19th ICCRTS

"C2 Agility: Lessons Learned from Research and Operations"

Paper 113

Issues in "Big-Data" Database Systems

Topics:
(3): Data, Information, and Knowledge

Jack Orenstein
Geophile, Inc.
61 Prospect Street
Needham MA 02492
USA
+1-781-492-7781
jao@geophile.com

Marius S. Vassiliou
Institute for Defense Analyses
4850 Mark Center Drive
Alexandria, VA 22311
USA
+1-703-887-8189
+1-703-845-4385
mvassili@ida.org

Point of Contact:

Marius S. Vassiliou
Institute for Defense Analyses
4850 Mark Center Drive
Alexandria, VA 22311
USA
+1-703-887-8189
+1-703-845-4385
mvassili@ida.org

ABSTRACT

"Big Data" is often characterized as high-volume, multi-form data that changes rapidly and comes from multiple sources. It is sometimes claimed that big data will not be manageable using conventional relational database technology, and it is true that alternative paradigms, such as NoSQL systems and search engines, have much to offer. However, relational concepts (although not necessarily current relational products) will still have an important role to play in building database systems that can support the performance, scalability, and integration demands of Big Data applications. To deal effectively with Big Data, we must consider many factors, including: number of datatypes, schema changes, data volume, query complexity, query frequency, update patterns, data contention and isolation, and system and database administration. Relational database technology has been very successful in dealing with these issues, albeit for a single, tabular data form. However, it has largely ignored the problem of integrating disparate and heterogeneous data sources, except in the most trivial ways. It is nonetheless the right starting point for research on big data systems. Significant changes may be needed, to the data model, to the query language, and certainly to physical database design and query execution techniques; but to ignore relational technology is to ignore over forty years of relevant research on data processing.

1. INTRODUCTION

"Big Data" is currently the subject of much discussion, and hyperbole, in the literature of government,[1] business,[2] and computer science[3]. Articles in the *Harvard Business Review* termed big data a "management revolution,"[4] and declared "data scientist" to be the "sexiest job of the 21st century."[5] There is no universally accepted definition of "Big Data," but it has frequently been characterized by the "three Vs" of Volume (large amounts of data; more than can be processed on a single laptop computer or server), Variety (data that comes in many forms, including documents, images, video, complex records, and traditional databases), and Velocity (content that is constantly changing, and arriving from multiple sources). To this, a fourth "V" is sometimes added, for "Veracity" (data of varying quality).[6] Some observers question whether or not big data is "revolutionary," or whether it is beneficial.[7]

Without entering into any of these debates, it seems clear that, for better or worse, government and business will, at least in some cases, need to contend with data of increasing volume, diverse provenance, and increased complexity. In this paper, we understand "big data" to mean "high-volume, complex, multi-source data." Such "big

[1] Executive Office of the President (2012)
[2] McAfee and Brynjolfsson (2012)
[3] Stonebraker (2013)
[4] McAfee and Brynjolfsson (2012)
[5] Davenport and Patil (2012)
[6] Berman (2013)
[7] Arbesman (2013)

data" does pose new challenges to database systems, and will require new research directions.

An important issue is the relevance of traditional relational database systems[8] for Big Data applications, particularly compared to NoSQL[9] ("Not SQL," or "Not Only SQL") systems. Our position is that relational concepts provide an excellent framework for the development of new systems supporting the creation of Big Data applications. Existing systems are not adequate because they do not scale well, and because integration with external data sources is so difficult. NoSQL systems are more open to this integration, and provide excellent scalability for the simplest queries and updates. The tradeoff seems to be that if one wants these scalability advantages, one is necessarily limited to very simple transactions, and that this is not simply a deficiency in current implementations[10].

2. CHARACTERIZATION OF DATABASE SYSTEMS

Database systems can be characterized via a number of important dimensions:

Data model: This denotes the set of primitives for structuring data. In relational databases, for example, the data model is the relational model, first enunciated by Edgar F. Codd[11], based on first-order predicate logic. In NoSQL databases, the data model is comprised of key/value pairs, where the value is often a hierarchically structured document. In search engines, the data model is a set of connected documents of many formats.

