White Paper

HPCC Systems: Models for Big Data

LexisNexis®

Table of Contents

Models for Big Data 3
Structured Data 3
Text (and HTML) 6
Semi-Structured Data 6
Bridging the Gap – The Key – Value pair 7
XML – Structured Text? 8
RDF 10
Data Model Summary 11
Data Abstraction – An Alternative Approach 11
Structured Data 12
Text 12
Semi-Structured Data 13
Key-Value Pairs 13
XML 14
RDF 14
Model Flexibility in Practice 15
Conclusion 16

Models for Big Data

The principal performance driver of a Big Data application is the data model in which the Big Data resides. Unfortunately most extant Big Data tools **impose** a data model upon a problem and thereby cripple their performance in **some** applications[1]. The aim of this paper is to discuss some of the principle data models that exist and are imposed; and then to argue that an industrial strength Big Data solution needs to be able to move between these models with a minimum of effort.

As each data model is discussed various products which focus upon that data model will be described and generalized pros and cons will be detailed. It should be understood that many commercial products when utilized fully will have tricks, features and tweaks designed to mitigate some of the worst of the cons. Notwithstanding, this paper will attempt to show that those embellishments are a weak substitute for basing the application upon the correct data model.

Structured Data

Perhaps the dominant format for data throughout the latter part of the twentieth century and still today the dominant format for corporate data is the data table[2]. Essentially structured data has records of columns and each column has a value the meaning of which is consistent from record to record. In such a structure, vehicles could be represented as:

Make	Model	Color	Year	Mileage
DODGE	CARAVAN	BLUE	2006	48000
TOYOTA	CAMRY	BLUE	2009	12000

Many professional data modelers would immediately wince when handed the above. This is because the proposed structure did not allow for a very common feature of structured databases: normalization. Normalization is a topic which can (and does) fill many university Computer Science courses. For brevity[3] we shall define normalization as occurring when two or more columns are found to be dependant and the combinations of field values occurring are placed into a separate table that is accessed via a foreign key.

*1. In fact, the performance is **so** crippled that the application just "doesn't happen". It is impossible to 'list all the things that didn't happen' although our paper "Math and the Multi-Component Key" does give a detailed example of a problem that would take 17 hours in a key-value model and which runs at about 60 transactions per second in a structured model. It is easy to imagine that the key-value version would not happen.*

2. This is changing very rapidly in some areas. The other models are tackled later; but it is probably still true that today this one deserves to be tackled first.

3. A rather longer and more formal treatment is given here: http://en.wikipedia.org/wiki/Database_normalization

Thus looking at the above we might decide that the Make and Model of a car are related facts (only Toyota makes a Camry) – the other three columns are not. Thus the above could be defined as:

Vehicle Type	Color	Year	Mileage
1	BLUE	2006	48000
2	BLUE	2009	12000

Key	Make	Model
1	DODGE	CARAVAN
2	TOYOTA	CAMRY

The advantage of the above may not be immediately apparent. But we have reduced the width of the vehicle file by two strings at the cost of one fairly small integer column. There is also a secondary table but that will have relatively few entries (hundreds) compared to the millions of entries one may have in a vehicle file. Better yet we can add new information about each vehicle type (such as Weight, Horsepower *et cetera*) in one place and all the other data referring to it is 'automatically right'.

The real power of structured data becomes apparent at the point you wish to **use** it. For example; if you wanted to find out how many miles travelled by all vehicles you simply need to sum the fourth column of the first table. If you want to find the ratio of vehicle colors in any given year – or trend it over time – a simple aggregate on the second and third table suffice.

Things start to get a little more interesting if you want to count the number of miles driven in cars for a given manufacturer for a given model year. The issue is that the key in the vehicle file no longer identifies manufacturer; rather it identifies manufacturer AND model. Put another way the required result wants information spanning two tables. To get this result efficiently some clever tricks need to be performed. If you use a high level language such as SQL[4] and if the database administrators predicted this query then the cleverness should be hidden from you. However, under the hood one of the following will be happening:

1) The Tables can be JOINed back together to create a wider table (the first table) and then the required statistic is an aggregate on three columns.
2) The aggregate can be formed on the larger table to produce summaries for manufacturer and model and then that table can be joined to the vehicle type table and a second aggregate performed
3) The vehicle type table can be scanned to produce SETs of vehicle-type ID for each manufacturer, these sets can then be used to produce a temporary 'manufacturer' column for the main table which is then aggregated.
4) Perhaps something else that an SQL vendor found that works in some cases

4. Structured data does NOT need to imply SQL – but SQL is, without doubt, the leading method through which structured data is accessed.

As can be seen, aggregating and querying data across the entire database can get very complicated even with two files; as the number of data files grows then the problem becomes exponentially[5] more complex. It is probably fair to say that one of the key indicators of a quality of an SQL optimizer is the way it handles cross-file queries.

