White Paper

# HPCC Systems: Aggregated Data Analysis: The Paradigm Shift

Author: David Bayliss, Chief Data Scientist
Date: May 10, 2011

# Table of Contents

## Executive Summary

The HPCC (High Performance Cluster Computing) architecture is driven by a proprietary data processing language: Enterprise Control Language (ECL). When considered briefly, the proprietary nature of ECL may be perceived as a disadvantage when compared to a widespread query language such as SQL.

The following paper compares and contrasts the traditional Relationship Database Management System (DBMS)/ Structured Query Language (SQL) solution to the one offered by the HPCC ECL platform. It is shown that ECL is not simply an adjunct to HPCC, but is actually a vital technological lynchpin then ensures that the HPCC offering achieves performance levels that an SQL system is not even capable of as a theoretical ideal.

While many of the points made are applicable to data processing in general, the particular setting for this paper is the integration of huge amounts of heterogeneous data. It will be argued that the relational data model is excellent for data which is generated, collected and stored under relational constraints. However for data which is not generated or collected under relational constraints[1], the attempt to force the data into the relational model involves crippling compromises. The model-neutral nature of ECL obviates these concerns.

The capabilities of the HPCC ECL platform are sufficiently disruptive in that certain algorithms can be considered that are not realistic using a traditional RDBMS/SQL solution. The paper ends by considering a couple of case studies illustrating the new horizons that are opened by the HPCC and ECL combination.

## Introduction

The relational database is the most prevalent database management system available today; however, it is not the most suitable system for the integration and analysis of massive amounts of data from heterogeneous data sources. This unsuitability does not stem from a defect in the design of the RDBMS but is instead a feature of the engineering priorities that went into their creation. The object of this paper is to contrast an idealized RDBMS processing model with the model employed within the HPCC platform in the context of the integration and analysis of massive volumes of disparate data.

The RDBMS as a theoretical concept is distinct from SQL which is just one manifestation of an RDBMS. However, the reality is that almost every RDBMS out there supports SQL either natively or indirectly. Thus in the following SQL will be used interchangeably with RDBMS.

## The RDBMS paradigm

Relational databases are built upon the principle that the physical data representation should be entirely disjointed from the way that the data is viewed by people accessing the database. An SQL system has a notion of a table which is an unsorted bag of records where each record contains columns. An SQL query can then return a sub-set of that table and may perform simple operations to alter some of the columns of that table. An SQL system then, at least theoretically, allows any two tables to be compared or joined together to produce a new composite table. The values in those fields are pure strings or pure numbers with an SQL defined behavior. Thus the application coder can use data in an extremely neat and clean format without any thought to the underlying data structures.

The underlying data structures are then managed by the database architect/administrator. It is the architect's job to map the underlying physical data within the warehouse to the logical view of the data that has been agreed between the architect and the programmers. In particular, the architect chooses the keys that are needed to allow data to be filtered and to allow tables to be joined together.

*1) Which includes all free text and every public record source prior to 1990 and almost every one today.*

The idealized model is below[2]:

