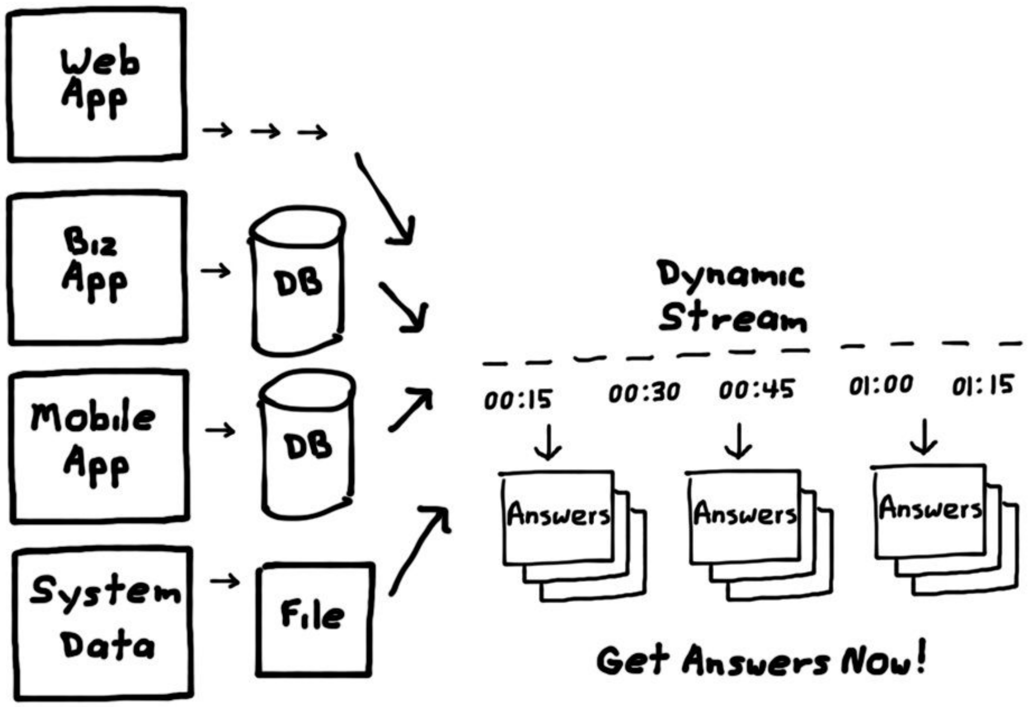

Figure 9 An agile view of Big Data

When considering data in an agile stream, we take virtually any data source (including applications, files, or existing databases) and turn them into a dynamic stream. By working with data as a stream it is possible to easily and quickly solve many problems, in a way that eases complexity for developers, yet greatly increasing the capabilities of the Big Data infrastructure. A dynamic, agile stream of data can be manipulated in a step-wise fashion, creating the exact *views* that mirror application requirements. Queries become simple, since agile views already match what the application needs. Another strength of the agile view of Big Data is that answers and meaning can be obtained *now*, in real time for On-Demand data in motion.

The key to an agile Big Data infrastructure is its ability to create dynamic *views* of data as needed to meet application requirements. The view structures can be persistent and query-able, and since these agile views mirror application requirements they can be optimized in both structure and sharding (partitioning) to accommodate high-performance access. Views are built and maintained by the agile infrastructure itself, on an incremental basis in real time. In other words, views are a picture of the data *as it happens*.

Views can be structured as indexes on other data, data in specific formats for easy application access, aggregate data, and external data engines such as data warehouses. The key is that all types of views are built and maintained from the dynamic real time stream of data.

In addition to dynamic views that meet application needs, it is very easy to add event processing to an agile data environment – triggering application functionality whenever certain data conditions are encountered.

The main trade-off with an agile Big Data infrastructure as compared to the traditional usage of today's databases is the use of more storage space. However, as covered earlier, storage is relatively inexpensive today, and the price-performance is expected to continue to improve as new technologies such as SSD become more prevalent.

In summary, an agile view of Big Data provides an exceedingly flexible infrastructure, one that can continue to adapt to evolving application requirements with relative ease.

The Ideal Agile Big Data Infrastructure

Now that we have covered the need for an agile view of data, here are some key capabilities for the *ideal* agile Big Data infrastructure.

- An agile Big Data platform must handle data in a *continuous, real time stream*;
- It must be capable of dynamically constructing and maintaining any number of *agile views* of the data in order to satisfy application requirements;
- Views must maintain *data integrity*;
- The agile platform should *isolate schema complexity* from application developers, *simplifying the application code*;
- The infrastructure must be *fully scalable, reliable and resilient*, both in terms of stream processing and dynamic views;
- An agile data platform must support *event processing*, enabling a new class of real time application capabilities;

AGILDATA

- Views must be *accessible in real time*, supporting advanced dashboards and other similar capabilities

Summary

There is no doubt that Big Data is upon us, and will continue to accelerate rapidly in the coming years. There are a host of problems introduced by Big Data, and to maintain a competitive position organizations must be able to manage Big Data sets, and more importantly be able to extract meaning and value from the data. The traditional view of databases as a static repository is inadequate to the challenge, and new innovation is required to address these new emerging needs.

An A*gile* view of Big Data – wherein data is viewed as a real time stream – offers a new look at how data is managed. Using an agile data infrastructure, organizations can conquer Big Data challenges with a level of ease, flexibility and performance never before possible.

To learn more about Agile Big Data, please refer to: www.AgilData.com

AGILDATA