Mehmood *et al. J Big Data (2017) 4:8*
DOI 10.1186/s40537-017-0068-5

Journal of Big Data

RESEARCH Open Access

Modeling temporal aspects of sensor data for MongoDB NoSQL database

Nadeem Qaisar Mehmood*, Rosario Culmone and Leonardo Mostarda

*Correspondence:
nadeemqaisar.mehmood@unicam.it
Department of Computer Science, UNICAM, Via del Bastione, 62032 Camerino, Italy

Abstract

Proliferation of structural, semi-structural and no-structural data, has challenged the scalability, flexibility and processability of the traditional relational database management systems (RDBMS). The next generation systems demand horizontal scaling by distributing data over autonomously addable nodes to a running system. For schema flexibility, they also want to process and store different data formats along the sequence factor in the data. NoSQL approaches are solutions to these, hence big data solutions are vital nowadays. But in monitoring scenarios sensors transmit the data continuously over certain intervals of time and temporal factor is the main property of the data. Therefore the key research aspect is to investigate schema flexibility and temporal data integration aspects together. We need to know that: what data modelling should we adopt for a data driven real-time scenario; that we could store the data effectively and evolve the schema accordingly during data integration in NoSQL environments without losing big data advantages. In this paper we explain a middleware based schema model to support the temporal oriented storage of real-time data of ANT+ sensors as hierarchical documents. We explain how to adopt a schema for the data integration by using an algorithm based approach for flexible evolution of the model for a document oriented database, i.e, MongoDB. The proposed model is logical, compact for storage and evolves seamlessly upon new data integration.

Keywords: NoSQL, MongoDB, Big data, Schema modeling, Time-series, Real-time, ANT+ protocol

Introduction

The emergence of Web 2.0 systems, the Internet of Things (IoT) and millions of users have played a vital role to build a global society, which generates volumes of data. At the same time, this data tsunami has threatened to overwhelm and disrupt the data centers [1]. Due to this constant data growth the information storage, support and maintenance have become a challenge while using the traditional data management approaches, such as structural relational databases. To support the data storage demands of new generation applications, the distributed storage mechanisms are becoming the de-facto storage method [2]. Scaling can be achieved in two ways, vertical or horizontal, where the former means adding up resources to a single node, whereas in the latter case we add more nodes to the system [3]. For the problems that have arisen due to data proliferation, the RDBMS fail to scale the applications horizontally according to the incoming data traffic

[2]; because they require data replication on multiple nodes, so they are not flexible to allow data and read/write operations distributed over many servers. So we need to find systems that would be able to manage big volumes of data.

This flood of data passes challenges not only due to its sheer size but also due to the data types, hence demands more robust mechanisms to tackle different data formats. The Web, e-science solutions, sensor laboratories and industrial sector produce in abundance both structural, semi-structural and non-structural data [4–6]. This is not a new problem, and can be traced back to the history of object-relational databases, under the name of Object-Relational Impedance Mismatch [7]. This mismatch is natural, when we try to model an object into a fixed relational structure. Similarly, the digital information with different structures, such as natural text, PDF, HTML and embedded systems data, is not simple enough to capture as entities and relationships [8]. Even if we manage to do this, it will not be easy to change afterwards, hence such mechanisms are rigid for schema alteration because they demand pre-schema definition techniques. Several new generation systems do not like to fix their data structure to a single schema; rather they want their schema to evolve in parallel to an entity data type's adaptation, hence they want flexibility [9, 10].

Besides the data abundance and different formats, the rapid flow of data has also attracted the researchers to find mechanisms to manage the data in motion. Typically this is to consider that, how quickly the data is produced and stored, and its associated rates of retrieval and processing. This idea of data in motion is evoking far more interest than the conventional definitions, and needs a new way of thinking to solve the problem [11]. This is not associated only with the growth rate at the data acquisition end, but also data-flow rate during transmission; as well the speed at which data is processed and stored in the data repositories. Any way, we are aware of the fact that today's enterprises have to deal with petabytes instead of terabytes; and the increase in smart object technology alongside the streaming information has led the constant flow of data at a pace that has threatened the traditional data management systems [11]. RDBMS use two-dimensional tables to represent data and use multi-join transactional queries for the database consistency. Although they are mature and still useful for many applications, but processing of volumes of data using multi-joins is prone to performance issues [12, 13]. This problem is evident when extensive data processing is required to find hidden useful information in huge data volumes; but such data mining techniques are not in our current focus [14–17], as we limit our discussion to NoSQL temporal modeling and schema based data integration.

Above discussed problems of prolific, multi-structured heterogeneous data in flow urge the researchers to conduct research to find alternate data management mechanisms, hence NoSQL data management systems have appeared and are now becoming a standard to cope with big data problems [11, 18]. Such new data management systems are being used by many companies, such as Google, Amazon etc. The four primary categories of their data model are: (i) key-value stores, (ii) column-oriented, (iii) document, and (iv) graph databases [19, 20]. For rationality, sanity and demonstrating the storage structure, the researchers follow the database schema techniques without losing the advantages of schema flexibility provided by NoSQL databases. Such schema modeling strategies in NoSQL databases are quite different from the relational databases.

Collections, normalization and document embedding are few variants to consider during building schema models because they affect the performance and storage effectively because such databases grow very quickly.

While dealing with real-time data, in continuous or sliced snapshot data streams, the data items possess observations which are ordered over time [21]. During previous years, research efforts had been conducted to capture temporal aspects in the form of data models and query languages [22, 23]. But mostly those efforts were for relational or object-oriented models [23], and can be considered as a conceptual background to solve advanced data management challenges [24]. The emerging applications, such as sensor data [25], Internet traffic [26], financial tickers [27, 28] and e-commerce [29], produce large volumes of timestamped data continuously in real-time [30, 31]. The current methods of centralized or distributed storage with static data impose constraints in addressing the real-time requirements [32], as they inflict pre-defined time convictions unless timestamped attributes are explicitly added [31]. They have limited features to support the latest data stream challenges and demand research to augment the existing technologies [31, 33].

In remote healthcare long term monitoring operations, based on Body Area Networks (BAN), demand low energy consumption due to limited memory, processing and battery resources [34]. These systems also demand communication and data interoperability among sensor devices [35]. Recently a propriety protocol ANT+ provides these low energy consumption features; and strengthens the goals of IoT through the interoperability of devices based on Machine-to-Machine (M2M) mechanisms, which employs use case specific device profile standards [34, 36]. Device interoperability, low energy and miniaturisation features allow the building of large ecosystems, hence enable millions of vendor devices to get integrated and interoperated. IoT ecosystems want general storage mechanisms having structural flexibility to accept different data formats arriving from millions of sensory objects [37]. The non-relational or NoSQL databases are schema-free [2]; and allow storage of different data formats without prior structural declarations [34, 37]. However for the storage we need to investigate the NoSQL models to design and develop [8, 22]; besides flexibly preserving the big data timestamped characteristics for the massive real-time data flow during acquisition processes [24]. Although all NoSQL databases have unique advantages, but document-oriented storage, as MongoDB provides, is considered robust for handling multiple structural information to support IoT goals [38]. This rejects the relational structural storage and favours Java Script Object Notations (JSON) documents to support dynamic schemas; hence provide integration to different data types besides scalability features [39, 40].