Schema: The schema describes how data in a database is organized, using the structures provided by the data model. In relational databases, for example the schema is a set of tables whose cells contain atomic values. Schemas may be highly constrained, as in relational databases, or provide minimal structuring, as in NoSQL systems.

Application Program Interfaces (APIs): These are the interfaces for connecting to the system, retrieving data, loading and modifying data, and performing configuration and maintenance. APIs can be provided in the form of function call interfaces in a variety of languages, or as function call interfaces conveyed by a network protocol.

Query language: In many database systems, APIs carry commands, specified in some query or search language, to the database. These commands represent the means by which users obtain the results they need from the database. Structured Query Language (SQL) is an example of a query language. Google-style search commands constitute another. Not all databases have a query language – some key/value stores provide only APIs. In many database systems, APIs carry commands, specified in some query or search language, to the database. These commands represent the means by which users

[8] Codd (1969); Codd (1970)
[9] Grolinger et al. (2013)
[10] Helland (2012)
[11] Codd (1969); Codd (1970)

obtain the results they need from the database. Structured Query Language (SQL) is an example of a query language. Google-style search commands constitute another. Not all databases have a query language – some key/value stores provide only APIs.

Procedural language: Some database systems provide a procedural language for specifying functions and procedures that are stored in a database, and executed by the database system itself. These invocations can be explicit, carried by some API call or invoked from a query; or they can be implicit, run by the database system, e.g. from a trigger. Many NoSQL systems use JavaScript as the procedural language. A common arrangement is for JavaScript functions to run under the control of the NoSQL system's implementation of the map/reduce algorithm. Relational procedural languages, such as Oracle's PL/SQL, embed SQL, and include language constructs for combining SQL query results with language constructs for processing individual rows, and fields within rows.

Authentication & Authorization: This denotes the set of methods for identifying users, and limiting access (for example, based on identity and IP address). Authentication and Authorization can limit the use of operations, schema objects, or combinations thereof.

Session: This establishes a context in which a user interacts with the database. In an implicit session, each API call is its own session. Such a session is *stateless*, because there is no state connecting one request by a user to the next. An explicit session can connect a sequence of API calls from one user, providing some guarantees about that sequence, for example transaction consistency and caching.

Durability: Data that is stored in the database, survives the application that created it, and can be shared by users of the database, is characterized as durable. APIs exhibit great variation in the sequence of actions that lead to durability: they can result in immediate durability following a modification, or eventual durability following a modification. Durability might also be ensured when instructed through a special API call, (e.g. “commit”).

Interactions of Concurrent Sessions: This describes how sessions interact with each other, if at all. For example, if one session is updating a data item, while another is reading the same item, what happens to each session? Does the reader see the state before or after the update? If two sessions update a data item concurrently, what happens to each session and what is the final value of the data item?

Table 1 provides a summary characterization, along the above dimensions, of three important classes of database systems: relational databases, NoSQL databases, and search engines.

Table 1: Characterization of Various classes of Database Systems
(for acronym definition, see Glossary)

	Data Model	**Schema**	**Application Program Interfaces (APIs)**	**Query Language**	**Procedural Language**	**Authentication and Authorization**	**Session**	**Durability**	**Interactions of Concurrent Sessions**
Relational Database System	Relational Data Model	A set of tables whose cells contain atomic values	Native API (e.g. Postgres's libpq), ODBC, JDBC, command-line interface	SQL. Also, SQL embedded in a procedural language for stored procedures	Stored procedural language embedding SQL.	ID Based on host operating system identities, or identities registered with the database, (depending on system). Authorization based on model of privileges associated with tables and views (i.e., table expressions).	Provides both stateless (auto-commit mode) and long-lived sessions	Immediate for auto-commit, and on commit otherwise.	ACID transactions, with varying levels of isolation, that can be set per connection.
NOSQL System	Key/value pairs, where the value is often a hierarchically structured document	Loosely enforced or absent	Key/value operations available in various language APIs; REST APIs are typically native.	Absent, or simple	Typically JavaScript.	Usually RFC 2617 (HTTP Authentication: Basic and Digest Access Authentication)	Stateless	Usually synchronous with updates	Not Applicable (because sessions are stateless)
Search Engine	Text documents in a wide variety of formats	The recognized document formats	Search requests encoded as URLs	Some search engines have query languages controlling ranking of documents, significance of various	Various languages which run as part of the document ingestion pipeline.	Various approaches	Stateless	Document ingestion done in a “pipeline” of processing steps. Documents become durable at	Not Applicable (because sessions are stateless)

