

Advances in Data Management - NoSQL, NewSQL and Big Data

A.Poulovassilis

1 NoSQL

So-called “NoSQL” systems offer reduced functionalities compared to traditional Relational DBMS, 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 requiring quick response times and high availability for users, or requiring real-time or near real-time data analysis: so-called “Big Data” e.g. web log data, social media data, data collected by mobile and ubiquitous devices, large-scale scientific data from experiments and simulations.

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

NoSQL systems partition and replicate their data, aiming to achieve *scalability* by adding more servers as needed, and also aiming for *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. Rather than ACID, such applications only require the so-called “BASE” properties:

Basically available, Soft state, Eventually consistent

Such applications require updates to continue to be executed on the available replicas, 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.

Examples of NoSQL systems include:

- implementations of the **MapReduce** parallel computing framework¹:
 - MapReduce processes data that is distributed and replicated across multiple nodes in 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.
 - Hadoop is an open-source system consisting of the Hadoop Distributed File System (HDFS) on top of which a MapReduce data processing framework is supported².
 - Pig Latin³ and Hive⁴ are examples of higher-level query languages built on top of Hadoop, that translate into lower-level MapReduce tasks.
 - Michael Stonebraker et al.⁵ 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 DBMSs, identifying the need for interfaces between the two types of systems.
- key-value stores, such as Dynamo, Voldermort:

¹Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified data processing on large clusters. Proc. 6th Conf. on Operating System Design and Implementation, Berkeley, 2004

²hadoop.apache.org

³C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In Proc. SIGMOD, 2008

⁴Hive - A Warehousing Solution Over a MapReduce Framework, A. Thusoo et al., Proc. of the VLDB Endowment, 2(2), 2009, pp 1626-1629

⁵MapReduce and Parallel DBMSs: Friends or Foes?, M.Stonebraker et al., Comm. ACM 53(1), 2010, pp 64-71

- store data values that are indexed by a key
- support insert, delete and look-up operations
- data is distributed across nodes according to values of the key
- document stores, such as MongoDB, SimpleDB, CouchDB:
 - store more complex, nested-structure, data
 - support both primary and secondary indexes to the data
 - data may be more flexibly partitioned and/or replicated across nodes
- wide-column stores such as BigTable, HBase, Cassandra, PNUTS:
 - store records that can be extended with additional attributes
 - records can be partitioned both vertically and horizontally across nodes

More recently, 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 groups⁶.

- graph DBMSs such as Neo4J, HyperGraphDB, Sparksee, Horton, 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;
 - 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.

2 In-Memory Database Systems; NewSQL

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.

Example applications are real-time web analytics, online trade monitoring, telecommunications data management and analysis, social networking applications.

In-memory database systems are made possible by the increasing availability of larger main memories at lower costs.

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

⁶Megastore: Providing Scalable, Highly Available Storage for Interactive Services, Jason Blake et al, Proc. CIDR 2011, pp 223-234.

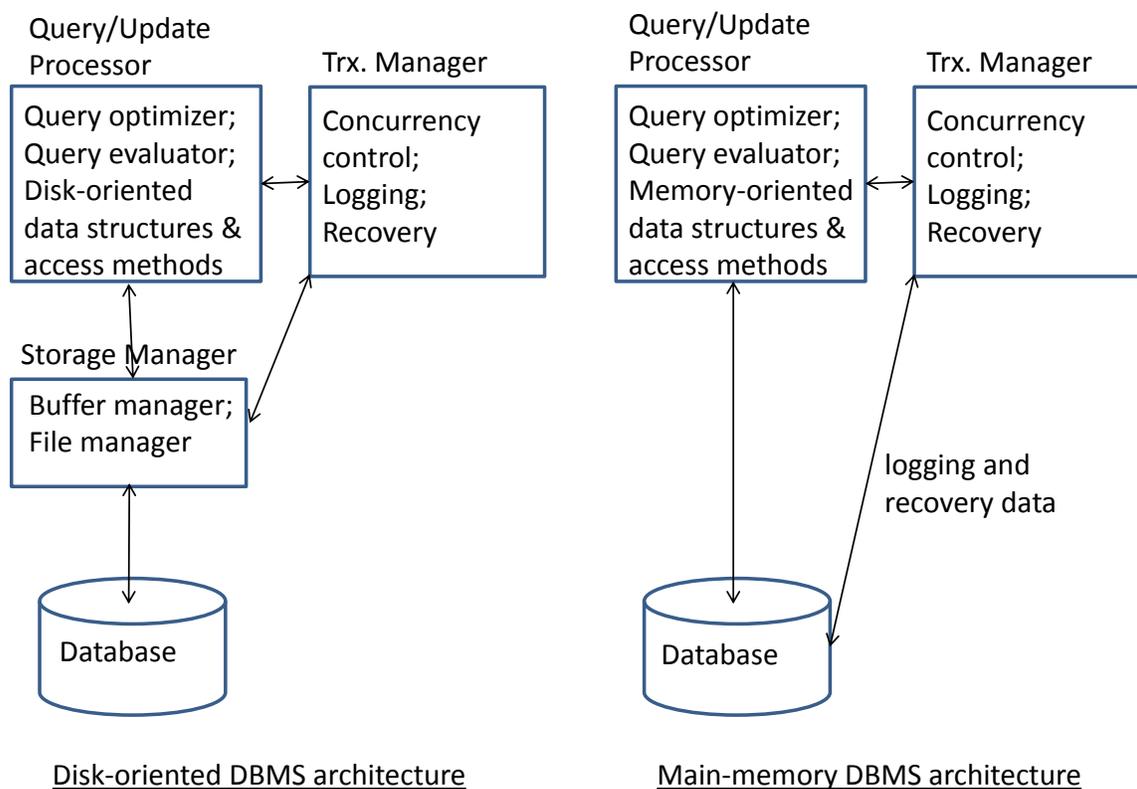


Figure 1: Disk-Oriented vs. Main-Memory DBMS Architecture

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 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 project⁷.

VoltDB has been 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 maintaining full ACID guarantees.

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.

⁷HStore: A High Performance, Distributed Main Memory Transaction Processing System, R.Kallman et al., PVLDB’08, 2008, pp 1496-1499

- 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 algorithm⁸).

- Flexible logging is supported whereby DBAs can choose whether to deploy synchronous or asynchronous logging, and how often 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 6) is used rather than traditional physical or 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 6 reports a factor of x1.5 in the speed up of TPC-C transaction throughput compared with using physiological logging.

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 6 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

⁸David 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.

Homework Reading

All of the following are on Moodle:

1) Sections 1 and 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) 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

You can skip subsections 2.3 - 2.6; 3.1; 3.4; 4.2; 5.3 - 5.6.

3) Chapter 19, Section 19.9 from Silberschatz et al. on “Cloud-Based Databases”.

4) Volt DB Technical Overview White Paper, 2014.

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

Further reading, for interest

6) 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>

7) For 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