EC|Wise

SOFTWARE DESIGN & DEVELOPMENT

Data Processing, Database Management and Analytics in the 21st Century

Tom Spitzer, VP, Engineering

Bill House, Sr. Architect

May, 2014

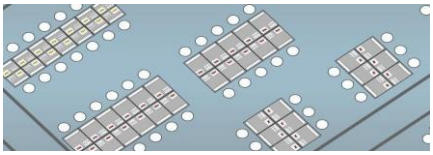
Design

Enterprise Systems

Integration

Data Management

Business Intelligence



Off-shore Operations Delivery

World Class Team

Table of Contents

Concepts

Data storage vs. database

What differentiates data storage from a database? While not particularly useful, it is semantically correct to observe that any repository of information is a data store. One could further posit that, in order to be useful, there has to be a way to retrieve and manipulate stored data. Therefore, a database is a system that provides enhanced or value-added methods of dealing with data in a data store.

For example, consider a collection of books in pile. To find a particular book, you have no choice but to randomly select individual books and inspect each one until you find the book you want. Contrast this to a library, where the books have been organized by Subject, Title and Author, and numbered according to the Dewey Decimal (or equivalent) indexing system. Leveraging that system, you can find an index card or electronic record representing the book in question relatively easily and then immediately locate the book you want based on its assigned number. This is far more efficient, because index cards (or the equivalent electronic index records) are small – you can scan many index cards without taking a step, while inspecting the books directly would require many steps, walking up and down aisles. Also, because the cards and books are organized (i. e., sorted), scanning all the cards is not necessary – you can skip all the irrelevant Subjects and search alphabetically in only the Subject of interest.

- Pile of books on a the floor – just a data store
- Organized books on shelves and index cards to quickly find them – a database

What is Data?

Before we go further into the many database systems that exist, we should first take some time to clarify what we mean by "data". In the software world, data comes in two broad categories – structured and unstructured.

- Unstructured Data – arbitrary blobs of text or binary data (examples: books and complex documents, sounds, photographs, video)
- Structured Data – tuples or rows, tables, attributed objects

Of course things are not that simple, because one often encounters structured data that contain chunks of unstructured data. For now, let us concentrate on Structured Data.

A simple example of structured data is the index card, as in a Rolodex. Each index card has a fixed set of information items that record specific facts about the subject of the card. In that sense, we can say that the index card *represents* the subject. We can also say that the index card *contains* the representation of an instance of the subject.

For example:

- Name: John Smith
- Address: 123 Main Street
- Telephone: (234) 444-3839

If we have two index cards, one for John Smith and one for Jane Smith, we have no problem differentiating these cards because they have different Name values. Even if they have the same Address, we know they are different people. However, as soon as John and Jane produce little John, things become more complicated. How can we tell them apart? Adding an item for Suffix to the Name can distinguish between generations.

- Name: John Smith
- Suffix: Sr.
- Address: 123 Main Street
- Telephone: (234) 444-3839

- Name: John Smith
- Suffix: Jr.
- Address: 123 Main Street
- Telephone: (234) 444-3839

This works within an Anglicized family structure, but what if John and Jane rent a room in their house to another individual who is also named John Smith, Sr.? We need a way to uniquely identity each index card, but family names are clearly not unique enough.

One way to solve this problem is to number the cards.

- Card Number: 1
- Name: John Smith
- Suffix: Sr.
- Address: 123 Main Street

- Card Number: 2
- Name: John Smith
- Suffix: Jr.
- Address: 123 Main Street

- Card Number: 3
- Name: John Smith
- Suffix: Sr.
- Address: 123 Main Street

Unfortunately, while we now have a reliable way to distinguish the cards from each other, the data they contain is not so well discriminated. If we look for cards where the name is John Smith and the Suffix is Sr., we find two cards. Which is which?

One method of improving the card's discrimination capability would be to add descriptive attributes until you arrive at an attribute, or set of attributes, that uniquely describe not the card, but the data the card contains. For example, all email addresses are unique.

- Card Number: 1
- Name: John Smith
- Suffix: Sr.
- Address: 123 Main Street
- Email: john@mymail.com

- Card Number: 2
- Name: John Smith
- Suffix: Jr.
- Address: 123 Main Street
- Email: johnjr@mymail.com

- Card Number: 3
- Name: John Smith
- Suffix: Sr.
- Address: 123 Main Street
- Email: jsmith@mymail.com

Now we can tell the cards apart by Card Number. We can also tell the data they contain apart by Email. What we still can't be sure of is which John Smith, Sr. is John Jr.'s father. We are also having difficulties if not all the persons we need to have index cards for happen to have email addresses. To resolve these issues requires a more advanced theory of data and data management.