				search terms, required proximity of search terms in the document, and other aspects of searching				different stages. However, in general, documents are ingested in batches.	

3. ISSUES ASSOCIATED WITH HIGH-VOLUME, COMPLEX, MULTISOURCE DATA

The profusion of high-volume, complex, multi-source data raises a number of issues for database systems, and exposes some important differences in how various types of database systems can deal with them. The issues include:

- Number of Datatypes
- Schema changes
- Data Volume
- Query Complexity
- Update Patterns
- Data Contention and Isolation
- System and Database Administration

We discuss each of these areas in turn below.

3.1 Number of Datatypes

Common datatypes include the following:

- Collections of records of simple values (numbers, short strings, date/time)
- Collections of hierarchically structured documents
- Geographic data
- Collections of text documents
- Collections of binary formats (e.g. images, audio)

There are many off-the-shelf systems that can handle subsets of these, but there is nothing that can handle all of these together. For example, relational databases do an excellent job of handling fixed-format records containing simple values, but their capabilities for handling text documents are usually unsatisfactory. Collections of binary data are usually organized in a file system, with metadata stored elsewhere. Integration is done at the application level. Hierarchically structured documents interoperate poorly with relational systems[12].

In general, big data applications need to organize collections of data of varying types in a single framework, leading to two main requirements. First, there must be a way to associate entities. For example, records in a database representing the legal status of a plot of land (ownership history, zoning, etc.) might need to be connected to a vector representation in a geographic information system (GIS) containing the boundaries of that plot. Relational databases are well suited for this purpose, as long as the records are

[12] Tahara & Abadi (2014)

within the same database. They are also useful when storing identifiers that are meaningful in other systems, such as a GIS. However, in this case, the reference can only be used at the application level, sacrificing many database capabilities, such as referential integrity, query expressiveness, and query optimization. Another difficulty in this area is that references may be embedded in different ways in different collections. A relational database may store a reference in a column of a table, while a NoSQL system might use a key/value pair in a JSON[13] document. There are currently no systems that can accommodate this difference, and treat both kinds of references in the same way.

Second, there must be a way to express queries across these collections. Ideally, this would be done at a high level, using a non-procedural language, with an optimizer making decisions about how to most efficiently implement a query. Querying across system boundaries at the application level is the only choice currently, but this results in systems that are brittle, subject to failure or unexpected behaviors when any of the component systems changes, (e.g. changing a public API, or an existing API starts operating more slowly due to internal indexing changes).

Big Data applications will typically need to accommodate many of the above datatypes simultaneously. One important consequence of this is that different types of information, with different representations, must be connected together. For example, documents concerning ownership of a plot of land over time may need to be integrated with vector overlays for utility lines, geologic surveys, etc. Relational databases are rather well suited for the metadata management associated with making these types of connections. However, relational databases are closed environments, and complete incorporation of various datatypes is difficult.

It seems unlikely that a single system will ever accommodate all of these collections of data. One obvious reason for this is the sheer volume of data in existing systems, and all of the human expertise and organizational processes that depend on these systems. It is easy to envision providing additional access to such systems, while the wholesale movement of data, applications, and procedures seems very unlikely. Another reason is that the boundaries of data collections tend to match organizational boundaries. In the far simpler world of business systems, in which relational databases are generally sufficient, there are databases for applications and for departments. There is only rarely a single corporate database that contains all data and supports all applications, even though this was the goal of the early database researchers and system builders.