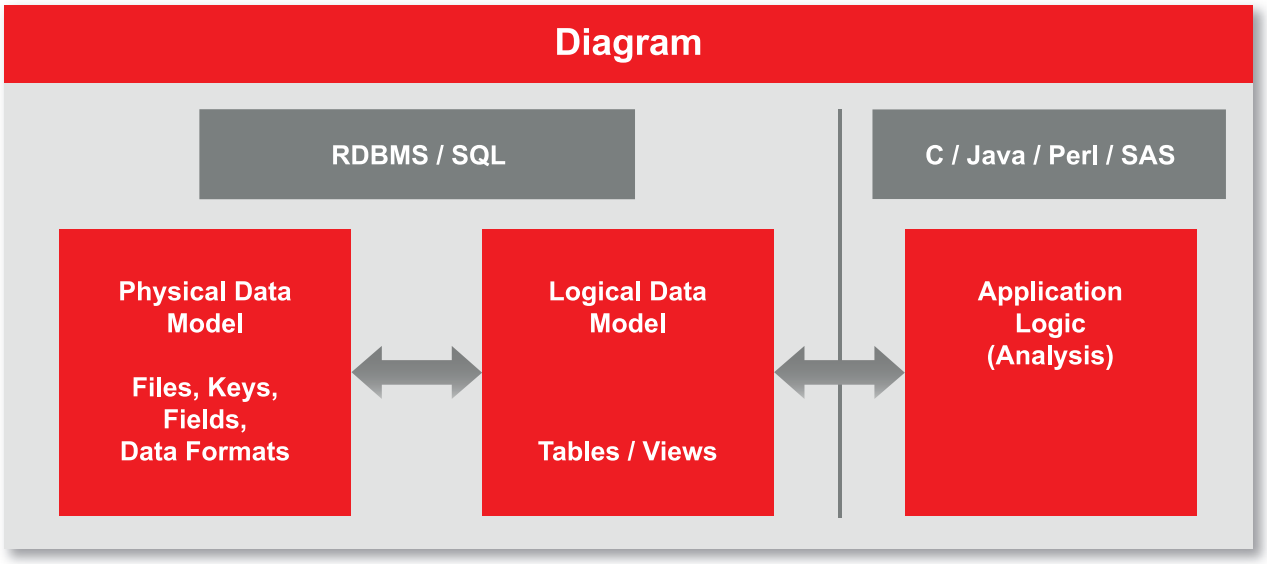


**Figure 1**

The other significant factor in this design which is not mandated but is extremely common is the notion of normal form. The premise behind normal form is that no scrap of data should appear in the system twice and that the database should be able to maintain and enforce its own integrity. Thus, for example, the 'city' column of a relational database will not typically contain the characters of the city name. Instead they contain a foreign key that is a link into a city file that contains a list of valid city names. Most RDBMS systems allow logic to be defined in the physical layer to enforce that only valid records enter the system. The beauty of removing the burden of data validation from the application logic is that the programmers cannot 'infect' the database with bad data.

## The Reality of SQL

Even within the domains for which the SQL paradigm was designed there are some flaws in the Figure 1 model that render the system practically unusable. For example, the SQL system allows for any columns in a table to act as a filter upon the result set. Thus theoretically, every query coming in to an SQL system requires the system to read every one of the records on the systems hard disk. For a large system that would require terabytes of data to be read hundreds of times a second.

The database administrator (DBA) therefore produces keys for those filters he or she believes are likely to be required. Then as long as the coders happen to use the filters the administrator happens to have guessed they wanted the query will execute rapidly. If either side misguesses then a table scan is performed and the system grinds to a halt which is not unacceptable in the business world. So what actually happens is that the encapsulation which is supposed to exist between the three boxes in Figure 1 is actually circumvented by a series of memos and design meetings between the programmers and the DBAs.

*2) Please note: in this model it is assumed that the visualization logic is separate from the application (or analysis) logic. However the visualization logic may not have been placed on this diagram. It would naturally be located to the right-hand side of the Application Logic discussed in this paper.*

**It is important to realize that modern, significant SQL applications are not independent of the underlying physical data architecture; they just don't have a systematic way of specifying the dependence that exists.**

A consequence of Figure 1 is that many SQL vendors have extended the SQL system to allow the application codes to specify 'hints' or 'methods' to the underlying database to try to enforce some of the semantic correlations that really need to exist. Unfortunately, these extensions are not covered by the SQL standard and thus the implementation of them differs from SQL system to SQL system and sometimes even within different releases of a given SQL system.

This can result in a tie between the application logic and the SQL system itself that would prevent database portability. Thus a layer of logic has been inserted to recreate that database independence. Two famous examples of this logic are Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC). However, there is a trade off. While an application using ODBC can port between different SQL vendors, it can only take advantage of those extensions that are supported by all, or most, of the SQL vendors. Furthermore, while some SQL vendors support certain extensions the quality of that support can vary from feature to feature. Even when the support is high quality there is usually some performance overhead involved in the insertion of another layer of data processing.

It should be noted that ODBC and JDBC interface come in different forms:

- Those in which the ODBC is a native interface of the SQL system and executes on the database servers, and
- Those where the ODBC layer is actually a piece of middleware acting either upon dedicated middleware servers or in the application layer itself.