The Relational Model and RDBMS

The Relational Database Management System (RDBMS) has become synonymous with "database" in popular usage, but this has not always been the case. Early database systems, such as IMS/DB (http://en.wikipedia.org/wiki/IBM_Information_Management_System), while very fast and efficient, were plagued by issues that arose from their close correspondence between the physical implementation of the data store and programmatic access to the data. Dr. E. F. Codd provided his Relational Model of data and database management (in part) to separate physical data implementation from logical data semantics, as well as data access. Initially criticized by vendors as difficult to implement, the Relational Model and Relational Database Management Systems nevertheless became the industry standard for dealing with business data.

Among the influential aspects of Codd's work are:

1) Relational Data Definition (including NULLs)
2) Data Normalization Rules
3) Logical Data Access (which became SQL)
4) Table Relationships (foreign keys)
5) Server-based data integrity constraints
6) RDBMS Catalog (schema and table metadata)

As RDBMS became more widely adopted for multi-user business applications, Jim Gray defined the properties of a reliable transaction system in the early 1970s, leading to Reuter and Härder coining the ACID terminology in 1983 to describe these properties:

- Atomic: each transaction is all or nothing
- Consistent: each transaction results in a valid database state
- Isolated: concurrent transactions result in the same state as if the same transactions had been executed serially
- Durable: committed transactions remain so, even in the event of power loss, etc.

Trends

RDBMS maturity

RDBMS products became the mainstays of business systems in the 1990s, as they offered the ideal combination of performance, price, ease of use, and scalability to meet the needs of transaction-

oriented business applications. Products from Sybase, Microsoft, IBM, and Oracle evolved in parallel, with slight divergences in product strategy reflecting the market orientation of each of those companies. In its mature form, the RDBMS provides a near-ideal foundation on which to build business applications such as Accounting, Inventory, and other systems, in settings ranging from small businesses to large enterprises. As a result, they became the foundation for commercial ERP products from Peoplesoft (since absorbed by Oracle), Oracle, SAP, and others, as well as for CRM products and other packaged line–of-business applications. In addition, they typically served as the data repositories for custom line-of-business applications as well.

As Internet applications became widely developed and used in the early to mid-2000s, they continued to make heavy use of RDBMS products for data storage. However, with the widespread adoption of Linux as the operating system for web servers, open source software became more acceptable. In addition, there was the realization that the business requirements for Internet-facing applications were often quite different than those for transactional applications. In particular, Internet applications often used databases as content repositories and storage systems for tracking user events, rather than managing transactions, which was their strength. As a result of this de-emphasis of traditional transaction support, developers started to use less mature (but less costly) open source RDBMS systems like MySQL and to look at entirely different ways to store data.

The Emergence of RDBMS Alternatives

Despite all of their virtues, the RDBMS faced serious challenges when confronted with "Internet scale" applications. Scaling for the RDBMS is most easily accomplished by scaling up (buying a bigger server). The economics of the Internet requires that we instead scale out – adding inexpensive commodity servers, rather than buying costly super-servers. This is because web service providers usually started small, with no guarantees of gaining significant numbers of visitors, and then wanted to be able to incrementally grow their capacity, as the number of visitors they served increased. At the other end of the spectrum, we saw Google, Yahoo and Facebook introducing services that had rather novel data management requirements (the ability to manage, search and navigate through millions of links, for instance). This necessitated a different approach to data management.

It was interesting that, as the volumes started to grow, developers created simple data structures that they could write fast code to traverse. In many cases, these were very simple frameworks that sat on top of custom versions of the Linux file system. Since Linux was open source, developers were free to replace key subsystems with versions optimized for their application requirements. At companies like Yahoo and Google, these initiatives evolved into more formal approaches like BigTable and Hadoop. These initiatives were intended to enable distribution of relatively structured data sets over customized file systems on hundreds or thousands of nodes in a network, with specialized tooling (MapReduce algorithms) developed to make use of the distributed data.

This resulted in programming models that were relatively difficult to use, and therefore accessible only to companies that could invest in the expertise necessary to use them effectively. Over time, additional layers of software infrastructure were added on top of Hadoop and BigTable, to make them more accessible to less sophisticated database users. This has evolved to the point where Oracle and Microsoft now offer the ability to integrate queries against Hadoop clusters into their databases.