The other key feature of an SQL system to keep in mind is that in its purest form the performance is dismal. The reason is simply that each query, however simple, requires the entire dataset to be read. As the data becomes larger this becomes prohibitive. For this reason almost every SQL system out there has the notions of KEYs (or Indexes) built in. A key is an access path that allows records matching on one or more columns to be retrieved without reading the whole dataset. Thus, for example, if our vehicle dataset had been indexed by Color it would be possible to retrieve all of the red vehicles without retrieving vehicles of any other Color.

A critical extra feature of most SQL systems is the concept of multi-component keys; this is simply a key that contains multiple columns. For example, one could construct a key on Color / Year. This allows for two extremely fast access paths into the vehicle file: Color and Color/Year[6]. The criticality of this feature comes from an understanding of how the Color/Year value is found. The index system does NOT have to fetch all of the records with a matching color looking for a matching year. Rather the index itself is able to resolve the match on both color and year in one shot. This can give a performance lift of many orders of magnitude[7].

Naturally a performance lift of orders of magnitude is always going to come at a price; in this case the price is the double whammy of build cost and flexibility. For these multiple-component keys to be useful they have to exist; in order to exist you have to **predict** that they are going to be useful and you have to have spent the time building the key. The visible manifestations of this cost are the high price attached to good system DBAs and also the slow data ingest times of most SQL systems.

The final, though obvious, flaw in the structured data model is that not all data is structured. If data is entered and gathered electronically then it has probably been possible to force the data to be generated in a model that fits the schema. Unfortunately there are two common reasons for data not existing easily in this model:

1) The data is aggregated. In the examples I have used vehicles as an example of a structured data file. In fact vehicle data is gathered, at least in the USA, independently by each state. They have different standards for field names and field contents. Even when they have the same column-name that is supposed to represent the same thing, such as color, they will have different words they use to represent the same value. One state might categorize color using broad strokes (Blue, Green, Red) others can be highly detailed (Turquoise, Aqua, Cyan). Either the data has to be brutalized into a common schema (at huge effort and losing detail), or left un-integrated (making genuine cross-source queries infeasible) or the schema has to get extremely complicated to capture the detail (making queries extremely complex and the performance poor).
2) Data cannot be readily split into independent columns. Consider any novel, white paper or even email. It doesn't represent a readily quantified transaction. It can contain detail that is not known or understood a-priori; no structured model exists to represent it. Even if a structured model were available for one particular record (say Pride and Prejudice) then it would not fit other records (say War and Peace) in the same file.

In closing the structured data section we would like to mention one important but often overlooked advantage of SQL – it allows the user of the data to write (or generate) queries that are related to the problem (question being asked) rather than to the vagaries of the underlying data. The user only has to know the declared data model – the data has already been translated into the data model by the time the user is asking questions.

5. Many people use 'exponentially' as an idiom for 'very'. In this paper the term is used correctly to denote a problem that grows as a power of the scaling factor. In this case if you have three choices as to how to perform a join between two files, then between 10 files you have at least 3^10 = 59,049 choices. In fact you have rather more as you can also choose the order in which the files are processed; and there are 3,628,800 orders of 10 files giving a total of 2.14x1011 ways to optimize a query across 10 files.

6. Because of the way most Key systems work it does not in general provide a fast access path for Year only. Some good structured systems can access year quickly if the earlier component has low cardinality.

7. Again, this is not an idiomatic expression. If a field is evenly distributed with a cardinality of N then adding it as a component of a key in the search path reduces the amount of data read by a factor of N. Thus if you add two or three fields each with a cardinality of 100 then one has produced a system that will go 4-6 orders of magnitude (10000-1000000x) faster.

Text (and HTML)

At the complete opposite end of the scale you have text. Within text the data has almost no explicit structure and meaning[8]. A slightly more structured variant is HTML which theoretically imposes some form of order upon the chaos but in reality the generations of evolving specifications and browsers which do not comply to any specifications mean that HTML documents cannot even be assumed to be well formed. The beauty of text is that anyone can generate it and it often contains useful information. The World Wide Web is one example of a fairly large and useful text repository.

The downside of text, at least at the moment, is that it has no computer understandable semantic content. A human might be able to read it, and a thought or impression or even sets of facts might be conveyed quite accurately – but it is not really possible to analyze the text using a computer to answer meaningful questions[9].

For this reason query systems against Text tend to be very primitive analytically. The most famous interface is that presented by the internet search engines. Typically these will take a list of words, grade them by how common the words are, and then search for documents in which those words are prevalent[10]. This work all stems from term and document frequency counts first proposed in 1972[11].