Thus, the way forward appears to be in integrating existing systems. A necessary result of this approach is that there will be inconsistencies across system boundaries. To use an obvious example, taxpayer records may refer to recently deceased taxpayers. The revenue department will interact with that person as if still alive, until the news of that person's demise is accounted for. Similarly, with more complex datatypes, any system that aims to integrate across systems will need to deal with stale and inconsistent data.

[13] ECMA (2013)

Currently, all integration across data collections is done at the application level. Many questions about the design and implementation of database systems that can do this integration are wide open.

3.2 Schema Changes

Application programming is easier when one can assume a static set of entities, with each entity having a static set of attributes. Unfortunately, Big Data applications are characterized by a constantly growing set of data, potentially involving new schemas. Changes to the schema are painful because they can entail major changes in applications, further aggravated when very large data sets must adapt. This can result in system downtime and large costs.

Schemas may change in a number of ways. One of the simplest is not really a full schema change at all, but rather a situation where we are faced with new datasets that closely or exactly match an existing schema. An example of this is a set of new vehicle data for a new model year in the database of a state Department of Motor Vehicles (DMV). Data cleansing may be needed, perhaps with the help of an Extract-Transform-Load (ETL) system, but there are no major conceptual problems.

A somewhat more difficult case is when there are new datasets for existing entities, but with a different schema. For example, the new vehicle data may contain a new attribute for the degree of electric assist in the motor. Data may need to be restructured (again, with the help of an ETL system). There may also be problems arising due to differences in entity identification. For example, if data from the DMVs of multiple states need to be imported, each state may have its own scheme for identifying drivers (social security number, DMV-issued number, etc.). Data cleansing may be needed, e.g. to ensure that social security numbers are formatted identically before they are compared. Additional integration steps may be needed, e.g. to map DMV-issued numbers to social security numbers.

We may also be faced with situations where new entities are related but not identical to existing ones. Continuing with our vehicle example, suppose that DMV data must now be integrated with data from toll-booth collection systems such as *EZPass* in the northeastern United States. The schema of the toll-booth collection system may differ. Where the DMV system may key on the Vehicle Identification Number (VIN), the toll system may key on a license plate number captured, perhaps imperfectly, by a camera, and contain other different attributes. There may again be entity identification problems.

There may also be situations where the existing schema changes significantly. There may be new entities with new attributes, and obsolete entities with obsolete attributes. There may also be restructuring, with major changes in relationships, such as a one-to-one map becoming one-to-many. Imagine, for example, a situation where polygamy is legalized, and multiple spouses must now potentially be accommodated for a single person.

Different types of systems will deal with schema changes in different ways. In the relational approach, we may update schema, and migrate data to match, fixing applications to use the updated definitions. This will typically require a huge effort in planning and implementation. It will be painful, but for a limited time period.

A NoSQL system is schemaless, but this does not mean applications do not have to face hard problems. In the DMV example, applications must know about all possible versions of vehicle definition. Schema changes may be painful forever.

One possibility is to formally version the schema. This is not an idea that has been widely applied commercially. It was tried by the Israeli Defense Forces for soldier records—with each record being tied to a particular schema version.[14]

3.3 Data Volume

The difficulties in using, maintaining, and managing a database grow with its size. Many of these problems arise indirectly. The issues are discussed here by imagining a small database, and then examining the problems that arise as it grows. To start, suppose we have a tiny database, so small that the entire database fits into physical memory. (This database is still disk-based, because durability is still important, but a read never requires disk access.) There will be very fast queries, and very slow queries, depending on the complexity of the query, available indexes, etc. It is safe to say that any query will run much faster if disk access can be avoided. Making an update durable still requires disk writes, but most database systems use a journal so that the latency experienced by the application is minimal.

As the database grows, the database will no longer fit in memory. At this point, memory serves as a cache, and performance drops rapidly as the cache contains ever smaller portions of the database. Queries become input/output (IO)-bound, and the best execution plan for a given query will be the one that makes best use of the cache. For updates, durability latency stays the same (because a journal write is required, just as before), but the time to actually update the main part of the database from the journal will take more time and IO bandwidth. Some indexes may become unaffordable as a result of this increased IO burden, and this will cause some queries to slow down dramatically.