Other Database Architectures

The advent of the Internet as a mass-market medium has led to the creation of massive amounts of data, far in excess of the data volumes previously required by business applications. Over the past ten years, many companies smaller than Facebook, Yahoo, and Google (who could afford to throw many

programmers at the challenge) have encountered the Big Data problem. We saw small game developers, messaging startups, service providers, and alternative network offerings encounter vast data sets. This created opportunities for tool builders to create next generation database products that addressed different usage scenarios than those best supported by traditional RDBMS products, offering higher-level programming abstractions than those provided by technologies like Hadoop.

Collectively, the industry refers to these non-relational database models as "NoSQL" databases. To understand the value they seek to provide, it is useful to consider the CAP considerations of:

- Consistency (all nodes see the same data at the same time)
- Availability (a guarantee that every request receives a response about whether it was successful or failed)
- Partition Tolerance (the system continues to operate despite arbitrary message loss or failure of part of the system).

As they have emerged over the past ten years, the rationale for many of these products is that many applications do not require complete respect for ACID transactions, while the economics of developing highly-scalable systems that provide all three CAP properties are daunting. Certainly, no large, distributed system can provide total ACID-style Consistency without impacting Availability, Partition Tolerance, or both. To address these challenges, "eventual consistency" models were developed by major web companies, such as Amazon, Google, Facebook, etc. Eventual consistency means taking a pragmatic, data-centric approach to CAP. How consistent do social media status posts really need to be? Must they become visible to all connected users simultaneously, or can they propagate to the users over a period of a few seconds, with some users seeing them later than others? If the latter, then multiple replicas of distributed data may be updated asynchronously, data changes queued up for later commit, with no need to block users from reading previously-committed data that might be a bit stale. By relaxing consistency to allow eventual consistency, availability and partition tolerance are significantly enhanced.

There are several classes of NoSQL databases, but the four main types are key-value stores, document oriented databases, column stores and graph databases. A key-value store, as its name suggests, represents data as a collection of key-value pairs. Document-oriented databases provide the abstraction of a document, with each document in a collection being identified by a key. Graph databases are designed to represent relationships between objects or entities, and their primary data representation is the relationship. Column stores represent tabular data, but instead of physically storing attributes related to an entity together, they store all the attributes of a common type together. There are many documents available on the web going into extensive detail on the different types of databases and the products in each class. Our purpose here is to introduce the concepts and use them later as a framework when we discuss product evaluation strategies.

The Hadoop ecosystem

Let's go back to Hadoop, because the word has become almost synonymous with "big data". When organizations adopt Hadoop today, they are often adopting an entire ecosystem of software technology that has been developed to work together reasonably well. At its core, Hadoop is a set of open source libraries, maintained by the Apache Software Foundation, that enable distributed processing of large datasets. Hadoop manages those datasets in a distributed file system, called HDFS for "Hadoop Distributed File System". Over the years, a number of libraries have been developed to make it easier to work with data in HDFS and to perform analysis and other parallel operations with MapReduce

algorithms. One of the core libraries is "YARN", which is a framework for job scheduling and managing resources in large compute clusters. Per the Hadoop page at Apache, related Apache projects include:

- **Ambari™**: A web-based tool for provisioning, managing, and monitoring Apache Hadoop clusters which includes support for Hadoop HDFS, Hadoop MapReduce, Hive, HCatalog, HBase, ZooKeeper, Oozie, Pig and Sqoop. Ambari also provides a dashboard for viewing cluster health such as heatmaps and ability to view MapReduce, Pig and Hive applications visually alongwith features to diagnose their performance characteristics in a user-friendly manner.
- **Avro™**: A data serialization system.
- **Cassandra™**: A scalable multi-master database with no single points of failure.
- **Chukwa™**: A data collection system for managing large distributed systems.
- **HBase™**: A scalable, distributed database that supports structured data storage for large tables.
- **Hive™**: A data warehouse infrastructure that provides data summarization and ad hoc querying.
- **Mahout™**: A Scalable machine learning and data mining library.
- **Pig™**: A high-level data-flow language and execution framework for parallel computation.
- **Spark™**: A fast and general compute engine for Hadoop data. Spark provides a simple and expressive programming model that supports a wide range of applications, including ETL, machine learning, stream processing, and graph computation.
- **Tez™**: A generalized data-flow programming framework, built on Hadoop YARN, which provides a powerful and flexible engine to execute an arbitrary DAG of tasks to process data for both batch and interactive use-cases. Tez is being adopted by Hive™, Pig™ and other frameworks in the Hadoop ecosystem, and also by other commercial software (e.g. ETL tools), to replace Hadoop™ MapReduce as the underlying execution engine.
- **ZooKeeper™**: A high-performance coordination service for distributed applications.

Sophisticated Hadoop adopters build solutions that integrate several of these tools into a coherent data processing solution tailored to address the class of problems they face.

