Advances in Data Management - NoSQL, NewSQL and Big Data

A.Poulovassilis

1 NoSQL

So-called "NoSQL" database systems offer reduced functionalities compared to traditional Relational DBMSs, with the aim of achieving higher performance and scalability for specific types of applications.

The functionality reductions may include:

- not offering full ACID guarantees for transactions,
- not supporting a high-level query language such as SQL, but instead a low-level programmer interface, and/or
- not requiring data to conform to a schema.

The query processing and data storage capabilities of NoSQL systems tend to be oriented towards supporting specific types of applications.

The archetypal examples are settings where there are very large volumes of relatively unstructured data supporting web-scale applications that require quick response times and high availability for users, or that require real-time or near real-time data analytics. This is so-called "Big Data", examples being web log data, social media data, data collected by mobile and ubiquitous devices on the Internet of Things, and large-scale scientific data from experiments and simulations.

A key aim of NoSQL database systems is *elasticity* i.e. undisrupted service in the face of changes to the computing resources of a running system, with adaptive load-balancing.

Two other key aims are *scalability* and *fault-tolerance*:

NoSQL database systems *partition* and *replicate* their data so as to achieve scalability by adding more servers as needed, and also so as to achieve fault-tolerance.

In the presence of replicated data, the concurrency control protocols described in the earlier Notes on Distributed Databases require all, or a majority, of sites to be available in order for updates to proceed. If there is a partitioning of the network, sites that find themselves in a minority partition cannot process updates. Multiple network failures may lead to multiple partitions, none of which contains a majority of sites: in such a situation, the system cannot process Writes and, depending on the protocol, no Reads either, leading to non-availability of the database.

The so-called **CAP Theorem** states that it is not possible to achieve all three of the following at all times in a system that has distributed replicated data, and that one of these needs to be sacrificed:

- Consistency of the distributed replicas at all times;
- Availability of the database at all times;
- Partition-tolerance, i.e. if a network failure splits the set of database sites into two or more disconnected groups, then processing should be able to continue in all groups.

In large-scale distributed systems network partitions cannot be prevented, so either consistency or availability need to be sacrificed in practice.

Protocols such as ROWA and Majority sacrifice availability, not consistency.

However, for some applications (e.g. social networking applications), availability is mandatory while they may only require the so-called "BASE" properties:

Basically available, Soft state, Eventually consistent

Such applications require updates to continue to be executed on whatever replicas are available, even in the event of network partitioning.

"Soft state" refers to the fact that there may not be a single well-defined database state, with different replicas of the same data item having different values.

"Eventual consistency" guarantees that, once the partitioning failures are repaired, eventually all replicas will become consistent with each other. This may not be fully achievable by the database system itself and may need application-level code to resolve some inconsistencies.

NoSQL systems typically do not aim to provide Consistency at all times, aiming instead for Eventual Consistency.

NoSQL systems that relax Consistency do not qualify for performance benchmarking using standard OLTP benchmarks such as TPC-C. New benchmarks are therefore being developed for such systems, such as Yahoo's YCSB benchmark — see "Benchmarking Cloud Serving Systems with YCSB", Brian F. Cooper et al., Proc. SoCC'10, at `http://dl.acm.org/citation.cfm?id=1807152`.

Examples of NoSQL systems include:

- key-value stores, such as Dynamo, Voldermort:
  - store data values that are indexed by a key
  - support insert, delete and look-up operations
  - data is distributed and replicated across nodes according to values of the key
- document stores, such as MongoDB, Couchbase:
  - store more complex, nested-structure, data
  - support both primary and secondary indexes to the data
  - data may be more flexibly partitioned and replicated across nodes
- wide-column stores such as BigTable, HBase, Cassandra:
  - store records that can be extended with additional attributes;
  - records can be partitioned both vertically and horizontally across nodes.
  - Google's Megastore builds on BigTable, providing a schema definition language, and multi-version concurrency control, synchronous replication and ACID transactional semantics for user-defined groupings of records ("entity groups"). Two-phase commit is also supported for achieving atomicity of updates that span different entity groups[1].
- graph DBMSs such as Neo4J, HyperGraphDB, Sparksee, Trinity:
  - although these are classified as NoSQL systems by some commentators, they predate the NoSQL movement and they generally do support full ACID transactions;
  - graph DBMSs focus on managing large volumes of graph-structured data;

[1]Megastore: Providing Scalable, Highly Available Storage for Interactive Services, Jason Blake et al, Proc. CIDR 2011, pp 223-234.