Data volume is not the immediate reason for introducing distribution, but some problems tend to associate with very large databases. Distribution is then introduced to cope with these problems, achieving scalability of some kind. Examples:

- **Read replicas:** There is a main database. Asynchronous replication is introduced to support read operations that can be satisfied with slightly stale data. This minimizes the IO load on the main database.

[14] Personal communication from a former software engineer in the Israeli Defense Forces

- **Vertical distribution:** Different data collections (e.g. tables), or sets thereof, can be placed on different machines. Obviously, this is only effective if operations tend not to require data on multiple machines. Otherwise, the cost of coordinating data access across machines, especially for transactional updates, will cancel out any gains in performance.

- **Horizontal distribution:** *Sharding* distributes the items in a single data collection across a set of machines. Unlike vertical distribution, this kind of distribution scales arbitrarily. As the database keeps growing, more shards can be introduced to handle the load. This technique is almost universal among NoSQL systems, and it works very well for scaling both single writes and single reads. If multiple operations need to be combined into a single transaction, then most developers have found it preferable to sacrifice transactions than performance. Also, complex operations, involving joins, or multiple indexes, will not benefit from sharding, and in fact will usually perform slower compared to a database on a single machine.

3.4 Query Complexity

NoSQL systems can adequately address the low end of the query complexity scale. Key/value lookup is cheap (a millisecond or less), regardless of data volume or use of distribution, and NoSQL systems are good at scalable key/value lookup. They are also good at exploiting parallelism for queries that fit the map/reduce paradigm, although almost anything else will need to be programmed at the application level. NoSQL systems are more open to dealing with complexity in datatypes, and entities outside the database.

Relational systems with SQL are better overall at dealing with complex queries. Query optimization is possible, and there is a rich set of data processing primitives. However, these apply only within the relational system itself. Capabilities for dealing with entities outside the database are rudimentary. Some early research in this area has been done in the context of the *Hadapt* project[15]. While not the main point, this work highlights how much still needs to be done in breaking down the walls dividing database systems from external data sources. Probably the best example of integration to date is from Postgres, which permits the integration of datatypes. This integration includes the mapping of functions to operator tokens reserved for this purpose, and for integration with the optimizer. However, this integration seems focused on relatively simple datatypes, e.g. spatial objects. Integrating search engine capabilities represents another level of complexity that would probably not work well with this datatype integration approach.

Also, as queries become increasingly complex in Big Data applications, several performance issues arise for relational systems. Joins can become very expensive, depending on data volume, the use of distributed systems, and the availability of suitable indices. This is compounded as joins also tend to multiply with complexity and the

[15] Tahara & Abadi (2014)

importation of new data sources. The proper selection of join order and join algorithms is crucial. The difference between a good execution plan and a bad one can be orders of magnitude in execution time. One alternative is to denormalize, but this is expensive for big database, and can complicate updates. It may also force changes in applications.

3.5 Update Patterns

There are many kinds of updates with which a database system must contend, including:

- Small transactions, such as inserting a new entity, or updating an entity, or deleting one.
- A bulk import of data into existing collection
- A bulk import of a new collection
- A bulk update (such as a 10% pay cut for everyone in many sectors of a large corporation)
- A bulk deletion (for example, removing many rows from a collection)
- Deletion of a collection.

Relational database systems handle all of these well, including the grouping of updates arbitrarily into transactions. These systems typically provide utilities and special optimization for bulk import to a new collection, (i.e. table), and deletion (or truncation) of a collection.

NoSQL systems are particularly good at the fast ingestion of data, because the inserts are spread across the nodes comprising the system. This assumes, however, that the insert is to a single collection and involves no secondary indexes. (The index is likely to be distributed differently from the collection owning the index, so an atomic update becomes very expensive.) Achieving scalable, transactional inserts and updates in a distributed database is currently not possible, and is an area needing research. Another approach to the problem is to research techniques for letting applications live with related collections (e.g. a table and its secondary indexes) that are not updated transactionally. As noted earlier, big data systems, which integrate independent data sources, must tolerate inconsistencies; perhaps non-transactional updates to related collections can be thought of as a special case.