Closing the Circle: SQL over Hadoop/BigTable

Lately, we have learned about initiatives to put a SQL interface on top of distributed data managed by Hadoop and BigTable. This trend is a nod to the fact mentioned earlier that Hadoop and BigTable are, on the one hand, the technologies with the greatest ability to scale and, on the other hand, they are the most difficult to program. By providing a SQL interface at the user layer, the developers of these new products are trying to make them usable by the legions of people (like some of us!) who have many years of experience and a high degree of expertise using SQL as the basis of their data acquisition and analytic strategies.

Classes of Problems and Solutions

At this point, I want to go back to basics and think about classes of applications and the types of data processing solutions that are appropriate to address them. Early databases were implemented to store and track lists of things, for example property, assets, people, livestock, events. We did not need relational databases to make lists. Almost any type of database can be used to manage lists, but when

lists get huge, which they do in the case where the items in the list are machine-generated events, this is where NoSQL databases shine. In addition to the Hadoop technology family, many NoSQL products like the aforementioned Cassandra, MongoDB and Couchbase are ideal for maintaining lists and enabling applications to retrieve items from the lists in question. They support basic analysis as well.

Similarly, many computer applications were built to manage information taxonomies and classification systems, or to store information content using a document model. As these information systems have grown to Internet scale, they now require distributed systems to enable efficient information storage and retrieval. CouchBase and MongoDB are considered document-oriented databases. Where Couchbase uses standard JSON for its data representation, MongoDB uses a custom variant it calls "BSON". The ability to model relationships in JSON and JSON-like structures makes document-oriented databases also an effective way to represent taxonomy. In addition, graph-oriented databases, like Allegro, Neo4J and Virtuoso, are well-suited to representing both classification and relationships.

Relational databases continue to have a significant role to play in transactional systems as the recorders of information and enforcers of integrity, consistency and availability. The classic example where eventual consistency is not sufficient is banking. In the case where two people share a bank account, the system cannot allow those two people to withdraw all the money from the account at the same time. There are many scenarios like this. When it comes to relational database products, open source products have developed extensive capabilities, in some cases comparable to commercial products. The open source products (especially MySQL and its clones) do tend to be more geared to supporting applications where database-level policy enforcement and complex query performance are not as critical as simple query performance, and where distributed storage is required.

Sybase, IBM, Oracle and Microsoft have each invested tens of millions of dollars over more than twenty-five years in designing and building extremely sophisticated RDBMS products. Each has employed database experts, who have advanced the field of information theory and data science. Each has also leveraged those experts to push their products in different directions. They also perform a significant amount of functional and performance testing, marshaling physical resources beyond the scope of typical open source projects, resulting in products that may be deployed in a variety of enterprise environments, with a relatively small amount of support. To a large extent, open source developers have emulated the commercial products from a database feature standpoint and innovated by introducing novel scaling strategies.

Another class of database application is analytics. Historically, RDBMS vendors and specialized competitors introduced products aimed at supporting reporting and analysis on the data collected by enterprise applications and managed by RDBMS. The primary strategy was to build a data warehouse and to pre-aggregate some of the data, or implement high-performance processors for aggregating along predefined dimensions. This worked reasonably well when all of the data to be analyzed was coming out of operational transactional systems and only amounted to millions or hundreds of millions of records. Now that we have a world where we are collecting ongoing data streams from potentially billions of machines and sensors, collecting unstructured data that also needs to be analyzed, and then integrating all of that information, seeking a holistic understanding of our operational systems, the traditional data warehouse-based paradigm starts to lose its economic leverage. Perhaps counter-intuitively, there are ample case studies available that document the cost and performance advantages of scaled-out systems of commodity hardware over scaled-up super-servers.

21st Century Analytics

As a result of the forces we have discussed, we have seen new classes of analytic toolsets emerge. To a large extent these involve analytic frameworks that can run against data managed in a Big Data stack

like Hadoop. The problem with Hadoop as an analytic platform is that it has not been ideal for real-time analytics, since the typical strategy for doing analysis is to run MapReduce jobs, which can have a significant amount of latency. For this reason, Hadoop is sometimes referred to as a "batch analytics" technology. However, we are starting to see "SQL on Hadoop" implementations that provide the ability to perform SQL queries in real time or near real time. Some of these are frameworks for MapReduce that optimize the process, while others are completely new data-processing techniques. This is an area where there is a substantial amount new development and innovation taking place; it will be some time before it stabilizes.