This article presents a general approach to model temporal aspects of ANT+ sensor data. The authors develop a prototype for the MongoDB NoSQL real-time platform and discuss the temporal data modeling challenges and decisions. An algorithm is presented which integrates JSON data as hierarchical documents and evolves the proposed schema without loosing flexibility and scalability.

This article is organized as follows. "Data stream and data stream management systems (DSMS)" is about time series data. Different NoSQL databases are discussed in detail in "Limitations of RDBMS". It is followed by a subsection discussing MongoDB as a well-known document oriented database. "Big data management frameworks", discusses the

different techniques to model time series data using MongoDB. This follows a middleware description explaining how to store data in the MongoDB. "Modeling aspects" and "Temporal modeling for an ANT+ sensor use case" give future work and a short summary respectively.

Time series in medical data

A time series is a sequence of numerical measurements from observations collected at regular durations of time. Such successive times can either be continuous or discrete time periods. These sequence of values at particular intervals are common in situations, such as weekly interest rates, stock prices, monthly price indices, yearly sales figures, weather reports, patient's medical reports and so forth.

Healthcare monitoring systems measure physiological and biological body parameters, using BAN, of a patient's body in real-time. Because timely information is an important factor to detect immediate situations and to improve decision making processes, based on a patient's medical history, so considering temporal aspects are vital. Such sequence of values represent the history of an operational context and is helpful in a number of use cases where history or order is required during the analysis. This sequences of data flows in streams of different speeds and also needs proper management.

Data stream and data stream management systems (DSMS)

Data streams, as continuous and ordered flow of incoming data records, are common in wired or wireless sensor network based monitoring applications [31]. Such widely used data intensive applications don't directly target their data models for persistence storage, because the continuously arriving multiple, rapid, time-varying, and unbounded streams lose the support for storage as an entirety [31], and a portion of arrived stream is required to keep in the memory for initial processing. This is not feasible using the traditional DBMS to load the entire data and operate upon it [41]. Golab et al. [31] highlights the following requirements for the DSMS.

- Data models and queries must support order and time based operations.
- Summarized information is stored, owing to the inability of entire stream storage.
- Performance and storage constraints do not allow backtracking over a stream.
- Real-time monitoring applications must react to outlier data values.
- Shared execution of many continuous queries is needed to ensure scalability.

DBMS comparison with DSMS

There are three main differences while comparing DSMS with the DBMS. First they do not directly store the data persistently rather keep the data in the main memory for some time for autonomous predictions to respond to outlier values, such as fire alarm, emergency situations as in healthcare domain etc [42]. Therefore DSMS computation is generally *data driven*, i.e. to compute the results as the data is available. In such cases the computation logic always resides in the main memory in the form of rules or queries. On the other hand DBMS approach is *query driven*, i.e. to compute the results using queries over permanently stored data. Because of *data driven* nature, the very first issue which DSMS must solve is to manage the changes in data arrival rate during a specific

query lifetime. Second, it is not possible to keep all the previous streams in the memory due to their unbounded and massive nature. Therefore only a summary or synopsis is kept in the memory to answer the queries whereas the rest of the data is discarded [21]. Third, since we cannot control the order of the data arrival, critical to consider the order of the arrived data values, hence their temporal attribute is essential. In order to handle the unboundedness of the data, the fundamental mechanism used is that of *window* - which is used to define slices upon the continuous data to allow correct data flow in finite time [42].

Data-driven computation, *unbounded streams* and *timestamped data* are the main issues that have arisen while dealing with streaming data, such as during sensor data acquisition in monitoring scenarios. This poses novel research challenges and exciting directions to follow with focus on temporal model, techniques and algorithms. These issues need proper management for any of the relational, object-relational or big data management research paradigms; and aim at data modeling and successfully exploiting the time-dependent characteristics for these paradigms ranging from the temporal based models to query models. Although the directions, developed in previous years for the relational or object-relational domains, provide the basic fundamental footsteps to follow; but require further insights to tackle the advanced Big Data challenges [31, 41]. In particular the emerging real-time data-driven applications, having volumes of various data velocities, demand such research inputs to bring number to advantages to the Information and Communication Technology (ICT) world, specially in promoting IoT and Web 2.0 and 3.0. Hence it is becoming mandatory to tackle the challenges associated with temporal data streams for which the relational database management systems have given in.

Limitations of RDBMS

This section explains what traditional relational approaches lack, why they are not best fit for managing time-variant, dynamically large and flowing data. This absence has opened the door for a disruptive technology to enter into the market and to gain widespread adoption in the form of NoSQL databases, as it offers better, efficient, cheaper, flexible and scalable solutions [43]. Features lacking in RDBMS are:

- *Flexibility* RDBMS restrict the applications to a predefined schema; so any change in the application requirements will lead to the redefinition of the database schema [8, 9]. Using RDBMS the developers have to rely on the data architects or modelers, as they miss developer centric approach from application inception to the completion [10]. Even in the case of the schema evolution especially in dynamic applications scenarios [9], as this is observed in information changing scenarios during dynamic events generation in new generation systems [11]. NoSQL systems have a strong coupling between data models and application queries, so any change in the query will require changes to the model, which is flexible [10]. In contrast to this RDBMS systems have logical and physical data independencies, where database transaction queries come into play only when the data model is defined physically [10].
- *Scalability* Over more than half a century RDBMS have been used by different organizations as they were satisfying the need of business dealing with static, query

intensive data sets since these were comparatively small in nature [44]. For the large data sets the organizations had to purchase new systems as add-on to the system, as the single server host the entire database. For scaling we need to buy a large more expensive server [43]. For the big data applications RDBMS systems are forced to use distributed storage, which includes table partitioning, data redundancy and replication; because of disk size limits and to avoid hard disk failures [10]. Such data distribution involve multiple table joins, transaction operations running upon multiple distributed nodes, disk lock management and ACID (atomicity, consistency, isolation, and durability) compliance hence affect the performance adversely. The notion of vertical scalability allows the addition of more CPUs, memory and other resources [8]; but quickly reaches its saturation point [10].