3.6 Data Contention and Isolation

In theory, relational databases provide a simple notion of correctness for concurrent execution: any implementation, which may run transactions concurrently, must provide the same result as some serial execution of the same transactions. In this serial execution, each transaction sees the database in some state, and transforms the database to some other state, (which is then visible to the next transaction). This must be the same set of visible states in any valid concurrent execution.

This mode of operation is called SERIALIZABLE. In practice, SERIALIZABLE restricts concurrency too much, and so relaxed criteria are used. For example, in REPEATABLE

READ mode, a transaction must see the same state for some row each time it is read, even if commits modifying that row occur during the transaction. READ COMMITTED is weaker, permitting different values to be read. This is further from the SERIALIZABLE ideal, but permits greater concurrency, and therefore improved performance.

Big Data systems complicate this picture in two ways. First, there are added difficulties in obtaining correct transactional behavior in a distributed system. Two-phase commit[16] is the best-known approach, but can be expensive in practice. Some techniques such as Paxos[17] seem to have the same problem, and tend not to be used for database transaction management. Some NoSQL systems use an imprecise notion of "eventual consistency", in which all of the updates that need to go together (and that would be combined into a transaction in a relational system), are eventually applied. The problem, of course, is what an application is supposed to do with data that is inconsistent because *eventually* hasn't happened yet.

Second, there are the problems inherent in dealing with external data sources. While the two-phase commit approach was designed to extend to external data sources, in practice this usage of two-phase commit is even more uncommon that usage in a distributed relational database system. Also, as a practical matter, it is not reasonable to expect that different data sources, managed by different organizations, even *can* be updated synchronously. Inconsistencies are to be expected. It is likely that all relevant city, state and federal government offices will eventually find out that someone has moved to a new address, but it will take time, and eventual consistency seems to be all that can be hoped for.

3.7 System and Database Administration

Big Data also raises issues in system and database administration, particularly in the areas of backup and restoration of data, storage management, and replication: In Big Data systems, conventional backup/restore systems may be infeasible: Some very large archives rely on replication to a second site for backup. If many different component systems are used (relational databases, search engines, file systems), each will have its own system administration tasks. Each component will have to be replicated, or otherwise backed up. It is probably not feasible to centralize control over backup and replication, especially since the component systems are probably existing, independently managed systems. However, centralized monitoring systems for backup and replication does seem feasible.

4. DISCUSSION AND NEW DIRECTIONS

Currently, the only way to create a Big Data application involving a variety of datatypes and data sources is to create an application that does the integration. References across

[16] Bernstein and Newcomer (2009)
[17] Lamport (1998)

systems are managed by the application. Any "query" across systems is again coded at the application level, in a non-procedural way. In developing such an application, it becomes clear that relational concepts are broadly applicable. If an application needs to query a collection of real estate records with a repository of geographic entities representing plot boundaries, it is tempting to resort to the language of set-oriented queries, and then think about implementations using query processing techniques familiar from relational databases. One would like an index to help with the most selective part of the query, whether that is a geographical predicate (e.g. lots within ten miles of this point, adjacent to Route 109), or a predicate on real estate records (e.g. lots owned by a given individual within the past ten years). And then one would like to navigate from those selected records to the other data collection to complete the query.

While no relational system operates like this across system boundaries, it is easy to imagine how the relational approach can unify these disparate systems, and imagine writing an application in this way. Much of the integration work, currently done at the application level, would be eliminated.

We believe that the concepts and practices associated with the relational approach to data—data modeling, querying, and application development—provide a good starting point for imagining how Big Data systems might be built in the future. Research will be needed in a number of areas before this imaginary system can be built. Some examples are discussed in the following subsections.

4.1 Data Model and Language

SQL has proven the feasibility and power of non-procedural query languages in the context of the relational model. It represents a very rich paradigm for application development, so there has been little impetus to take a general approach to extension and integration. Customers have gotten used to doing this integration on their own, at the application layer. As we have seen, new requirements call for the steady incorporation of or integration with new data sources and databases, and attempting to solve this problem at the application layer may not be the best long-term approach.