  - graph-structured data differs from other "big" data in its greater focus on the relationships between entities, regarding these relationships as important as the entities;
  - graph DBMS typically include features such as
    - special-purpose graph-oriented query languages
    - graph-specific storage structures, for fast edge and path traversal
    - in-database support for graph algorithms such as subgraph matching, breadth-first/depth-first search, path finding and shortest path.

"Big data" is characterised by the 5 Vs of: volume, velocity, variety, veracity and value.

In addition to managing the data, there is the need to undertake fast processing and analysis of such data.

The **MapReduce** computing framework from Google[2] proposed a way to parallelise computations over distributed, replicated data stored in the Google File System:

- MapReduce processes data that is distributed and replicated across multiple nodes within a shared-nothing architecture, in three steps:
  - (i) the Map step executes a task in parallel on each of the nodes (without the need for any communication between nodes);
  - (ii) the results of the Map step are repartitioned across all the nodes (this is the so-called "shuffle" step);
  - (iii) the Reduce step then executes another task in parallel on each node, on the partition allocated to that node.
- Any number of successive MapReduce cycles can be repeated.
- The Map and Reduce steps are written by the programmer.
- A typical MapReduce task consists of scanning through all the data in parallel, searching for matches to a given pattern (the Map step); grouping all the matches found in the data according to some attribute (shuffle step); and applying an aggregation function to each group (Reduce step).
- Fault tolerance is achieved by automatic reassignment of the tasks of failed nodes to other nodes that store the same data.
- Michael Stonebraker et al.[3] compare MapReduce with parallel databases, concluding that the most common uses of MapReduce are for parallelised data Extraction-Transformation-Load (ETL) tasks and for complex analytics. As such, they consider MapReduce as a complementary technology to conventional DBMSs, identifying the need for interfaces between the two types of systems.

Hadoop is an open-source system consisting of the Hadoop Distributed File System (HDFS) on top of which a MapReduce data processing framework is supported[4].

Pig Latin[5] and Hive[6] both provide higher-level query languages on top of Hadoop that aim to be easlier to program with than MapReduce. Pig and Hive queries are automatically translated into lower-level MapReduce tasks.

[2]Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified data processing on large clusters. Proc. 6th Conf. on Operating System Design and Implementation, Berkeley, 2004

[3]MapReduce and Parallel DBMSs: Frieds or Foes?, M.Stonebraker et al., Comm. ACM 53(1), 2010, pp 64-71

[4]hadoop.apache.org

[5]C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In Proc. SIGMOD, 2008

[6]Hive - A Warehousing Solution Over a MapReduce Framework, A. Thusoo et al., Proc. of the VLDB Endowment, 2(2), 2009, pp 1626-1629

HBase is a wide-column store for Hadoop, modelled on Google's BigTable and implemented on top of HDFS. The Apache Phoenix and Apache Trafodion projects provide SQL-like capabilities over HBase.

Thus, we are seeing a growing convergence between SQL and NoSQL technologies: NoSQL stores are gradually moving towards supporting full ACID transactions, database schemas and declarative querying facilities. Conversely, relational DBMS are extending their capabilities to support also NoSQL functionalities, e.g. Oracle is now a "multi-model" DBMS supporting storage of XML, JSON, graphs and RDF.

In a recent paper describing Google's Spanner DBMS[7] the authors state:

"A prime motivation for this evolution towards a more "database-like" system was driven by the experiences of Google developers trying to build on previous "key-value" storage systems. The prototypical example of such a key-value system is Bigtable [4], which continues to see massive usage at Google for a variety of applications. However, developers of many OLTP applications found it difficult to build these applications without a strong schema system, cross-row transactions, consistent replication and a powerful query language. The initial response to these difficulties was to build transaction processing systems on top of Bigtable; an example is Megastore [2]. While these systems provided some of the benefits of a database system, they lacked many traditional database features that application developers often rely on. A key example is a robust query language, meaning that developers had to write complex code to process and aggregate the data in their applications. As a result, we decided to turn Spanner into a full featured SQL system, with query execution tightly integrated with the other architectural features of Spanner (such as strong consistency and global replication)."