- *Data structures* RDBMS were in the era when the data was fairly structured, but now due to the advent of new generation trends, such as IoT, Web 2.0 etc. the data is now no more statically structural [43], which involves unstructured data (e.g., texts, social media posts, video, email). Therefore RDBMS database transactions and queries come to play their role in already designed data models; in contrast to the NoSQL databases, which support application specific queries and allow dynamic data modeling [10].
- *Source codes and versioning* Relational databases are typically closed source with licensing fees. Off-the-shelf RDBMS do not provide support for data versioning, in contrast to the NoSQL databases which support natively [10]. A list of open and closed source NoSQL databases is present in [45].
- *Sparsity* RDBMS databases when deal with large data sets, upon missing values there is possibility of a lot of sparsity in the data sets.

Data proliferation, schema flexibility and efficient processing are the problems appearing during the development of latest data-driven applications, as we learned in "Introduction". We learned that RDBMS are not sufficient to deal these issues, and don't meet the latest requirements of the next generation real-time applications [24, 31, 33]. *Volume, Variety* and *Velocity* are the three corresponding big data characteristics [10, 18, 46], which are discussed in "Big data management frameworks", which is about a precise discussion regarding big data frameworks.

Big data management frameworks

A big data management framework means the organization of the information according to the principles and practices that would yield high schema flexibility, scalability and processing of the huge volumes of data, but for which traditional RDBMSs are not well suited and becomes impractical. Therefore, there is a need to devise new data models and technologies that can handle such big data. Recent research efforts have shown that big data management frameworks can be classified into three layers that consist of file systems, database technology, and programming models [19]. However in this article we shall focus upon database technologies only in context of the healthcare domain with real-time and temporal perspective.

NoSQL also be interpreted as the abbreviation of "NOT ONLY SQL" or "no SQL at all" [45], whereas this was first used by Carlo Strozzi in 1998 for his RDBMS that did not offer

an SQL interface [47]. NoSQL databases are often used for storing of the big data in non-relational and distributed manner, and its concurrency model is weaker than the ACID transactions in relational SQL-like database systems [14]. This is because NoSQL systems are ACID non-compliant by design and the complexity involves in enforcing ACID properties does not exist for most of them [10]. For example some of the ACID compliant NOSQL databases are: Redis [48], Aerospike [49] and Voldemort [50] as key-value stores [51]; where as Neo4jDB [52] and Sparksee are as graph-based data stores [8, 51]. In contrast to this MongoDB is not ACID compliant document-oriented database [8].

The V's of big data

The V's of big data is paramountly, even in healthcare, refer to as mainly for *Volume, Variety, Velocity* and *Veracity* [14]. The first three V have been introduced in [53], and the V for *Veracity* has been introduced by Dwaine Snow in his blog *Thoughts on Databases and Data Management* [54].

Volume Prolific data at scale creates issues ranging from storage to its processing.

Velocity Real-time data, data streams—analysis of streaming data, data snapshots in the memory for quick responses, availability for access and delivery.

Variety Many formats of data—structured, unstructured, semi-structured, media.

Veracity Deals with uncertain or imprecise data, its cleaning before the processing. Variety and Velocity goes against it as both do not let to clean the data.

NoSQL database categories

Based on the differences in the respective data models, NoSQL databases can be organized into following basic categories as: key-value stores, document databases, column-oriented databases and graph databases [10, 14, 19, 20].

Key-value stores

These are systems that store values against the index keys, as key-value pairs. The keys are unique to identify and request the data values from the collections. Such databases has emerged recently and are influenced heavily by Amazon's Dynamo key-value store database, where data is distributed and replicated across multiple servers [62]. The values in such databases are schema flexible and can be simple text strings or more complex structures like arrays. The simplicity of its data model makes the retrieval of information very fast, therefore supports the big data real-time processing along the scalability, reliability and highly available characteristics. Some of the key-value databases store data ordered on the keys, such as Memcached [55] or Berkeley DB [66]; while others do not, such as Voldemort [50] etc. Whereas some keep entire data in memory, such as Aerospike [49], Redis [48]; others use it after writing it to the disk permanently (like Aerospike, MemcacheDB [56] etc.) with the trade-off replying to the queries in real-time. The scalability, durability and flexibility depends upon different mechanisms like partitioning, replication, object versioning, schema evolution [19]. Sharding, also known as partitioning, is the splitting of the data based upon the keys; whereas the replication, also known as mirroring, is the copying of the data to the different nodes.

Amazon's Dynamo and Voldemort [50], which are used by Linkedin, apply this data model successfully. Other databases that use this model of data category are such as:

Redis [48], Tokyo Cabinet [67] and Tokyo Tyrant [68], Memcached [55] and MemcacheDB [56], Basho Riak [60], Berkeley DB [66] and Scalaris [69]. Whereas Cassandra is a hybrid of key-value and column-oriented database models [57]. Table 1 summarizes the characteristics of some of the Key-value stores.

Column-oriented databases

Relational databases have their focus on rows in which they store the instances or the records and return rows or instances of an entity against a data retrieval query. Such rows posses unique keys against each instance for locating the information. Whereas column-oriented databases store their data as columns instead of the rows and use index based unique keys over the columns for the data retrieval. This supports attribute level access rather than the tuple-level access pattern.

Only query relevant necessary columns are required to be loaded, so this reduce the I/O cost significantly [70]. These are good for read-intensive applications, as they only allow relevant data reads because each column contains contiguous similar values; so calculating aggregate values will also be very fast. More columns are easily addable and a column may be further restructured called super-column, where it contains nested (sub) columns (e.g., in Cassandra) [14]. Super columns are key-value pairs, where the values are columns. Columns and super-columns are both tuples with a name and value. The key difference is that a standard column's value is a string, whereas a super-column's value is a map of columns. Super-columns are sorted associative array of columns [71].

Google's Bigtable, which played the inspirational role for the column databases [74], is a compressed, high performance, scalable, sparse, distributed multi-dimensional database built over a number of technologies, such as *Google File System* (*GFS*) [75], a cluster management system, SSTable file format and Chubby [76]. This provides indexes over rows, columns, as well as a third timestamp dimension. Bigtable is designed to scale across thousands of system nodes and allows to add more nodes easily through automatic configuration.

This was the first most popular column oriented database of its type however latter many companies introduced some other variants of it. For example Facebook's Cassandra [77] integrates the distributed system technologies of Dynamo and the data model from Bigtable. It distributes multi-dimensional structures across different nodes based upon four dimensions: rows, column families, columns, and super columns. Cassandra was open sourced in 2008, and then HBase [72] and Hypertable [78], based upon a proprietary Bigtable technology, have emerged to implement similar open source data models. Table 2 provides the description about some column-oriented databases in a categorical format.