The box 'on the line' in Figure 2 is an attempt to reference this. The system diagram now looks like:

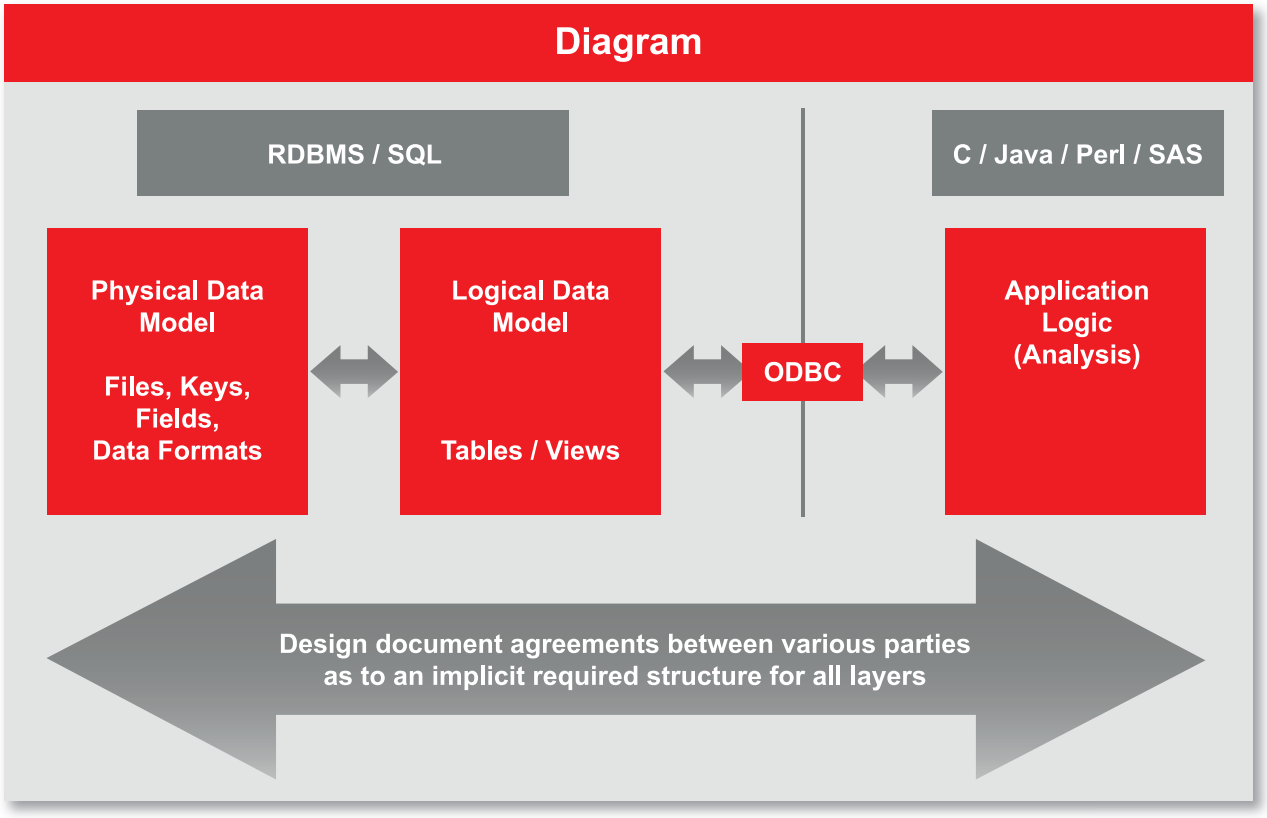


Figure 2

## Normalizing an Abnormal World

A premise of the RDBMS concept is that the data is generated, stored and delivered according to the same data model. For those in the business of collecting data from external sources, this premise is fundamentally broken. Each data source that is collected will, at best, have been generated according to a different data model. Far more commonly, the data has not really been collected according to a data model at all. The procedures in place to ensure a RDBMS has integrity simply do not apply for the majority of data that is available today.

Here are some examples of constraints placed upon a well designed RDBMS that are violated by most data that is ingested:

1) **Required fields** – When ingesting real world data, you cannot assume ANY of the fields will always be populated.

2) **Unique fields are unique** – Consider the SSN on a customer record. Often these will be mis-typed or a person will use a family member's SSN, resulting in duplications.

3) **Entity can be represented by a single foreign key** – Many of the fields relating to a person can have multiple valid values meaning the same thing. Therefore, if you wish to store not just what was referenced but how it was referenced you need at least two tables.

4) **A single foreign key can refer to only one entity** – Consider the city name. A city can be replicated in many different states.

5) **A field can take one of a discrete set of values** – Again misspellings and variations between different systems mean that the standard field lookup is invalid.

**A result of the above is that it is impossible to construct a normalized relational model that accurately reflects the data that is being ingested without producing a model that will entirely destroy the performance of the host system**.