An alternative approach, used in fields where precision is more important and the users are considered more capable, is the Boolean search. This method allows sophisticated expressions to be searched for. The hope or presumption is that the query constructor will be able to capture 'all the ways' a given fact might have been represented in the underlying text.

Text databases such as Lucene are extremely flexible and require very little knowledge of the data to construct. The downside is that actually extracting aggregated information from them is the field of data-mining; which is generally a PhD level pursuit.

Semi-Structured Data

Almost by definition it is impossible to come up with a rigorous and comprehensive definition of semi-structured data. For these purposes we shall define Semi-Structured data as that data which **ought** to have been represented in a structured way but which wasn't.

Continuing our vehicle example; consider this used car classified:

> 2010 Ferrari 599 GTB HGTE, FERRARI APPROVED, CERTIFIED PRE-OWNED WITH WARRANTY, Don't let this exceptional 599 GTB HGTE pass you by. This car is loaded with desirable options such as black brake calipers, carbon ceramic brake system, heat insulating windscreen, front and rear parking sensors, full recaro seats, and a Bose hifi system., The HGTE package comes with a retuned suspension consisting of stiffer springs, a thicker rear anti-roll bar, returned adjustable shocks, and wider front wheels. In addition, the car sits 0.4 of an inch lower to the ground and has a retuned exhaust note, and is fitted with the stickier Pirelli P Zero rubber. Inside, the HGTE package includes every possible carbon-fiber option., This vehicle has been Ferrari Approved as a Certified Pre Owned vehicle. It has passed our 101 point inspection by our Ferrari Factory trained technicians., 100% CARFAX, CERTIFIED!!!

8. Many people refer to text as 'unstructured data'. I have generally avoided that term as good text will usually follow the structure of the grammar and phonetics of the underlying language. Thus text is not genuinely unstructured so much as 'structured in a way that is too complex and subtle to be readily analyzed by a computer using the technology we have available today.' Although see the section on semi-structured data

9. Of course, people are researching this field. Watson is an example of a system that appears to be able to derive information from a broad range of text. However if one considers that 'bleeding edge' systems in this field are correct about 75% of the time it can immediately be seen that this would be a very poor way to represent data that one actually cared about (such as a bank balance!)

10. Google pioneered a shift from this model; the 'page ranking' scheme effectively places the popularity of a page ahead of the relevance of the page to the actual search. Notwithstanding the relevance ranking of a page is still computed as discussed.

11. Of course one can build multi-billion dollar empires by 'tweaking' this formula correctly

All of the text is free form and without pre-processing it would need to be stored as a variable length string. None of the queries of our structured section could be applied to this file. As it stands one would be limited to one or more 'string find' or 'search' statements in the hope of keying to text in the description. In an aggregated database this restriction results in a crucial shift of responsibility. The person asking the question (or writing the query) now has to understand all the possible ways a given piece of information might have been represented[12]. Of course, many internet search engines operate exactly this way.

However, with even some simplistic pre-processing it should be possible to turn the text above into:

Make	Model	Year	Other
Ferrari	599 GTB HGTE	2010	FERRARI APPROVED, CERTIFIED PRE-OWNED WITH WARRANTY, Don't let this exceptional 599 GTB HGTE pa...

Of course if one were clever and dedicated then even more information could be taken. In general, one could write a suite of routines to parse car classifieds and build up a fairly strong data model. This process of transforming from unstructured to semi-structured text requires up-front processing but the pay-off is that structured techniques can be used upon those facts that have been extracted; with all of the inherent performance and capability advantages that have been mentioned. Exactly how much data is 'parsed out' and how much is left unparsed is (of course!) ill defined; however for the process of classification we shall define semi structured data as data in which one or more (but not all) of the fields have been parsed into columns.

Thus far the presumption has been that the data is starting unstructured and that we are 'creating' some fields from nothing. In fact many structured databases contained semi-structured components. At the most extreme these are represented as BLOBs inside an SQL database. Rather more usefully there may be text strings which appear free-format but which realistically have a limited range of values. These are often the 'comment' fields stored in relational databases and used as a catch-all for fields that the modelers omitted from the original data model and which no-one has ever fixed.

Bridging the Gap – The Key – Value pair

Whilst the processing paradigm of Map-Reduce is much vaunted, the equally significant enabler, the Key-Value pair, has gone relatively unheralded. In the context of the foregoing it may be seen that it is the data model underlying Map-Reduce (and thus Hadoop) that is actually the fundamental driver of performance. In fact those with a sharp eye may notice that key-value pairs derive their power from their position part-way between semi-structured and textual data.

12. Or not care; if one is just 'surfing the web' then as long as the page offered is 'interesting enough' then one is happy – whether or not it was the 'best' response to the question is immaterial.