There are some well-known problems with this approach. For example, the relational model is notoriously bad at dealing with sequences. A relation (*i.e.*, a table), is a set of rows, and sets are unordered by definition. Yet sequences are important in many applications. Another example is Google-style searching. SQL provides for precisely defined logical operators used in queries, while searching is a bit looser. If I search for the terms "suitcase" and "bomb", then I'm more interested in documents where these words are close together, but may not want to reject documents that contain only one of these terms; or documents where the terms are far apart; or ones that use synonyms instead. Simply tacking on an interface for these searches is not adequate because applications will sometimes need both capabilities in the same operation, and because we want query optimizers to be able to be effective in planning the execution of that operation.

4.2 Query Processing

SQL has succeeded because non-procedural queries can be reliably compiled into efficient execution plans. Query engines and optimizers of relational database products are, in fact, responsible for SQL's widespread adoption. Because SQL is standardized, developments in query processing technology have been focused on SQL and the relational model. If a future version of SQL, promoting better integration, is to succeed, then query processing technology will again be the determining factor. To date, any ability to query external APIs or data sources has been "tacked on," repurposing existing mechanisms. More research is needed on how query processing would be accomplished if interactions with external sources were a top-level design goal.

4.3 Data Integration

Many ETL systems, data cleaning suites, and *ad hoc* applications have been built for the purpose of combining data sources that share a data model but implement different schemas. As a result, the mechanics of importing data are often not too difficult, but specifying how the import should be done is difficult. Recurring problems include resolving identity, resolving differences in representation (units, encodings, etc.), cleaning up data, and handling data discovered to be in error in the process of integration

4.4 Scalability

Shared-nothing, sharded architectures achieve scalability only for the very simplest queries and updates[18]. Query optimization research in distributed systems is mostly about performance, not scalability. It is currently unclear how to obtain scalability for complex queries. (It is also unclear to what extent scalability is needed for such queries.)

4.5 Transaction Management

Research in this area needs to start with the application area. If the major database updates are the incorporation of new data sources, does that lead to simplifications? How common are concurrent updates to the same data? What sort of isolation is required?

5. CONCLUDING REMARKS

"Big Data," defined as complex, high-volume, rapidly changing, multi-form data from multiple sources, poses many challenges in database technology. Some, but not all, of these challenges can be met using search engines or NoSQL systems. For example, NoSQL systems are relatively open to the integration of external sources. However,

[18] In this case, scalability is defined as the ability to provide uniform performance as the number of queries, and the volume of data, increases.

relational databases will also have an important role to play, although much new research will be needed.

To deal effectively with big data, we must consider many factors, including: number of datatypes, schema changes, query complexity, update patterns, data contention and isolation, and system and database administration. Relational concepts offer potential advantages in many of these dimensions. For example, they are likely to be effective at enabling scalability for complex queries. NoSQL systems provide excellent scalability only for the simplest queries and updates. On the other hand, relational database technology has largely ignored the problem of integrating disparate and heterogeneous data sources—except in the most trivial ways.

There are three broad avenues that can be followed to support the creation of big data applications in the future:

(1) Continue to leave application developers to do the required integration of disparate and heterogeneous data sources themselves, repeatedly facing the same difficult implementation issues;
(2) Radically rethink the entire problem space, developing entirely new, still unimagined approaches from scratch;

or,

(3) Use the data modeling, language, and implementation approaches taken by relational database systems as a starting point for supporting big data.

Following the first avenue essentially amounts to ignoring the problem. It is conceivable, under some circumstances, that we may ultimately be forced to follow the second avenue. However, doing so should not be our first resort, since it involves ignoring decades of useful research in relational systems, in the absence of any similarly broad approach. The third avenue, for now, is best.

GLOSSARY

ACID: Atomic Consistent Isolation Durable. This is essentially a list of transaction properties. Transactions run in parallel as if independent, although they are not stateless.