Towards the end of the paper they say:

"But it is perhaps fair to say that from the perspective of many engineers working on the Google infrastructure, the SQL vs. NoSQL dichotomy may no longer be relevant. Our path to making Spanner a SQL system lead us through the milestones of addressing scalability, manageability, ACID transactions, relational model, schema DDL with indexing of nested data, to SQL. For other development organizations that will need to go a similar path we would recommend starting off with the relational model early on, once a scalable, available storage core is in place; well-known relational abstractions speed up the development and reduce the costs of foreseeable future migration."

2 In-Memory Database Systems

These target applications where the data can fit into main memory — typically, partitioned and replicated across multiple servers in a shared-nothing architecture — and where there is a need to achieve very high transaction throughput.

In-memory database systems have increased in popularity recently due to the availability of larger main memories at lower costs.

Example application areas are real-time web analytics, online trade monitoring, telecommunications data management and analysis.

If all the data fits into main memory then I/O and buffer management costs are reduced.

Moreover, data storage and indexing structures can be designed specifically for main memory, rather than the traditional disk-oriented data structures.

Log records still need to be written to persistent storage when a transaction commits, to ensure durability. However, it is possible to wait until blocks containing log records are full before writing them to stable storage, i.e. committing a set of transactions using just one write operation to the persistent log rather than committing each transaction separately. This is known as *group* commit and it reduces the overheads

[7]Bacon, D. F. et al., Spanner: Becoming a SQL system. Proc. 2017 ACM International Conference on Management of Data, pp. 331-343.

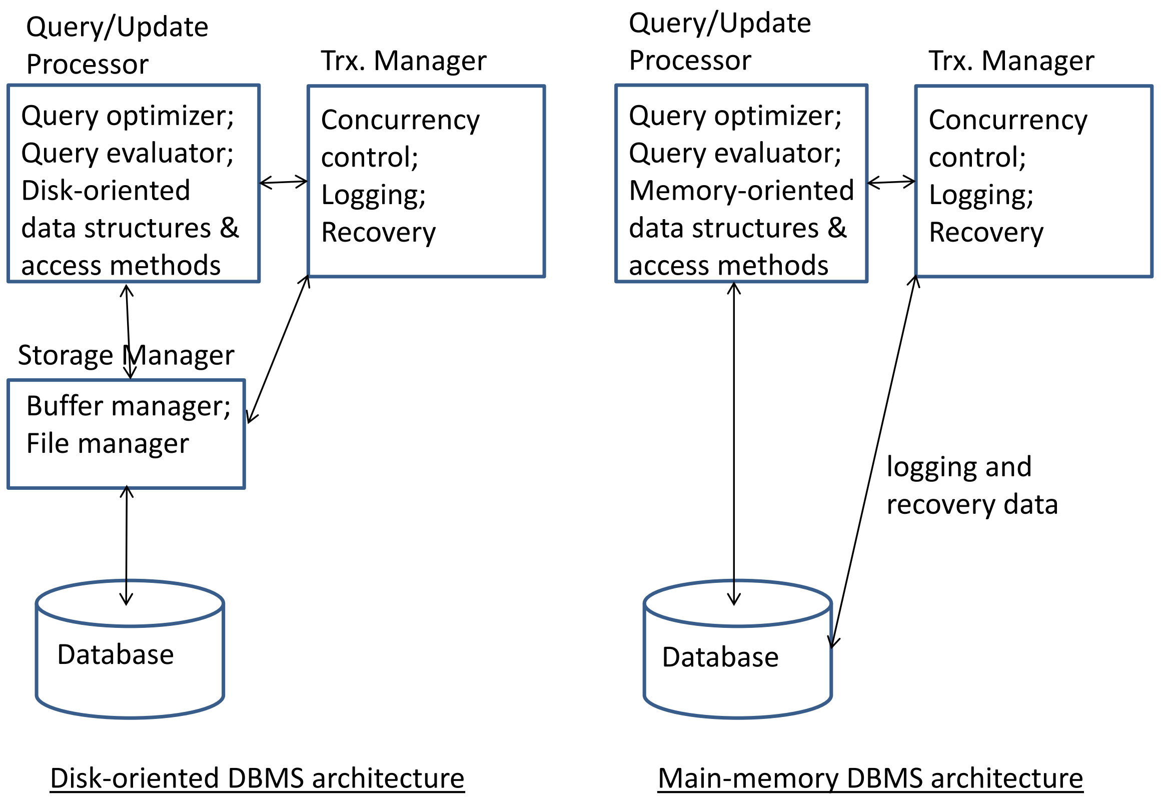
Query/Update Processor
Query optimizer; Query evaluator; Disk-oriented data structures & access methods
Trx. Manager
Concurrency control; Logging; Recovery
Storage Manager
Buffer manager; File manager
Database
Disk-oriented DBMS architecture
Query/Update Processor
Query optimizer; Query evaluator; Memory-oriented data structures & access methods
Trx. Manager
Concurrency control; Logging; Recovery
logging and recovery data
Database
Main-memory DBMS architecture