A file of key value pairs has exactly two columns. One is structured – the KEY. The other, the value, is unstructured – at least as far as the system is concerned. The Mapper then allows you to move (or split) the data between the structured and unstructured sections at will. Thus our vehicle table could be:

Key	Value
Dodge | Caravan	Blue | 2006 | 48000
Toyota | Camry	Blue | 2009 | 48000

The reducer then allows data to be collated and aggregated **provided** it has an identical key. Thus with the Mapper I used above it would be very easy to perform statistics aggregated by Make & Model. Aggregating by Color & Year would be extremely painful; in fact the best idea would probably be to re-map the data so that Color and Year were the key and then perform a different reduce. The advantage of this system over a keyed and structured SQL system is that the sort-order required is defined at query time (no prediction is required) and until that sort order is used then there is no overhead: in other words data ingest is quick.

In pure map reduce any query requires at least one full table scan of every file involved; Hadoop has no keys. Of course, a number of add-ons have been written that allow Key-Value pairs to be stored and quickly retrieved. These function in a very similar way to SQL keys except that the keys can only have one component, they must contain the payload of the record, and there is no existing support for normalization or the concept of foreign keys. In other words, if you have one or two access paths there is no problem – but you cannot access your data using a wide range of queries.

It should now be evident why Key-Value (and Map-Reduce) has been able to achieve general purpose popularity. It can read data in either structured, text or semi-structured form and dynamically translate it into a semi-structured data-model for further processing. The restrictions within the current implementations are:

1) A single map-reduce only supports one semi-structured datamodel
2) Multiple map-reduces require multiple-maps (and thus reprocessing of the data)
3) Either: all queries require a full data scan (Hadoop) OR all data-models require their own (potentially slimmer) copy of the data (Hive etc)
4) The work of constructing the semi-structured model is entirely the responsibility of the programmer
5) If the data is at all normalized then optimizing any joins performed is entirely the responsibility of the programmer

XML – Structured Text?

It is really incorrect to speak of XML as a data model. It is really a data transmission format that allows a plethora of different data models to be defined and utilized. In particular many if not all of the preceding data models can be expressed and utilized from within XML. That said, XML databases are usually thought of in the context of the storage of hierarchical databases where each record has a different but conformant format. An example might be a personnel database where the person is the outer container and various nested details are provided about that person, some of which themselves have nested details. Another very common usage of an XML database is document storage where the annotations upon and structure of the document is expressed in XML. In this model it is very common for the XML to also have an HTML rendering generated via XSLT.

From the perspective of data access and summarization the key to an XML document is epitomized by the name of the most common query language for them: XPATH[13]. Each element of an XML document is identified by a path which defined from the root; the 'navigation path' to the individual element or elements. Along the route various filters and conditions based upon the elements and tags encountered along the route may be executed; it is even possible to include data derived from routes into other parts of the data within the expressions.

Thus the pain of any given query; both in terms of encoding and execution is directly proportional to the extent to which that query can be expressed as a linear path. Simple fetches of one or more columns are easy; queries that rely upon the relationship between the columns of a record, or between columns of different records are harder to express and execute[14]. In short, whilst XML supports almost any data model; its features, search syntax, and performance footprint encourage, if not mandate, a hierarchical data model. Hierarchical data models are 'great if it is true'. If the data naturally fits a model whereby every element has a single parent and that the children of different parents are logically independent then the model is efficient in both usage and expression. If, however, it is the relationship between child elements that is interesting then the imposition of a rigid hierarchy will simply get in the way[15]. One might conclude that XML/XPATH provides most of the features of SQL but with an antithetical data model.

A hierarchical view of vehicles might be:

Parent	Child	
Vehicle One	**FactType**	**Value**
	Make	**Dodge**
	Model	**Caravan**
	Year	**2006**
	Color	**Blue**
	Mileage	**48000**
Vehicle Two	**FactType**	**Value**
	Make	**Toyota**
	Model	**Camry**
	Year	**2009**
	Color	**Blue**
	Mileage	**12000**

13. XQuery has probably surpassed XPATH in more modern installations

14. A good XML database such as MarkLogic will allow for optimization of complex queries provided the access paths can be predicted and declared to the system.

15. Within the academic literature there have been numerous attempts to extend XPATH towards more relational or graph-based data.