Among the commercial vendors, a large number are following column-store persistence strategies and doing an increasing amount of analysis in memory. The objective of both these technologies is to reduce the amount of data that has be read from disk, or moved around a network, to satisfy a query. Using these strategies enables more reliable real-time analytics performance. Interestingly, this is more or less the antithesis of the Hadoop model, in that it is dependent on some amount of structure in the data and somewhat purpose-built hardware stacks. Column-store strategies are available to the open source community with products like Monet DB.

Integration and Events

We will very briefly touch on considerations of integration and event processing – these are areas better left to another document. We want to touch on integration from a data perspective because any significant organization collects data via multiple systems and needs to such data into a form where it can be analyzed as a holistic view of the organization's activities and behaviors. We want to mention event processing because, in order to do real-time or near real-time analysis, we believe that the increasing velocity of data flow makes it essential to perform incremental analysis of the event stream as it arrives. This "divide and conquer" approach enhances the availability and timeliness of aggregates and avoids long-running batch processes.

Historically, integration meant moving data from multiple relational data sources, and other relatively well-structured data repositories, into a data warehouse or similar repository. Over the past few years, the problem of data integration has become much more complex, due to the common "big data" factors -- Internet scaling of human-facing applications, the desire to capture people's interactions with web sites at a very detailed level, the increasingly unstructured nature of those interactions, and the number of device and sensor networks attached to our data collection systems, all combine to create an astronomical amount of data. The biggest integration challenge has always been the semantic one of trying to represent the meaning of data consistently across multiple data feeds. This becomes more difficult with more sources of data representing different populations and time frames. Unfortunately, tooling is having a difficult time keeping up with this challenge.

We view event processing as at least a partial solution to this challenge. With event processing systems, event streams feed into a process that analyzes the streams and can filter out noise and determine what existing data the new data relates to. From there, it can determine patterns in the incoming data and intelligently summarize it and add it to existing stores of related data. The fact that event processing occurs in memory eliminates the cost of storing the data and then retrieving it in order to correlate it with other data sets; all of this makes it immediately available for further use. This strategy for data acquisition is relatively new, and we expect an increasing amount of tools support over the next few years.

Conclusion

Data processing has always been an interesting field, but over the past few years, the pace of new technology development has significantly quickened with the imperative to understand "big data" sets. Any organization of significant size must consider a number of dimensions as they consider their go-forward data processing, integration and analytic strategy. That strategy will require taking advantage of the evolution of relational databases for the workloads that require them, and the adoption of "big data" and NoSQL databases as appropriate to address the organization's needs. In conjunction with a data capture and storage strategy, careful consideration of integration techniques will be the keys to enabling effective use of large data sets.

References

"Oracle Information Architecture: An Architect's Guide to Big Data", Oracle White Paper in Enterprise Architecture, March 2012

"Top 5 Considerations When Evaluating NoSQL Databases", MongoDB White Paper, June 2013

"Migrating from Oracle: How Apollo Group Evaluated MongoDB", Brig Lamoreaux, Forward Engineering Project Lead, Apollo Group; September 2012

"Exploiting the Internet of Things with investigative analytics", White Paper by Bloor Research, Philip Howard, May 2013

"ColumnStores vs. RowStores: How Different Are They Really?", Daniel J. Abadi (Yale University), Samuel R. Madden (MIT), Nabil Hachem (AvantGarde Consulting, LLC), SIGMOD '08 paper, June, 2008

"Column-Oriented Database Systems", Stavros Harizopoulos (HP Labs) VLDB 2009 Tutorial

"16 Top Big Data Analytics Platforms", Doug Henschen (Information Week), May 2014

"Introduction to Column Stores", Justin Swanhart, Percona Live, April 2013

Cassandra vs **MongoDB** vs **CouchDB** vs **Redis** vs **Riak** vs**HBase** vs **Couchbase** vs **Neo4j** vs **Hypertable** vs**ElasticSearch** vs **Accumulo** vs **VoltDB** vs **Scalaris** comparison (http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis), Kristof Kovacs

Presto Web site (http://prestodb.io/)

Dr. Codd's Twelve Rules (http://en.wikipedia.org/wiki/Codd's_12_rules)

Database Normalization (http://en.wikipedia.org/wiki/Database_normalization)

"Comparing VoltDB to Postgres", Blog entry (http://pgsnake.blogspot.com/2010/05/comparing-voltdb-to-postgres.html), David Page (Postgres Chief Architect),

"Big Data Using Erlang, C and Lisp to Fight Tsunami of Mobile Data", Jon Vlachogiannis (founder and CTO of BugSense) (http://highscalability.com/blog/2012/11/26/bigdata-using-erlang-c-and-lisp-to-fight-the-tsunami-of-mobi.html)