Graph databases

Graph databases, as a category of NoSQL technologies, represent data as a network of nodes connected with edges and are having properties of key-value pairs. Working on relationships, detecting patterns and finding paths are the best applications to be solved by representing them as graphs. Neo4j [52], Allegro Graph [79], ArangoDB [80] and OrientDB [81] are few examples of such systems, and are described along their characteristics in a categorical format in Table 3. Neo4j is the most popular open source, embedded, fully transactional with ACID characteristics graph-based database. This is schema

Table 1 Key-value stores

Name	Data model	Scalability	Description	Who uses it
Memcached *from* http://www.brad-fitz.com [55]	Set of key-value in associative array	Auto sharding, no replication and persistency	In-memory cache systems, no disk persistence (MemcacheDB give persistent storage [56]); file system storage, ACID	LiveJournal, Wikipedia, Flickr, Bebo, Craigslist
Aerospike *from* http://www.aerospike.com [49]	Associate keys with records (i.e rows); namespaces (for dataset) divide into sets (i.e tables); key index records	Auto partition, synch. replication	In-memory very fast database with disk persistence. ACID with relax options	AppNexus, Kayak, blueKai, Yashi, Chango
Cassandra *from* facebook [57]	Hybrid of key-value and column-oriented models; Cassandra query language: SQL like model	Auto partition, synch. and asynch. replication	In-memory database with disk persistence; highly scalable	CERN, Comcast, eBay, Netflix, GitHub
Redis *from* S. Sanfilippo [48]	Het of (key, value); complex types (string, binary, list, set, sorted set, hashes, arrays); key can be any binary e.g. JPEG	Auto partitioning, replication, persistent levels	Most popular in-memory with disk persistence; ACID; MapReduce through Jedis [58], r^3 [59]	Twitter, GitHub, Flickr, StackOverflow
Riak [60] *from* Basho Technologies	Data types (flags, counter, sets, registers, maps, hyperlog)	Sharding, replication, master-less [61], backup and recovery	High available; Dynamo base [62]; in-memory with disk persistence; relax consistency; MapReduce, REST-full; enterprise and cloud versions; Riak KV base Riak TimeSeries [63]	AT and T, Comcast GitHub, UKHealth, weather channel
Voldemort *from* LinkedIn [50]	Complex key-value compound objects (e.g. lists/maps); supported queries: get, put and delete; no complex query filters; simple api for persistence [64]; schema-evolution	Auto data partitioning and replication, versioning	In-memory with disk persistence, big fault-tolerant hash table; no ACID, pluggable serialization (e.g. avro, java) and storage engine (concurrentHashMap, mysql, BDB JE)	LinkedIn, Gilt
DynamoDB *from* Amazon [65]	Dynamo [62] based to support both document and key-value models [65]; secondary indexes; DynamoDB Titan: integratable graph database	Replication, partitioning, highly available, versioning	Popular, two of the four ACID properties: consistency and durability; elastic MapReduce for Hadoop; AWS SDK to store JSON	Amazon, BMW, duolingo, lyft, redfin, adroll

Table 2 Column-oriented databases

Name	Data model	Scalability	Description	Who uses it
Cassandra *from* Facebook, Apache [57]	Multi-dimensional column family is a set of rows; Partition (single or more row) key: identify a partition; Row key: identify row in column family	Partitioning, replication, availability	Multi-master no point of failure; in-memory with disk persistence; masterless; query method: CQL anf Thrift; MapReduce; secondary indexing; eventual consistency	CERN, Comcast, GitHub, GoDaddy, Hulu,eBay Netflix
HBase *from* Apache [72]	Tables have rows and columns; rows: row key and one or more columns; column: consist of column family	Auto sharding; asynchronous replication; availability	BigTable based; Hadoop Distributed File System (HDFS); MapReduce; consistent read/writes; failover support; Thrift anf REST-ful; ZooKeeper	Adobe, Kakao, Facebook, Flurry, LinkedIn, Netflix, Sears
Druid *from* http://druid.io	Columns are one of three types: a timestamp, a dimension, or a measure. Nested dimensions	Low latency; replication; sharding	Highly optimized for scans and aggregates ; MapReduce; fault-tolerant; ZooKeeper; index structures, not ACID	Alibaba, Cisco, eBay Netflix, Paypal, Yahoo
Accumulo *from* Apache [73]	Keys-values both byte arrays, timestamp as long; adds new key element of column visibility; sorts keys by element, secondary indexes	Sharding, replication, persistence, fault tolerant	BigTable based Java technology, top of Hadoop, ZooKeeper and Thrift; Map-Reduce; Zookeeper (multi-master) locks for consistency; cell-level access	US National Security Agency (NSA)

Table 3 Graph databases

Name	Data model	Scalability	Description	Who uses it
neo4j *from* Neo Technology [52]	Flexible network structure of nodes; data stored in: edges, nodes, or attributes; neo4django: an Object Graph Mapper [84]; custom data types	No direct sharding but cache [82], no replication and persistency	Most popular high performance [85], ACID, monitoring:Neo4j Metrics; query methods: Cypher, SparQL, nativeJavaAPI, JRuby	42talents, ActiveState, Cisco Securus, Apptium, BISTel [52]
AllegroGraph *from* http://franz.com	Triplestore, resource description framework (RDF) and graph database	Data replication and synchronization; Partitioning with Federation	Linked data format; brings semantic Web to Twitter; Common Lisp: dialect of Lisp; eventual consistency; ACID	Stanford, IBM, Ford, Novartis, AT and T, Siemens, NASA, US Census
ArangoDB *from* ArangoDB GmbH [80]	Native multi-data models: key/value, document, and graph data to be stored together and queried with a common language [86]	Synchronous replication, tripple store sharding	Most popular having open source license; ACID-compliant for the master; eventualy consistent [87]; annotation query language (AQL) for RDF	DemonWare, Douglas, Craneware, ictual, mobility, egress
OrientDB *from* OrientDB Ltd [81]	Multi-data models: graph and document database; custom data types	Multi-master replication; supports sharding	Highly available; SQL using pattern matching to support MapReduce, eventual consistency; ACID; Schema-less, Schema mix	Progress, UltrDNS proteus, Enel Flux Gtech, NIH

flexible to store data as a network of nodes, edges and their attributes. This also supports custom data types with its Java persistence engine. Neo4j does not support graph sharding on different nodes, rather it supports in memory cache sharding [52, 82]. The reason having that, the mathematical problem of optimally partitioning a graph across a set of servers is near-impossible (NP complete) to do for large graphs [82]. Whereas AllegroGraph is a Resource Description Framework (RDF) [83] triple store for linked data and widely used by different organizations, such as Stanford University, IBM,Ford, AT&T, Siemens, NASA and United States Census department.