RDF

Given the characterization of XML data as 'typically' hierarchical it is probably wise to mention one extremely non-hierarchical 'data model' based upon XML which is RDF (Resource Description Framework). Like many things designed by committee this standard has many features and purposes; however at the data model level it is a method of describing data entirely using triples. Put another way RDF is a method of describing data as a collection of typed relationships between two objects. Our vehicle file could be represented using:

Object 1	Relationship	Object 2
Vehicle 1 [16]	MadeBy	Dodge
Vehicle 1	ModelName	Caravan
Vehicle 1	Color	Blue
Vehicle 1	Mileage	48000
Vehicle 1	Year	2006
Vehicle 2	MadeBy	Toyota
Vehicle 2	ModelName	Camry
Vehicle 2	Color	Blue
Vehicle 2	Mileage	12000
Vehicle 2	Year	2009

Viewed at the level of a single entity this appears to be a painful way of expressing a simple concept. The power comes from the fact that the objects in the third column are 'first class' objects; exactly the same as those on the left. Therefore this table, or another, could be expressing various facts and features of them. Viewed holistically data is no longer a series of heterogeneous tables that may or may not be linked; rather it is a homogenous web of information.

While not mandated by the data model RDF is often queried using declarative languages such as SPARQL. These languages take advantage of the simplicity and homogeneity of the data model to produce elegant and succinct code. The challenge here is that every query has been transformed into a graph query against a graph database. These queries represent some of the most computationally intensive algorithms that we know. To return all 'Blue Toyota Camry' we need to retrieve three lists – all blue things, all Toyota things, all Camry things and then look for any objects appear on all three huge lists. Compared to a structured data multi-component fetch, we have turned one key fetch into three; each of which would be at least two orders of magnitude slower.

16. The labels I am using here are for illustration; RDF contains a range of specifications to ensure unique naming, integrity etc

Data Model Summary

Eight pages is woefully inadequate to do justice to the fruits of thirty years or research; there are features, advantages and disadvantages to all of the preceding which are not even mentioned in this text. I have also ignored some popular binary level models – notably the multimedia and sensory data models. However, the aim was not to argue for one model over another; rather it was to argue that there is a broad range of highly divergent models out there. Further we hope to have shown that whilst one model almost certainly is perfect for one application it is relatively unlikely that one single model suits **all** of the applications that could (or should) utilize a particular dataset.

Big Data owners typically adopt one of three approaches to the above issue:

1) Adopt one model that excels in their core need and either accept that alternative forms of application run slowly; or simply ignore alternative forms of data usage. As long as the 'other uses' weren't really that important; this is probably optimal.

2) Adopt a model that isn't **too** bad across a range of potential applications – typically this results in 'lowest common denominatorism' – but it allows the data to be used to its fullest extent. This gets the most out of your data – but the cost of hardware – or lack of performance may be an issue.

3) Replicate some or all of the data from a core system into one or more data-marts; each in a slightly different data model. If the cost of maintaining multiple systems, and training staff for multiple systems can be justified then at an 'external' data level this appears optimal.

Data Abstraction – An Alternative Approach

A driving goal behind the HPCC[17] system was the abstraction of the data model from both the language and the system. As each of the previous data models was discussed the 'typical query language' was discussed alongside the model. The HPCC system language, ECL, allows all of the supported data models to be queried using the same language. Further most, if not all, of the previous data models have an engine built around them that is designed for one particular data model. HPCC offers two different execution engines each of which is designed to support each of the data models. Finally, and most importantly, much of the technology built into the HPCC is focused upon the translation of one data model to another as efficiently as possible – in terms of both programmer and system resources.

Of course, in a whitepaper such as this, it is always easy to conclude 'we do everything better than everyone without any compromises'; it is much harder to develop a system that renders the claim true. The truth is that HPCC has been used to implement all of the above data models and in some aspects[18] we beat all the existing systems we have been measured against; of course any system that specializes in only one data model will usually have some advantage within that data model[19]. Where we believe HPCC is untouchable is in its ability to natively support a broad range of models and in the speed with which we can translate between models.

As an aid to further research this paper will now review some of the system features available for each of the stated data models, the extent of the work done, and some features available for translation in and out of the data model.

17. http://en.wikipedia.org/wiki/HPCC

18. Usually including performance

19. Usually speed of update or standards conformance

Structured Data

The ECL[20] handling of structured data is based upon the RECORD structure and corresponding DATASET declarations. ECL has a huge range of data types from INTEGERs of all sizes from 1 to 8 bytes, packed decimals, fixed and variable length strings, floating point numbers and user defined types. ECL also supports variant records and arrays of structured types[21]. Like SQL, columns are accessed by name and type conversions occur to allow columns to be compared and treated at a logical level (rather than low-level). Unlike SQL it is also possible to define an ordering of records within a file; this allows for records to be iterated over in sequence. ECL also has NORMALIZE and DENORMALIZE functions built in so that questions regarding 'just how much to normalize data' can be changed and modified even once the system is in place.

By default ECL files are flat and they have to be fully scanned to obtain data (like Hadoop). However ECL also has the ability to create indexes with single or multiple components and it also allows for variant degrees of payload to be imbedded within the key. Thus an ECL programmer can choose to have a large number of highly tuned keys (similar to SQL) or one or two single component keys with a large payload (like Key-Value) whichever they prefer (including both[22]).