The above assertion is very strong and there seems to be wars between teams of data architects that will argue for or against the above. However the following example has convinced anyone that has spent the time to look at the issue.

Consider the fields: city, zip and state. Try to construct a relational model for those three fields that accurately reflects:

a) A single location can validly have multiple city names (vanity city and postal city).

b) A vanity city can exist in multiple postal cities.

c) A postal city contains multiple vanity cities.

d) A zip code can span between vanity cities and postal cities even if a given vanity city of which the zip code is a part does not span the postal cities.

e) There are multiple valid abbreviations for given vanity cities and postal cities.

f) The range of valid abbreviations for a given city name can vary, dependent upon the state of which the city is a part.

g) The same city can span states, but two states can also have different cities with the same name.

h) The geographical mapping of zip codes has changed over time.

i) City names are often misspelled.

j) A single collection of words could be a misspelling of multiple different cities.

From our experience, the best seen anyone has been able to tackle the above still took eight tables, and it relied upon some extensions in one particular data vendor.

There are a number of pragmatic solutions that are usually adopted:

a) **Normalize the data fully, investing in enough hardware and manpower to get the required performance**. This is the theoretically correct solution. However, it can result in a single but large file of ingested data producing multiple terabytes of data into tens or even hundreds of sub-files. Further the data architecture team potentially has to alter the model for every new ingested file.

b) **Abandon normalization and move the data manipulation logic down into the application layer**. With this approach, the fields contain the data as collected and the task of interpreting the data is moved down to the programmers. The application typically has to fetch a lot of data in multiple steps for a process that should have been executed atomically on the database server.

c) **Insert a significant data ingest phase where the data is 'bashed' into a format that has been pre-defined by the data architects**. This is the best in terms of performance of the query system but has the twin downsides of creating a significant delay during the data ingest phase and also throwing away potentially vital data that was not compatible with the pre-defined ingest data architecture.

d) Hybridize the above three approaches on a largely ad-hoc file by file basis dependent upon the particular restrictions that were uppermost in the programmers mind at the point the data came in through the door.

While each of the first three of the above solutions has been heralded in design documents, most functioning systems evolve towards the fourth solution – the hybrid approach.

The 'RDBMS' model has now become:

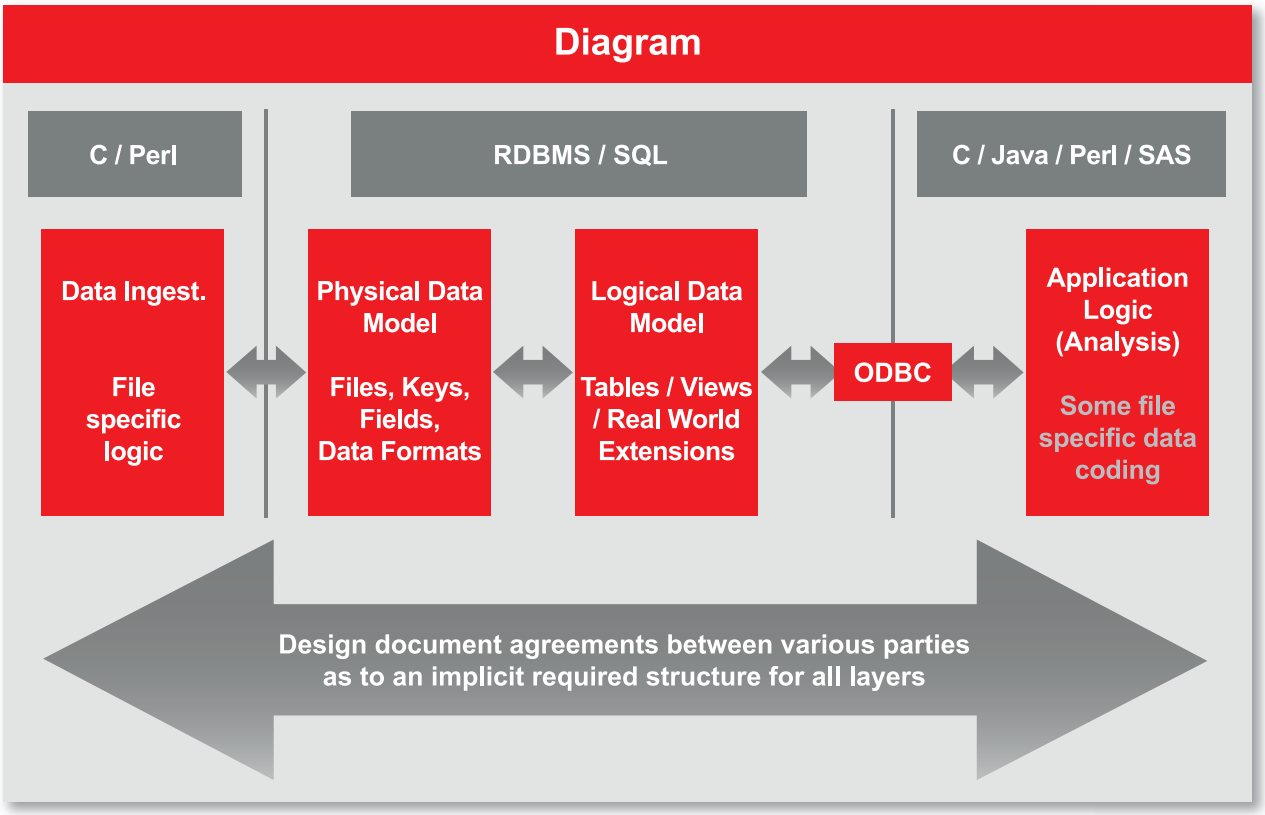


**Figure 3**

## A Data Centric Approach

In 1999, Seisint (now a LexisNexis Company) and LexisNexis independently conducted an evaluation of existing database technology and both concluded that the RDBMS was an unsuitable solution for large scale, disparate data integration.

fter coming to this conclusion, in 2000, a team of Seisint employees that had been world leaders in the field of utilizing RDBMS technology was handed a blank sheet of paper and asked to construct a model to handle huge amounts of real-world data. The result was almost diametrically opposed to the diagram presented above.

Remember, the basic premise of the RDBMS is that the physical and logical data models are entirely disjoint. This was necessary in the late eighties as programming languages lacked the ability to adequately support multiple programmers from cooperating upon a task. Therefore processes had to be produced that allowed for teams to operate. By 2000 procedural encapsulation was well understood; these layers did not need to be kept distinct for programmatic reasons. In fact, as has been discussed previously, the separation of these two layers would routinely be violated by implicit usage agreements to boost performance to acceptable levels.

Another observation is that a key factor in the integration of disparate datasets is the disparity of the datasets. Attempting to fix those disparities in three different layers with three different skill sets is entirely counterproductive. The skills need to be developed around the data, not specifically around the processes that are used to manipulate the data. The layers in a data process should represent the degree of data integration not the particular representation used by the database.

The next decision was that compiler optimization theory had progressed to the point where a well specified problem was more likely to be correctly optimized automatically than by hand. This is especially true in the field of parallel execution and sequencing. Therefore a commitment was made to invest whatever resources were required to ensure that performance could be tuned by an expert system; leaving the data programmer with the responsibility to correctly specify the correct manipulation of the data.

In particular implicit agreements between execution layers are not tolerated; they are explicitly documented in the code to allow the optimizer to ensure optimal performance.

The final piece of the puzzle was a new programming language: **Enterprise Control Language (ECL)**. This was designed to have all of the data processing capabilities required by the most advanced SQL or ETL systems but also to have the code encapsulation mechanisms demanded by systems programmers.

The result looks like this:

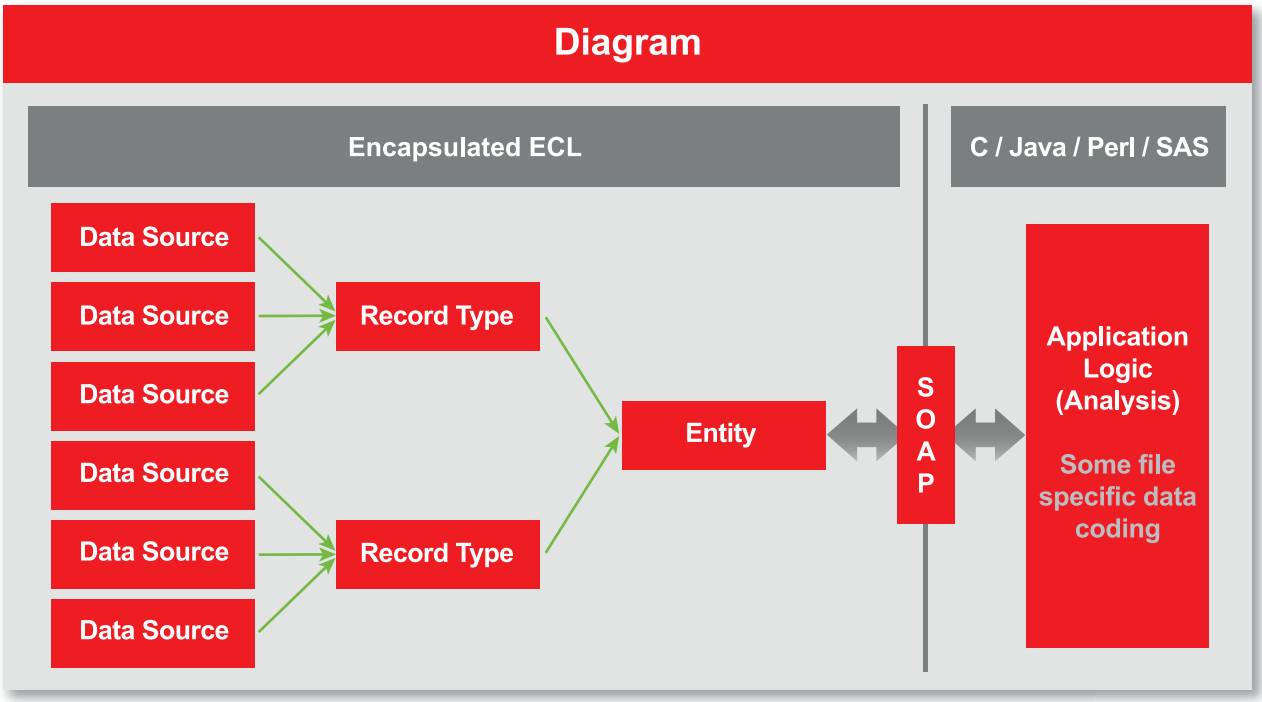


**Figure 4**

There some advantages of the system in Figure 4 that may not be immediately obvious from the diagram.

1) The data sources are stored unmodified, even though they are modified as part of delivery. Thus there is never any "loss" of information or significant pain in re-mapping the incoming files to the target formats
2) The data teams can be segmented by data type rather than language skill. This allows for every file type to be handled by individuals skilled in that field.
3) If required, a storage point between a batch ingest facility and a real-time delivery mechanism is available without a need to significantly recode the processing logic.
4) Introducing parallelism is natural and can even be done between remote processing sites.

## Data Analysis

The model in Figure 4 essentially allows for huge scale data ingest and integration. It allows for scaling of hardware and personnel. It encourages deep data comprehension and code reuse, which enhances development productivity and improves the quality of the results. However, it does not significantly improve either the quality or the performance of any analysis that is performed. This should not be surprising; the data analysis is being performed outside of the ECL system.

Data analysis is one of those terms that everyone claims to perform, although very few can define what it really is. Most will claim that it somehow involves the translation of data into knowledge, although exactly what that translation means is ill defined. In particular, some would suggest that analysis is occurring if data is simply presented on screen for an analyst to review. Others would say analysis has occurred if the data is aggregated or summarized. Others may consider data analysis if it has be represented in some alternative format.

For the purposes of this document, a much simpler definition will be used:

> *Analysis is the process of concentrating information from a large data stream with low information content to a small data stream with high information content.*

Large, small, low and high are deliberately subjective terms the definition of which changes from application to application. However, the following terms need to be defined that are crucial to the following.

• The ***integrity*** of the analysis is the extent to which the analysis process accurately reflects the underlying data. For data searching these are often measured by precision and recall.

• The ***strength*** of the analysis is the ratio of the about of data considered to the size of the result.

• The ***complexity*** of the analysis is the reciprocal of the number of entirely independent pieces that the data can be divided into prior to analysis whilst maintaining full analytic integrity.

A few examples might illustrate the previous terms:

- Entity extraction is a strong analytic process. The entity stream from a document is typically very small. As implemented today, entity extraction has very low complexity; typically every document is extracted independently of every other one.
- The LexisNexis process that ascertains whether or not people are using their correct SSN is not a strong process. The number of SSNs coming out is very similar to the number going in. Yet the process is complex in that every record in the system is compared to every other.
- The LexisNexis process that computes associates and relatives for every individual in the US is both strong and complex.
- Result summarization (such as sorting records by priority and counting how many of each kind you have) is a weak, non-complex process.
- Fuzzy matching is a strong, non-complex process.
- Pattern matching is a complex, weak process.
- Non-obvious relationship finding is a strong, complex process.