Figure 1: Disk-Oriented vs. Main-Memory DBMS Architecture

of logging, albeit at the cost of a delay in committing transactions. (Group commit can be used for disk-oriented databases too, not just main-memory ones.)

An example main-memory DBMS is VoltDB, which evolved from the H-Store research project (see Reading 7).

VoltDB is characterised as a “NewSQL” system: these systems aim to achieve high throughput and scalability on OLTP (Online Transaction Processing) workloads while still retaining the relational data model and the SQL query language and still maintaininng full ACID guarantees.

Other examples of NewSQL systems are MemSQL and SAP HANA (which are also main-memory systems) and Google Spanner, NuoDB, Clustrix (not main-memory systems).

Key features of VoltDB:

- Transactions are registered in advance with the DBMS, as Stored Procedures, so that the database can be optimized for running these specific transactions, e.g. with respect to data partitioning and indexing.

  Since the database is memory-resident, there are no disk waits. Since transactions are Stored Procedures, there are no user waits either.

  So the execution of transactions at each node does not need to be interleaved in order to avoid leaving the CPU idle, and can be single-threaded and serial.

- The expectation is that most application transactions will be designed so that they can either be executed at just one site, or executed in parallel as a set of independent sub-transactions running over the partitioned/replicated data.

  (In a distributed database, transactions that only access data at a single node do not incur the overhead of Two-Phase Commit in order to ensure atomicity.)

- Automatic, synchronous replication of data partitions is used in order to achieve fault tolerance and to facilitate recovery from failures.

  K-safety is supported, whereby the database can withstand the loss of up to K nodes.

- Transaction execution at each site is *serial*, in time-stamp order, with no latching or locking.

  Every transaction is assigned a timestamp, and these timestamps form a global total ordering (relying on the NTP local clock synchronisation algorithm[8]).

- Flexible logging is supported whereby DBAs can choose whether to deploy synchronous or asynchronous logging, and how ofen log information should be flushed from memory to disk.

  If a failure occurs at a node and, due to asynchronous logging, some part of the log hasn't been to flushed to disk, then K-safely ensures that another replica of the log will have been successfully flushed a some other node.

- *Command logging* (see Reading 8) is used rather than traditional physical/logical logging, whereby the name of the Stored Procedure (i.e. transaction) that is about to be executed, plus its input parameters, are written to the log. This requires just one log record to be written, compared with the multiple log records required by traditional logging.

  Reading 8 reports a factor of x1.5 in the speed up of TPC-C transaction throughput compared with using physiological logging (physiological logging uses physical identification of a page, and logical identification of changes to the page, e.g. through the use of record identifiers rather than byte offsets).

  After a system failure, the database state can be recovered by re-executing in order the committed transactions recorded in the command log subsequent to the last checkpoint.

  Use of command logging does incur a slower recovery time than use of physiological logging (Reading 8 reports a factor of x1.5). However, given the fact that failures are rare, this increased recovery time is less important than the reduced run-time overhead of logging.

These design decisions are motivated by the performance findings reported in the papers *The End of an Architectural Era (It's Time for a Complete Rewrite)* and *OLTP Through the Looking Glass, and What We Found There* (see Homework Reading).

These papers identify 4 major sources of overheads when OLTP workloads are run on conventional, disk-oriented, database architectures:

- locking
- logging
- latching
- buffer management

[8]David L. Mills, On the Accuracy and Stability of Clocks Synchronized by the Network Time Protocol in the Internet System, ACM SIGCOMM Computer Communication Review, 20(1), 1990, pp 65-75.