There are many, many ECL features to support structured data but some of the most important are PROJECT, JOIN (keyed and global variants) and INDEX. Simple ingest of structured data is handled via DATASET which can automatically translated from flat-file, CSV[23] and well structured XML. More complex (and partial) ingest of other data models will be handled under the section 'Semi-Structured Data'.

Text

ECL handles Text in two different ways depending upon requirements. At the most basic level variable length strings[24] are native within ECL and a full set of string processing functions are available in the Std.Str module of the ECL standard library. For text which needs to be 'carried around' or 'extracted from' this is usually adequate.

For text that needs to be rigorously searched, an ECL programmer typically utilizes an inverted index. An inverted index is a key that has a record for every word in a document and a marker for where in the document the word occurs[25]. This index can then be accessed to perform searches across documents in the manner described in the text section. Our 'Boolean Search' add-on module exemplifies this approach. At a low level ECL has two technologies, global smart stepping and local smart stepping[26], to improve the performance of this form of search.

Text can be brought into ECL as a 'csv' dataset; if it currently exists as a series of independent documents then a utility exists to combine that into one file of records. It will be covered in greater depth in the semi-structured section but it should be noted that ECL has an extremely strong PATTERN definition and PARSE capability that is designed to take text and extract information from it.

20. http://en.wikipedia.org/wiki/ECL,_data-centric_programming_language_for_Big_Data

21. For those familiar with COBOL this was a method of having narrower records whereby collections of fields would only exist based upon a 'type' field in the parent record

22. Our premier entity resolution system uses multi-component keys to handle the bulk of queries and falls back to a system similar to key value if the multi-components are not applicable.

23. Referred to as 'comma separated variable' although there are many variants; most of which don't include commas!

24. In standard ASCII and UNICODE formats

25. There may be other flags and weights for some applications

26. When accessing inverted indexes naively you need to read every entry for every value that is being searched upon; this can require an amount of sequential data reading that would cripple performance. 'Smart stepping' is a technique whereby the reading of the data is interleaved with the merging of the data allowing, on occasions, vast quantities of the data for one or more values to be skipped (or stepped) over. The 'local' case is where this is done on a single machine; 'global' is the case where we achieve this even when the merge is spread across multiple machines.

Semi-Structured Data

We firmly believe that the next big 'sweet spot' in Big Data exists in Semi-Structured data. The action of turning data which is otherwise inaccessible into accessible information is a major competitive differentiator. ECL has extensive support at both a high and low level for this process.

Firstly, at a high level, ECL is a data **processing** language, not a storage and retrieval language. The two execution engines (one batch, one online), coupled with PERSIST, WORKFLOW and INDEX capability all push towards a model where data goes through an extensive and important processing stage **prior** to search and retrieval.

Secondly, at a lower level, the ECL PARSE statement makes available two different text parsing capabilities that complement each other to extract information in whichever form it is in. One capability is essentially a superset of regular expressions; wrapped in an easy-to-read and re-use syntax. This is ideal where the data is highly unstructured[27] and particular patterns are being 'spotted' within in. Examples applications include screen scraping and entity extraction. The other capability follows a Tomita[28] methodology; it is designed for the instance where textual document follows a relatively rigorous grammar[29]. This is suitable for rapidly processing text which is expected to be well formed.

Thirdly, again at a higher level, the ECL support for Text and Structured data are BOTH built upon the same ECL record structure. Thus, once some information has been extracted from the text, it can still reside in the same record structure awaiting additional processing. Further, as keys can contain arbitrary payloads, and as joins can be made across arbitrary keys, it is even possible to combine structured and inverted index fetches within the same query.

Key-Value Pairs

Key-Value pairs are a degenerative-sub case of normal ECL batch processing. If required a Key-Value pair can be declared as:

```
KVPair := RECORD
     STRING KeyValue;
     STRING Value;
     END;
```

The map-reduce paradigm can then be implemented using:

1) PROJECT/TRANSFORM statement returning a KVPair – this is equivalent to a Mapper
2) DISTRIBUTE(KVPairFile,HASH(KeyValue)) – the equivalent of the first stage of the Hadoop shuffle
3) SORT(DistributedKVPairFile,KeyValue,LOCAL) – second stage of Hadoop shuffle
4) ROLLUP(SortedPairFile,KeyValue,LOCAL) – the reduce

This usage of ECL is 'degenerative' insofar as data can be distributed, sorted and rolled up using any of the fields of a record, any combination of fields of a record and even any expressions involving combinations of fields of a record. Thus the 'normal' ECL case only has values[30] and any number of different keys 'appear upon demand'.