Document databases

These are the most general models, which use use JSON (JavaScript Object Notation) or BSON (Binary JSON) format to represent and store the data structures as documents for the data management. Document stores provide schema flexibility by allowing arbitrarily complex documents, i.e. sub-documents within document or sub-documents; and documents as lists. A database comprises one or more collections, where each collection is a named group of documents. A document can be a simple or complex value, a set of attribute-value pairs, which can comprise simple values, lists, and even nested sub documents. Documents are schema-flexible, as one can alter the schema at the run time hence providing flexibility to the programmers to save an object instances in different formats, thus supporting polymorphism at the database level [92]. Figure 1 illustrates two collections of documents for both students and course within an academic management System. It is evident that a collection can have different formats of documents in JSON format and they have hierarchies among themselves, such as courses has an attribute books which contains a list of sub-documents of different formats.

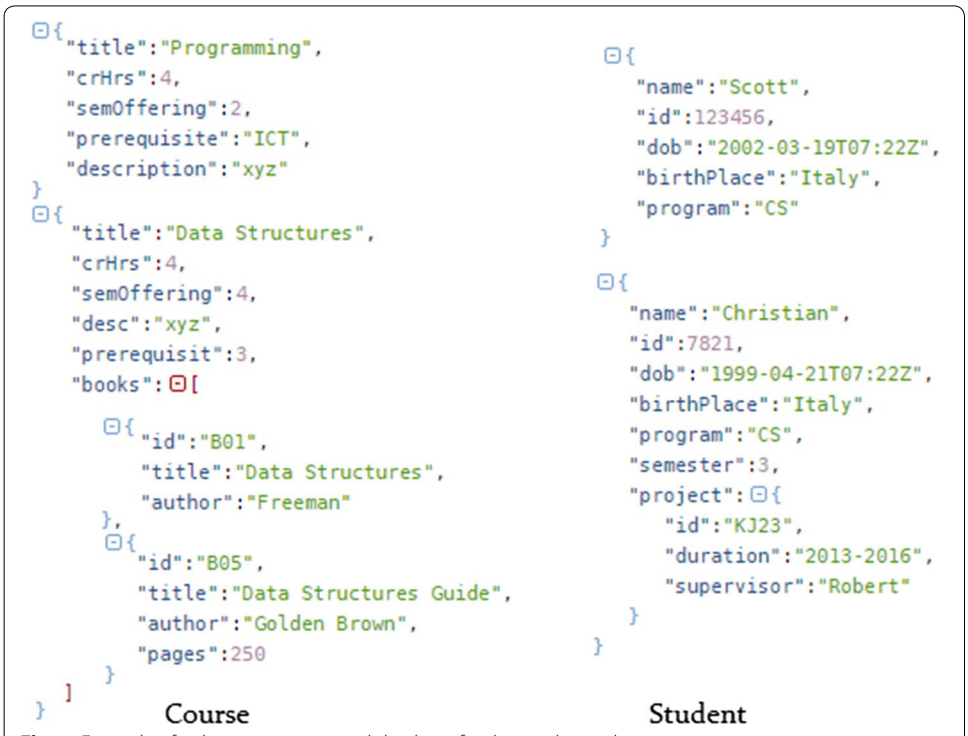


Fig. 1 Example of a documents-oriented database for the academic domain

These databases store and manage volumes of collections of textual documents (e.g. emails, web pages, text file books), semi-structure, as well as no structure and de-normalized data; that would require extensive usage of *null* values as in RDBMS [93]. Unlike key-value stores, the document databases support secondary indexes on sub-documents to allow fast searching. They allow horizontal scaling of the data over multiple servers called shards. MongoDB [38], CouchDB [88], Couchbase [89], ReThinkDB [90], and Cloudant [91] are some of the most popular document-oriented databases, as shown in the Table 4 in a categorical format along their different characteristics. Among these MongoDB is the most popular one due to its efficiency, in memory processing and complex data type features [94]. The other databases such as Couchbase, ReThinkDB, Cloudant and CouchDB do not offer in-memory processing features; although the former three offer a list of data types. MongoDB query languages more like the SQL of RDBMS, so is easy to use for the programmers. MongoDB is good for the dynamic queries, which the other document-oriented databases lack, such as CouchDB or Couchbase [95]. Besides this there are different object relational mapping middlewares available [96], to define out of the box multiple schemas depending upon the application requirements. The object nature of MongoDB documents makes this mapping even more fluid and fast, such as while using Mongoose [96] or Morphia [97].

Document databases: MongoDB

MongoDB, created by 10gen in 2007, is a document oriented database for today's applications which are not possible to develop using the traditional relational databases [98]. It is an IoT database which instead of tables (as in RDBMS) provides one or more *collection(s)* as main storage components consisted upon similar or different JSON or BSON based documents or sub documents. Documents that tend to share some of the similar structure are organized as collections, which can be created at any time, without predefinitions. A document can simply be considered as a row or instance of an entity in RDBMS, but the difference is that, in MongoDB we can have instances within instances or documents with in documents, even lists or arrays of documents. The types for the attributes of a document can be of any basic data type, such as numbers, strings, dates, arrays or even a sub-document.

MongoDB provides unique multiple storage engines within a single deployment and automatically manages the movement of data between storage engine technologies using native replication. MongoDB 3.2 consists of four efficient storage engines as shown in Fig. 2, all of which can coexist within a single MongoDB replica set [99]. The default WiredTiger storage engine provides concurrency control and native compression with best storage and performance efficiency. MongoDB allows both the combinations of in-memory engine for ultra low-latency operations with a disk-based engine for persistence altogether.

It allows to build large-scale, highly available, robust systems and enables different sensors and applications to store their data in a schema flexible manner. There is no database blockage, such as we encounter during *alter table* commands in RDBMS during schema migrations. However in rare cases, such as during the write-intensive scenarios in master-slave nature of MongoDB there may be blockage at the document level or bottleneck to the system if sharding is not used, but these cases are avoidable. MongoDB

Table 4 Document-oriented databases

Name	Data model	Scalability	Description	Who uses it
MongoDB *from* http://10gen.com, MongoDB Inc. [38]	JSON-like hierarchical documents with or without schemas, object mapping, BSON	Sharding, replication and persistency	Most popular JSON document store, ACID, MapReduce, primary secondary indexing, eventual consistency, RESTful	Expedia, Bosh, MetLife, Facebook, comcast, sprinklr
CouchDB *from* Apache [88]	Native JSON-document store; types: strings, numbers, dates, ordered lists and associative arrays	Multi-master replication,	JavaScript as query language using MapReduce, and HTTP for an API, multi-version concurrency, MapReduce, ACID, eventual consistency	meebo, AirFi Sophos, BBC, npm CANAL+
Couchbase *from* Couchbase, Inc. [89]	Multi-model: key/value store, document-store; JSON documents	Sharding, master-master and master-slave replication	CouchDB based with Memcached-compatible interface, eventual consistency, limited ACID, eventual consistency, RESTful HTTP API	Informatica, Joyent, intel, Wipro, Google, Simba
RethinkDB [90]	JSON documents with dynamic schemas	Sharding, master-slave replication	Push real-time data; RethinkDB Query Language (ReQL);Hadoop-style MapReduce; primary&secondary indexes; Not ACID	Jive SW, Mediafly, Pristine Platzi, CMUNE, Wise.io
Cloudant *from* IBM, Apache [91]	JSON based flexible documents	Sharding, master-master & master-slave replication	CouchDB based; primary and secondry indexes; MapReduce; Eventual Consistency; RESTful HTTP/JSON API	Samsung, IBM, Expedia, DHL, Microsoft, Pearson