2.1 What's new with NewSQL?

Pavlo and Aslett give an analysis of what is "really new" about the systems that are being characterised as "NewSQL" [9]. They come to the following conclusions:

- The idea of storing a database entirely in main memory is not new, with research prototypes appearing in the 1980s and commercial systems in the 1990s. They state however that "One thing that is new with main memory NewSQL systems is the ability to evict a subset of the database out to persistent storage to reduce its memory footprint. This allows the DBMS to support databases that are larger than the amount of memory available without having to switch back to a disk-oriented architecture." This is known as "anti-caching".
- Partitioning data into disjoint subsets and distributed transaction processing on partitioned data was proposed in early seminal work on distributed database prototypes in the 1970s and 80s. However, a new aspect in NewSQL is the ability to dynamically move data between different nodes for capacity-upgrading and load-balancing purposes (similarly to NoSQL systems, but maintaining ACID transactional guarantees as well).
- NewSQL systems generally use multi-version concurrency control (MVCC) for improved throughput, in combination with locking and/or timestamps — such schemes have been known and used in DBMSs since the 1980s. VoltDB is an exception that does not use MVCC but instead schedules transactions to be executed serially at each node.
- Techniques for maintaining consistency between distributed replicas and for distributed crash recovery have been available in commercial DBMSs since the 1990s.

Their overall conclusion is that NewSQL systems are not a radical departure from traditional DBMS architectures but an evolutionary step forwards, combining previous approaches and algorithms and extending them with many engineering innovations in an era when computing resources are ubiquitous and much more affordable but also where the demands of applications are much higher.

They identify an ongoing progression towards the next stage of combined OLTP and OLAP capabilities within one DBMS, incorporating advancements from the NewSQL systems on OLTP with ongoing advancements in OLAP systems (e.g. columnar storage, stream data processing). These are so-called hybrid transaction-analytical processing (HTAP) engines. They predict that this development is likely to lead in the long-term to the obsolescence of data warehouses supporting only OLAP operations.

Homework Reading

All of the following are on Moodle:

1) Read Section 1 and Section 2 of:

OLTP Through the Looking Glass, and What We Found There, Stavros Harizopoulos et al., Proc. SIGMOD'08, 2008, pp 981-992.

Accessible at `dl.acm.org/citation.cfm?id=1376713`

Read the rest of the paper, for interest.

2) Read:

Scalable SQL and NoSQL Data Stores, Rick Cattell, SIGMOD Record, 39(4), 2010, pp 12-27.

Accessible at `dl.acm.org/citation.cfm?id=1978919`

[9] Pavlo, A. and Aslett, M., 2016. What's really new with NewSQL?. ACM SIGMOD Record, 45(2), pp.45-55.

You can skip Subsections 2.3-2.6; 3.1-3.2; 3.4; 4.2-4.5; 5.1; 5.3-5.6; 7 to the end.

Although it is a few years old, this paper covers the main principles of NoSQL and NewSQL databases, and also describes some Use Cases. N.B. Some of the products described are now obsolete, and all of the rest will have evolved since the paper was written.

3) Read:

Data management in cloud environments: NoSQL and NewSQL data stores, Katarina Grolinger et al., Journal of Cloud Computing, 2(22), 2013, pp 12-27.

Accessible at `https://journalofcloudcomputing.springeropen.com/articles/10.1186/2192-113X-2-22`

This is a more up-to-date account of NoSQL and NewSQL data stores, motivated from the data management requirements of Cloud Computing.

You can skim/skip these subsections of the paper: “Related Surveys” on page 4; “Methodology” on page 5; “Partitioning” on page 10 through to the end of “Security” at the top of page 19.

4) Read the following sections of W. Lemahieu et al. 2018:

Pages 300-346 on NoSQL databases (you can skim/skip sections 11.2.3-11.2.8, pages 322-329, section 11.5.2). Pages 538-545 on Eventual Consistency and BASE Transactions (you can skim/skip pages 541-544). Pages 626-631 on the Characteristics of Big Data and the history of Hadoop and MapReduce.

5) Read the Volt DB Technical Overview White Paper, 2014.

6) Read Nigel Martin’s notes on NoSQL Databases, from the DKM module, which have a specific focus on MongoDB and Neo4J.

Further reading, for interest

7) *The End of an Architectural Era (It’s Time for a Complete Rewrite).* Michael Stonebraker et al., Proc. VLDB’07, 2007, pp 1150-1160. Accessible at
`http://dl.acm.org/citation.cfm?id=1325981`

8) For more information about recovery in VoltDB, see N. Malviya, A. Weisberg, S. Madden, M., *Rethinking Main Memory OLTP Recovery.* Proc. ICDE’ 2014, pp 604-615. Accessible at
`http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6816685`