If the application data analysis being performed by the system is weak and simple then the fact that the data analysis has not been improved by this architecture is insignificant[3]. In such a scenario, well over 90% of the system performance is dominated by the time taken to retrieve the data.

A system where the application analysis is strong will probably find that a bottleneck develops in the ability of the application server to absorb the data generated from the supercomputer. If the application code is outside of the control of the developer, then this problem may be insurmountable. If the application code is within the developer's control, then movement of the strength of the analysis down into the ECL layer will produce a corresponding improvement in system performance.

*Case Study: Fuzzy Matching*

> A classical example of the above occurs millions of times each day on our LexisNexis servers. We allow sophisticated fuzzy matching of the underlying records. This includes edit distance, nick-naming, phonetic matching, zip-code radius, city-aliasing, street aliasing and partial address matching – all of which can happen at the same time. The integrity of the fuzzy match is directly proportional to the percentage of the underlying data that you perform the fuzzy scoring up. The LexisNexis HPCC system will often scan hundreds of records for every one that is returned. By moving this logic down into the ECL layer the optimizer is able to execute this code with negligible performance degradation compared to a hard match. Had the filtering logic not been moved down into the ECL layer then the fuzzy fetch would have been hundreds of times slower than a simple fetch because the application logic (on a different server) would have been receiving hundreds of times as much data.

A complex system will suffer performance degradation that is exponential in the complexity of the query. This is because complexity implicitly strengthens a query. If the database is one terabyte in size, then a one-record result that requires all data to be considered requires the full terabyte to be exported from the system. Of course this proves unacceptable, so the application typically settles upon reducing the integrity of the system by using less than the full dataset for the analysis.

The LexisNexis HPCC system is designed to completely remove the barriers to high complexity, strong data analysis. However to leverage this capability the analysis code has to be entirely ported into the ECL layer so that the optimizer can move the code down into the data servers. This fact actually produces an extremely simple and accurate way to characterize the engineering decision involved:

**You either have to move the code to the data or move the data to the code.**

The LexisNexis HPCC system, utilizing ECL, has been designed on the premise that the dataset is huge and the code is relatively simplistic. Therefore, moving code into the ECL system generally results in performance improvement that is measured in orders of magnitude. The cost is that the algorithms have to be ported out of their existing format. For algorithms that don't yet exist, the modular, structured and data centric ECL language will actually speed up the development of the algorithm.

*Case Study: Non-obvious relationship discovery*

> The original implementation of relatives within the LexisNexis Accurint product was performed in the application layer. Going to three levels of relationship involved hundreds and sometimes thousand of fetches per query. The result, even on high performance hardware, was that many relationship trees would fail the systems default timeout.
>
> The logic was moved into an ECL process that simultaneously computes the associates and relatives for every person in the US. The process at one point is evaluating seven hundred and fifty billion simultaneous computations stored in a sixteen terabyte data file. The result is presented across the SOAP layer as a simplistic relatives table which can now be delivered up with a sub-second response time. Further these relationships now exist as facts which can themselves be utilized in other non-obvious relationship computations.

*3) In such a situation, the main strength of the analysis will have been performed during the record selection process.*

The work of LexisNexis in the fields of law enforcement and anti-terrorism has all been achieved using the processing model below:

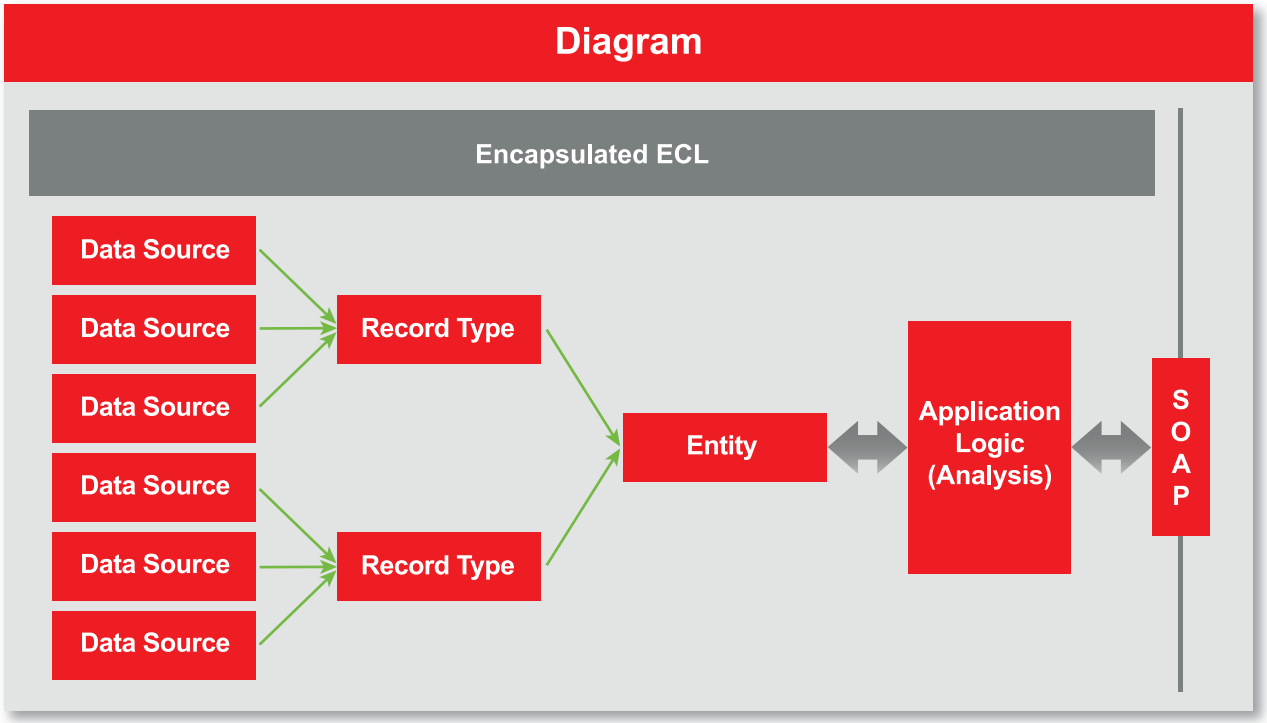


**Figure 5**

It should be noted that some third-party data analysis tools[4] are low complexity by this definition. In this situation LexisNexis has an integration mechanism whereby we can actually execute within the ECL environment. In this situation, HPCC acts as an operating environment to manage the parallelization of the third-party software. The lift provided by HPCC will be directly proportional to the number of blades the system is running upon. For those third parties that do not offer a parallel blade solution, this is a significant win. For those that do have a server farm solution, the HPCC architecture will represent an improvement in the ease of system management for a modest reduction in performance.

Note that high complexity analytic tools cannot be integrated in this manner.

## Conclusion

This paper has described, in stages, the premises behind the RDBMS engines, the data architectures of those engines and the problems associated with deploying either to the problem of high volume, real world, disparate data integration and analysis. It then detailed a platform architecture that allows teams to integrate large volumes of data quickly and efficiently while retaining third-party tools for analysis. Finally, it documented the pitfalls of external data analysis that is either strong or complex and outlined with case studies and architecture that solves this problem in a maintainable and efficient manner.

*4) Entity extraction being a good example.*

**For more information:**
**Website: http://hpccsystems.com/**
**Email: info@hpccsystems.com**
**US inquiries: 1.877.316.9669**
**International inquiries: 1.678.694.2200**

About HPCC Systems

HPCC Systems from LexisNexis® Risk Solutions offers a proven, data-intensive supercomputing platform designed for the enterprise to solve big data problems. As an alternative to Hadoop, HPCC Systems offers a consistent data-centric programming language, two processing platforms and a single architecture for efficient processing. Customers, such as financial institutions, insurance carriers, insurance companies, law enforcement agencies, federal government and other enterprise-class organizations leverage the HPCC Systems technology through LexisNexis® products and services. For more information, visit http://hpccsystems.com.

About LexisNexis Risk Solutions

LexisNexis® Risk Solutions (http://lexisnexis.com/risk/) is a leader in providing essential information that helps customers across all industries and government predict, assess and manage risk. Combining cutting-edge technology, unique data and advanced scoring analytics, Risk Solutions provides products and services that address evolving client needs in the risk sector while upholding the highest standards of security and privacy. LexisNexis Risk Solutions is headquartered in Alpharetta, Georgia, United States.