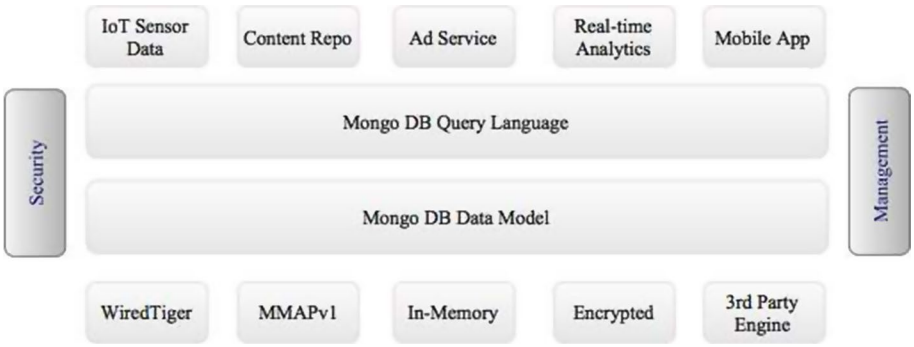


Fig. 2 Flexible storage architecture, optimising MongoDB for unique application demands [99]

enables horizontal scalability because table joins are not as important as they are in the traditional RDBMS. MongoDB provides auto-sharding in which more replica server nodes can easily be added to a system. It is a very fast database and provides indexes not only on the primary attributes rather also on the secondary attributes within the sub-documents even. For the cross comparison analysis between different collections we have different technologies, such as aggregation framework [100], MapReduce [101, 102], Hadoop [14, 19] etc. These processing techniques will be targeted in future and are not currently focused in this paper. Next we discuss the data modeling methodologies, which is followed by a data model description we used for the storage of the real-time temporal data of ANT+ sensors in the healthcare domain.

Modeling aspects

The representation of the concepts of data structures required by a database is called a data model. A data model determines the logical structure of a database and basically decides in which manner the data can be stored, organized, and manipulated. Such data structures decide the data objects and their relationships and the semantic data rules they govern. In software engineering data modeling is a process to create data models for an information system. There are mainly three types of data model instances, i.e. *conceptual, logical* and *physical* [103]. Conceptual data modeling defines the semantics of an application domain, to elaborate the scope of the model, in order to identify the various entities and relationships needed in an application. Logical data model is similar to conceptual, but with more details, which defines the structure of a domain of information. It is also called as a schema model, as it describes the objects with their properties and relationships to determine the structure of the data, whereas the physical data model concerns the physical storage of the data.

Although NoSQL approaches provide schema flexibility to the developers which allow them to store different formats of same entity's instances in a single storage collection, but for the rationality, sanity and demonstrating the storage structure they still have to follow some sort of format logically, at least at the application level and then to store the data in the selected format(s). In this way one can define as many schema formats for a single entity and the same *collection* will allow the storage to all of them, given that a logical rationality has been maintained through out the definitions, as shown in Fig. 3. Applications treat each schema differently based upon different logics using advanced

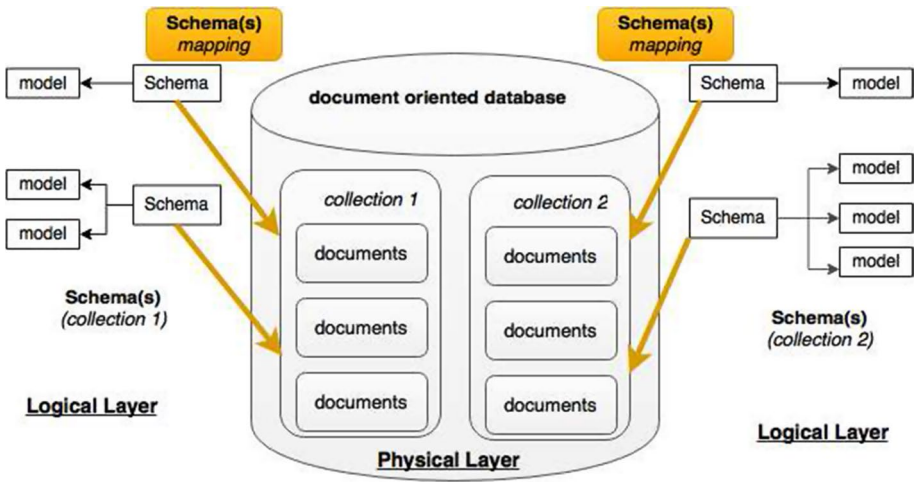


Fig. 3 *Collections*, *documents* and *sub-documents* in document oriented databases can refer to one or more schema models

programming, such as object oriented principles in the form of polymorphism, classes inheritance or object arrays etc. Hence developers eagerly define and model the schemas that yield robust characteristics for their problem domain in general. The initial research regarding NoSQL database modeling and insights is provided in [104, 105]. Bugiotti et al. deal with NoSQL modeling approaches in general [106]. How to have NoSQL related approaches inside the relational databases is discussed in [107]. For the initial thoughts about NoSQL modeling and its comparison between traditional relational models and NoSQL models is provided in [102]. A study about normalization and de normalization during the data modeling is done in [108].

Note that because the document-oriented data models allow sub-documents embedding, so hierarchical principles are their core features; and this create opportunity for the sub-entities to get embedded and to play the role. A collection can refer to one or more models, as shown in Fig. 3. A schema can have a single model representing an entity or can have multiple models representing more entities (may be hierarchical). For example in Fig. 1, we model two course documents for a *course* entity. The document titled as "Programming" is quite simple, whereas the document titled as "data structures" has a tree structure where a sub-entity *books* is shown at the second level. Similarly, two structures are shown for the *student* collection, where the first with name "Scott" is simple and the second with the name "Neo" contains a hierarchy. Therefore the collections having different schemas are actually having different format logics and must be treated differently.

What and how to model a schema for NoSQL or RDBMS ?

New technologies such as XML and NoSQL databases have put doubts on the usefulness of data models, but this is not the case because the significance of data models will always remain inevitable to understand and demonstrate the logical and storage structures of the data [8]. Regardless, NoSQL databases are schema flexible and support multiple schemas but in reality they actually support multiple variants of schema, which is

actually some sort of structural representation, if we differentiate and treat them individually. Therefore for such databases the terms like schema flexible would be more appropriate than schema less.