27. Specifically the case where linguistic rules of grammar not followed uniformly and thus the data really is 'unstructured'.

28. A brief treatment of these is given here: http://en.wikipedia.org/wiki/GLR_parser

29. This case works particularly well (and easily) in the case where the text being parsed is generated against a BNF grammar

30. Although it has any number of values; not just one per record

XML

XML probably represents the **biggest stretch** for ECL in terms of native support of a data model.

ECL does have very strong support for XML which is well specified and gently nested. Specifically it has:

1) XML on a dataset to allow the importing of XML records that can be reasonably represented as structured data
2) XMLPARSE to allow the extraction of sub-sections of more free-form XML into structured data
3) Extensive support for child datasets within record structures to allow some degree of record and file nesting on a hierarchical basis
4) An XML import utility that will construct a good ECL record structure from an XML schema.

Unfortunately there are XML schemas out there that really do not have sane and rational structural equivalents. More specifically; they may have equivalents but they are not the best representation of the data. For this reason we have been actively researching the construction of an XPATH equivalent within ECL. This essentially extends upon our work supporting arbitrary text processing and utilizes our ability to have multiple component indexes in an inversion and to blend the results of inverted and structured index lookups. The result is the ability to store XML documents in an extremely efficient shredded form and then process XPATH queries using the index prior to performing the fetch of those parts of the document required[31].

RDF

Much like Key-Value pairs RDF is simply a degenerative[32] case of standard ECL processing. Pleasingly the global smart stepping technology developed for Text searching can also be utilized to produce relatively rapid RDF query processing. Unfortunately being 'relatively quick' does not avoid the fundamental issue that graph problems are extremely compute intensive and the transformation of **all** queries into graph queries loses a lot of academic elegance once it is appreciated that we are transforming **all** queries into **graph** queries that we **cannot**, in general, solve.

From an execution standpoint ECL has an immediate solution that yields many orders of magnitude performance[33] improvement. You do not restrict yourself to triples; you allow the data store to be an arbitrary n-tuple. By doing this **each** fetch returns much less data (reduced by the cardinality of the fourth and subsequent columns) and the joins required are reduced by the products of those reductions[34]. This performance boost is achieved for the same reason that multi-component keys yield staggering lift over single component keys (see the description of structural data)[35].

The downside is that the query programmer now has to adapt their coding to the more heterogeneous data layout. As an area of active research a second language KEL (Knowledge Engineering Language) has been proposed and prototyped[36] which allows the programmer to work in a homogenous environment but then translates the queries into ECL that in turn uses an execution optimal heterogeneous data model. More information will be made available as the project nears fruition.

31. Fuller details of this will be published in due course; probably accompanied by a product Module offering

32. This term is being used technically; not in the pejorative. ECL works naturally with any combination of N-tuples. Asserting everything must be triples (or 3-tuples) is one very simple case of that.

33. Again, this is a mathematical claim, not a marketing one

34. If we can find it – we could reference the Lawrence Livermore study here

35. Yoo and Kaplan from Lawrence Livermore have produced an excellent study and independent on the advantages of DAS for graph processing: http://dsl.cs.uchicago.edu/MTAGS09/a05-yoo-slides.pdf

36. A team led by LexisNexis Risk Solutions, including Sandia National Labs and Berkeley Labs has an active proposal for further development of this system.

Model Flexibility in Practice

If ECL is compared in a genuine 'apples to apples' comparison against any of the technologies here on their own data model it tends to win by somewhere between a factor of 2 and 5. There are a variety of reasons for this, and they differ from data model to data model and from competitor to competitor but the reasons usually include some of:

1) HPCC/ECL Generates C++ which executes natively; this has a simple low-level advantage over Java based solutions
2) The ECL compiler is heavily optimizing; it will re-arrange and modify the code to minimize operations performed and data moved
3) The HPCC contains over a dozen patented or patent-pending algorithms that give a tangible performance boost to certain activities – sorting and keyed fetch being the two most significant
4) Whilst ECL supports all of these data models it does not (always) support **all** of the **standards** contained in the languages that usually wrap these models. Sometimes standards written a priori in a committee room impose low-level semantics on a solution that result in significant performance degradation[37]
5) Whilst continually evolving, the core of the HPCC has been around for over a decade and has been tweaked and tuned relentlessly since creation
6) The ECL language has explicit support for some feature that needs to be manufactured using a combination of capabilities of the other system[38]. In this situation the extra knowledge of the intent that the HPCC system has usually gives a performance advantage.

The above is a pretty impressive list; and going 2-5x faster allows you to use less hardware, or process more data, or get the results faster or even a little bit of all three. However, a factor of two to five is not really **game-changing**, it just lowers the cost of playing[39].