JDBC: Java Database Connectivity. This is a cross-database standard for Java client applications for databases.

JSON: JavaScript Object Notation. This is a lightweight data interchange format, standardized as ECMA-404, built on two structures: a collection of name/value pairs, or an ordered list of values.

NOSQL: No SQL or "Not Only SQL." (See SQL below). A class of database systems without the structure or many of the constraints of a relational database system.

ODBC: Open Database Connectivity; a cross-database standard for writing client applications against databases, used by a number of large products, including Oracle.

RFC 2617: "Request for Comment" No. 2617 of the Internet Engineering Task Force, "Basic and Digest Access Authentication."

REST: Representational State Transfer. This is an approach for requesting operations on a database through a URL.

SQL: Structured Query Language. This is a language for managing and interacting with data in a relational database system.

URL: Uniform Resource Locator. This is the familiar character string encoding a web address.

Acknowledgements

This work was performed under Institute for Defense Analyses Contract No. DASW01-04-C-0003, task order AK-2-3653. We thank David Jakubek for helpful discussions. We also thank Elizabeth Bowman and Carla Hess for reviewing the paper and making helpful comments.

Disclaimer

The views expressed are those of the authors and do not reflect the official policy or position of the Institute for Defense Analyses, the United States Department of Defense, or the United States Government.

REFERENCES

Arbesman, Samuel (2013). "5 Myths about Big Data." *The Washington Post*, 18 August 2013.

Berman, Jules K. (2013). *Principles of Big Data: Preparing, Sharing, and Analyzing Complex Information*. New York: Elsevier. 261pp.

Bernstein, Philip A., and Eric Newcomer (2009). *Principles of Transaction Processing*, 2nd Edition, Chapter 8. New York: Morgan Kaufmann (Elsevier).

Codd, E.F. (1969). *Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks*. IBM Research Report RJ599 (#12343), August 19, 1969. Yorktown Heights, New York: International Business Machines, Inc.
Retrieved from: http://www.liberidu.com/blog/images/rj599.pdf

Codd, E. F.(1970). "A Relational Model of Data for Large Shared Data Banks." *Communications of the ACM* Vol. 13 No. 6, 377-387.

Davenport, Thomas H., and D.J. Patil (2012). "Data Scientist: The Sexiest Job of the 21st Century." *Harvard Business Review* Vol. 90 No. 10, 70-76.

ECMA (2013). *The JSON Data Interchange Format*. Standard ECMA-404, 1st Ed. Geneva: ECMA International.
http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

Executive Office of the President (2012). *Big Data Across the Federal Government*. Washington, D.C.: Executive Office of the President of the United States.
http://www.whitehouse.gov/sites/default/files/microsites/ostp/big_data_fact_sheet.pdf

Grolinger, Katarina, Wilson A. Higashino, Abhinav Tiwarim and Miriam A.M. Capretz (2013). "Data Management in Cloud Environments: NoSQL and NewSQL Data Stores." *Journal of Cloud Computing: Advances, Systems, and Applications* Vol. 2 No. 22, 1-24.
http://www.journalofcloudcomputing.com/content/pdf/2192-113X-2-22.pdf

Helland, P. (2012). "Life beyond Distributed Transactions: an Apostate's Opinion." Seattle: Amazon.com
http://cs.brown.edu/courses/cs227/archives/2012/papers/weaker/cidr07p15.pdf

Lamport, Leslie (1998). "The part-time parliament." *ACM Transactions on Computer Systems*, Vol. 16 No. 2, 133–169.

Macafee, Andrew, and Erik Brynjolfsson (2012). "Big Data: The Management Revolution." *Harvard Business Review* Vol. 90 No. 10, 61-68.

Stonebraker, Michael (2013). "What does 'Big Data' Mean?" *Communications of the ACM*, Vol. 56 No. 9, 10-11.

Tahara D, and D. Abadi (2014). "SQL Beyond Structure: Text, Documents and Key-Value Pairs". *Proc. New England Database Day 2014.*
https://www.dropbox.com/s/7towlw11q7onrfb/sql_beyond_structure.pdf