Mainly most RDBMS are built around one particular data model, although it is possible for a RDBMS to provide more than one data model to an application. Such data models are fixed structure and require a database locking for a certain period of time upon migration processes, such as from one version to another version, adding more attributes to a table etc. Also RDBMS use Data Definition Languages to generate databases from the pre defined schema models [103]. Database designers use the traditional approach of Entity Relationship Diagrams (ERD) to concentrate on the database structures and constraints during conceptual database design [109]. ERDs are used to identify the main entities and their attributes for which the information is required to store in the database. Besides this it is also required to identify the relationships and relationship types between the entities, which lead to a natural and logical grouping of the attributes into relations during the mapping process from ERD's to the relational schema [103]. In order to define the linking of different entities between each other, their relationships are investigated in context to the measurement of the cardinalities.

How good is a schema design?

To be rational about a schema and speaking in context of the RDBMS, during the mapping process, we still require some formal efforts to analyse why a relational schema, with a grouping of attributes, may be better than another. This *goodness* or measurement of the quality of a structural or relational schema design follows the procedures of checking data redundancy and data dependency features extensively [110]. For a technical application domain, we should investigate and compare the available data modeling approaches to chose the best.

In general, we should consider all the requirements and constraints related to a problem domain, such as temporal, streaming, distributed aspects, multiple messages formats and their attributes. Modeling for the schema flexibility this investigation needs intensive care; to allow the storage of multiple structural schemas. This way one main goal of big data can be achieved i.e. *variety*. It is to note that schema flexibility, such as provided by MongoDB, can permit all vendor devices to store data simultaneously with the time-stamp regardless of different message formats. In context of the IoT perspective, it supports storage independence of data related to any "thing(s)".

Goals of normalization

Katsov in his online guide [104], discusses NoSQL data modeling techniques for all the NoSQL management frameworks in general. He says that denormalization and aggregation are the main fundamental design drivers, beside thinking about how the application would address the end user queries. RDBMS do normalization because of three goals, i.e (i) To free the database of modification anomalies, i.e. to update only one row in one table against a wrong instance entry, (ii) to minimize the redesign of the database, and (iii) to avoid biasness towards any particular access patterns. For the third point MongoDB do not care at all, therefore with MongoDB we should not worried about being biased to a particular access pattern.

One of the big advantage of RDBMS is that they keep the data relational, normalized and enforce the foreign Keys for the data consistency. But in MongoDB it is only upto the programmer to keep such a consistency, therefore MongoDB do not provide such a guarantee. Whereas on the other side, MongoDB allow the embedding of the entire document and in this case do not have to care about even about the foreign keys at all. With normalization RDBMS usually perform transactions based upon ACID principles, which MongoDB does not have to care about. However it allows atomic operations at a document level.

Normalization or embedding

Normalization was first introduced by E.F. Codd in the field of rational databases and is explained in many of the database related text, such as [110, 111]. In RDBMS the traditional practices are to properly normalize the database because it increases the data integrity and efficiency of the space utilization [103]. In denormalization we reject the splitting of the physical tables, and keep the data together in a single database table which have frequent access, to reduce the query processing time, as in this way we can avoid joins as well and the disk scanning at different location to answer the queries. Following can be the database schemas in normalized forms; for the de-normalized student and course collections depicted in Fig. 1.

```
Student (ID, name, DOB, birthplace, program, semester, PID)
Project (PID, duration, supervisor)
and
Course (title, crhrs, SemOffering, prerequisite, desc, bookID)
Book (ID, title, authorID, pages)
Author  (authorID, name)
```

Kanade et al. [108] did a study for NoSQL databases with both normalized and denormalized forms using a similar dataset, and have found that the embedded MongoDB data model provides a much better efficiency as compared to a normalized model. Therefore, Database designers have confronted that tight normalization degrades the system performance, hence they recommend to denormalize in many cases. Denormalization supports the data read performance, as the spinning disks have a very high latency; therefore the idea is to keep the data close and together to help avoid disk read overheads and to decrease the latency. Therefore, denormalization is a key feature of document databases to support efficient data retrieval and to optimize the data storage capacity by allowing dataset hierarchies.

Schema modeling using object relational mapping (ORM)

There exists a specific driver for each programming language, including few languages which are still not much popular, one can see here [112]. Recently a new trend is seen which provides facility to the programmers to develop NoSQL database applications through the usage of Object Relational Mapping (ORM) mechanisms. ORM is related with the mapping of programming language objects to the persistent data storage in the database and then to the objects usage in the application [113]. In this way the data becomes more fluid and natural both for the application as well as the used programming language. There are different ORM available for Node.js to model schemas for

the MongoDB database, such as Mongoose [96, 114], Morphia [97], Iridium [115] and Node-ORM2 [116]. Among these Mongoose provide built-in automatic validation of the data which you are inserting/updating. Using Mongoose we can also pre-define events before a document gets saved. In this way we can place the code where it should be next to the document logic [117].

A beauty of MongoDB is that its documents are objects themselves, which performs this process even more easily, hence the MongoDB architecture lends itself very well to ORM as the documents. One example is mongo-Java-orm or "MJORM" a Java ORM for MongoDB [97, 118]. A plain old Java object (POJO) is an ordinary Java object, not bound by any special restriction or bogged down by framework extensions [119, 120]. One of the main advantages for using POJO is its decoupling from the infrastructure frameworks [119]. POJOs accelerate the development process by separating the persistent object properties and business logic; hence facilitating the programmers to concentrate on one thing at a time; i.e, between the business logic and persistence storage. Other variants of POJO are described in [121–123], and the .Net framework variant of POJO is POCO [124].

ORM based on mongoose

Although storing accessing and manipulating data in MongoDB is not difficult, but still application developers find it more convenient and flexible to have a mapping layer between the application and the NoSQL database, as discussed above. Such a service is provided by an application layer middleware, which is mostly written in JavaScript and is designed to work in an asynchronous environment; and helps in validation, type casting, object mapping, document mapping, query building and building business logic. It can deliver direct, schema-based modeling solutions to an application in MongoDB database, for example Mongoose [96, 114]. We may think that, either if it consists upon the concepts which are not part of the MongoDB framework but are out of the box for a database management system. There are other ORMs also available, such as Mongorito [125], Ming [126] and Backbone-ORM [127].

Time series data modeling in MongoDB

As streaming data contains timestamp values within each sequenced message element. MongoDB is a great fit to handle timestamped data and many organizations are using it [128]. Some proprietary services are also available and popular in the community, like MongoDB Management Service (MMS) which models and displays the time-series data as a dashboard [129]. This facilitates a developer to develop visualization and alerts on different datasets using metrics.