Those occasions when ECL is game-changing, when it delivers performance which is one or more orders of magnitude faster than the competition, usually stem from the data model. External evidence and support for this claim can be gleaned from the websites of many of the solutions mentioned in this paper. They **all** claim, and cite examples, where their solution to a data problem produces a result three to ten times faster than their competitors. Can they **all** claim this without deception[40]? Yes; they are citing examples where a customer with a given problem switched to them and they were providing a more suitable data model to the underlying problem. It is quite conceivable that two data solution vendors could swap customers and **both** customers get a performance enhancement!

37. As a recent example: ECL provides two sets of string libraries, one which is UNICODE complaint, one of which is not. The non-compliant libraries execute five times faster than the compliant ones. That said; string library performance is not usually the dominant factor in Big Data performance.

38. Put simply – if you can write it in one line of ECL the ECL compiler knew exactly what you were trying to do – if you write the same capability in 100 lines of Java then the system has no high level understanding of what you were doing.

39. As an aside, the HPCC was one of the first systems available that could linearly scale work across hundreds of servers. As such it could often provide two orders of magnitude (100x) performance uplift over the extant single-server solutions; which clearly is game changing. For this paper, given the title includes 'Big Data', it is presumed that HPCC is being contrasted to other massively parallel solutions.

40. One subtle form of deception is 'measuring something else'; the overall performance of a system needs to include latency, transaction throughput, startup time, resource requirements and system availability. There are systems that are highly tuned to one of those; this is legitimate – provided it is declared.

Given the preceding paragraph one might naturally expect data owners to be swapping and changing vendors with very high frequency to continually benefit from the 'best model for this problem' factor. Unfortunately the reality is that many data systems have been in the same model for years (even decades) even once it is abundantly clear it is the wrong model. The reason, as described previously, is that changing **model** usually involves a change of language, vendor and format; all of which are high risk, high cost, and disruptive changes. The ECL/HPCC system allows for model-change without any of those factors changing. Of course, the **first** time ECL is used many of those changes **do** occur; but once within the HPCC system data can transition from one model to another on a case by case basis with zero disruption[41].

To understand all aspects of the ECL impact one does need to accept that zero disruption is **not** the same thing as zero effort. Data will generally come to an ECL system in a natural or native[42] model. For reasons of time and encouragement the data will usually be made available in the original model through the ECL system. This will garner any 'ECL is quick' benefits immediately but it **may** not be game-changing. At that point a skilled data architect (or aspiring amateur!) is needed that can explore the options for alternative data models for the data. This might be as big as a completely different core model or it might be a change of model as the data is used for certain new or existing processes. The exploration is performed alongside the service of the data using the extant model. Then, if and when these opportunities are found and the results are shown to be as good or better, then the game can be changed.

One note of caution needs to be sounded; giving people the ability to choose also gives them the ability to choose poorly. As the data models were discussed it should have become apparent that some of them were extreme, niche models and others were more general purpose and middle of the road. ECL is flexible enough to allow the most horrific extremes to be perpetrated in any direction. For this reason ECL operates best and produces its most dramatically good results when early design decisions are made by one or more experienced data personnel. If these are not available it is probably wise to **mandate** that the existing data model is adhered to; at least initially.

Conclusion

In many ways this entire paper is a conclusion; any one of the subject headings could have been the title for a 100 page paper. I am sure that as each of the data models was discussed, the zealots of the model could have listed a dozen extra advantages and detractors could list a dozen extra flaws. Notwithstanding the purpose of the descriptions was simply to detail that there **are** many alternative models and that they each have lists of pros and cons. Further, most of them have languages, standards, and vendors wrapped around them, designed to make data processing in that particular model as proficient as possible.

Next, under the title of data abstraction, we proposed that the optimal system would allow for the problem to dictate the model to be used; rather than the model used dictating the problems which are soluble. We then detailed how the ECL/HPCC system supports each of the data models mentioned and also the large array of features and tools provided to assist in the transition between models. It was also noted that whilst ECL/HPCC generally executed against a given model more efficiently than alternative implementations it did not always support all of the standards or language features associated with a given model.

Finally the paper ended by suggesting that ECL gains performance lift from general efficiencies but that for the performance lift to reach the level of dramatic then generally a data-model shift was required. It was noted that data-models can be shifted without ECL by changing vendors; the unique advantage of ECL is the ability to shift model (or even hybridize models) within the same language and system. This advantage reduces the cost and risk of model shift and therefore increases the chances of one occurring.

41. This is a mathematical zero, not a marketing one. ECL/HPCC supports all of these models and hybrids of them within the same system and code. Many, if not all, ECL deployments run data in most of these models simultaneously and will often run the same data in different models at the same time on the same machine!

42. Where a native model is an extremely un-natural model imposed upon the data by an earlier system

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.

LexisNexis®