Schema design for time series: a case study

It is beneficial to have a schema mapping out of the box; thus facilitating the programmers in developing scalable distributed systems more rapidly and flexibly. One can design as many schemas as possible for MongoDB as NOSQL databases are flexible for this purpose. But to be rational there is a tradeoff between different schemas, and as we learned that the level of normalization or de-normalization is important. This section discusses an approach described in a blog, which models time-series data for MongoDB

with respect to different levels of normalization [128, 130]. Before explaining the design approach for a NoSQL database; first let us observe, how an RDBMS handles time-series data. This is quite straightforward, i.e. by storing each event as a row within a Table. For example, a monitoring system stores a temperature sensor measurement in relational format, as shown in Table 5: But, this approach will save a single document in each row for MongoDB.

```
{ timestamp: ISODate("2016-04-10T22:04:23.000Z"),
  type: "temperature",  value: 24.1},
{ timestamp: ISODate("2016-04-10T22:04:24.000Z"),
  type: "temperature",  value: 24.1},
{ timestamp: ISODate("2016-04-10T22:04:25.000Z"),
  type: "temperature",  value: 24.0}
```

This *doc-per-event* approach does not take advantage from the document embedding, and there will be lot of latency in reading the data; for example around 3600 reads for 1 min data, if the sensor transmits one message per second [130]. An alternative is, to have a single *document-per-minute* and to save the temperature value against each second. Such an approach, as shown bellow, is better and more optimized in the context of the storage as well as data retrieval.

```
{ timestamp_minute: ISODate("2016-04-10T22:04:23.000Z"),
  type:"temperature", values:{0:25.1, ... 23:24.1, 24:24.1, ... 59:22}}
```

This model contains sub-documents as values, which are embedded in the main document. To store one value per second (against a minute), we can simply represent each second as separate fields between 0 and 59, as shown in the above code listing. Such kind of approaches affect the efficiency of systems, in context of the number of inserts and updates. For example, for one hour read 60 total documents need to scan; and some systems copy the entire data to a new location against an update, which is not robust when the data is in large quantity. If the database also has to perform indexing after the copying process, it will affect adversely. A feature of MongoDB is that it can manage in place field-level updates as long as the size of the document does not grow more than a certain limit (i.e, 16 megabytes), which may also contain sub-documents [131]. However GridFS API of MongoDB provides flexibility to store a document more than the maximum size [132]. Conclusively, because field level updates are more efficient, so the document-embedding is better than the relational design. Writes will be faster as: for updates (one per second) than for inserts (one per minute) [128]. Because instead of writing a full document at a new place we will actually request a much smaller update that can be modeled as described below for the temperature collection:

Table 5 TimeSeries data storage in a traditional RDBMS

Timestamp	Temperature
2016-04-10T22:04:23.000Z	24.1
2016-04-10T22:04:24.000Z	24.1
2016-04-10T22:04:25.000Z	24.0

```
db.temperature.update({
  timestamp_minute: ISODate("2016-04-10T22:04:23.000Z"),
  type: "temperature"}, { $set:{"values.23": 24.1 } } )
```

A more compact approach is to store per second data at hourly level, i.e. *doc-per-hour*. It has two variants: (i) by second, and (ii) by minute. In the former case seconds are stored from 0 to 3599 for an hour. This approach cause extra workload during update operations, because to update the last second 3599 steps are required. To avoid document movements pre allocation of the structure space is recommended. However the later case stores per-second data at the hourly level but with nesting documents for each minute. This approach is also update driven, but to update the last second it requires 59 + 59 steps. The following MongoDB query depicts the later model, and it is followed by an update query to modify the last second.

```
{ timestamp_hour: ISODate("2016-04-10T22:04:23.000Z"),
  type: "temperature", values: {
    0: { 0:25.1, ..., 23:24.1,  24:24.1, ..., 59:22 },
    ...,
    58: { 0:25.1, ..., 23:22,  24:22, ...,  59:22 },
    59: { 0:25.1, ..., 23:24,  24:24, ...,  59:24.2},  }}


db.temperature.update( {
    timestamp_minute: ISODate("2016-04-10T22:04:23.000Z"),
    type: "temperature" }, {$set: {"values.59.59" :24.5 }})
```

and, to update the last second:

In the next section we select a real time use case regarding sensor network data and shall model the temporal properties of the message generated from different sensor devices.

Temporal modeling for an ANT+ sensor use case

To preserve the NoSQL schema flexible goals, i.e, data variety; the model should reside outside the database and should not be tied with it; and should provide a mirror to the programming language objects to get mapped seamlessly for persistent storage. The less energy resourced sensor devices do not communicate directly with the web servers rather they use a gateway, display device, PC, service or a running application. In our system, a monitoring Windows service transmits the ANT+ sensor data in near real-time over the WebSockets protocol, where as the NodeJS [96] context data server accepts, processes and stores the JSON messages in MongoDB database according to the defined schema models. The context data server is a NodeJS based JavaScript engine, which provides a WebSockets based interface for the communication middleware layer and data processing. The server uses the Mongoose based schemas for the direct document mapping to persistently store the JSON messages. Figure 4 depicts a general view of the real-time distributed system architecture for the prototype; whereas the complete architecture components are explained in [133]. For temporal storage, data integration and schema evolution the server executes relevant ANT+ device profile related Mongoose schema based storage algorithms against the specific event notifications.

To illustrate the approach through a prototype, we deal with three ANT+ wearable sensor categories, such as heart rate, foot pod and temperature to monitor a patient's body parameters [34, 133]. Next we shall rationally observe the temporal modeling

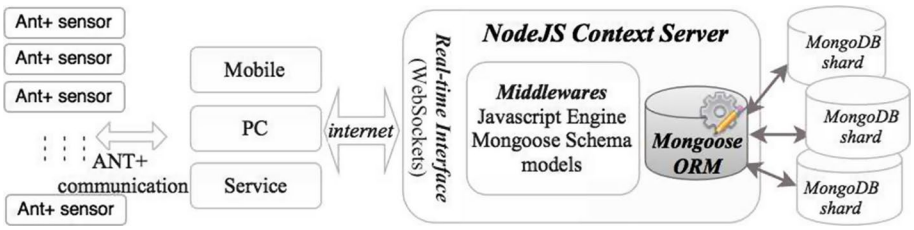


Fig. 4 MongoDB storage architecture: WebSockets based real-time interface with middlewares for JSON message processing, storage and evolution according to the schema model(s) [133]

perspectives of the ANT+ sensor data and will learn the Mongoose schema based algorithm to evolve the schema during the data integration process.

Collection identification vs. entity and entity relationships' identification: What to model?

An important question during document-oriented schema flexible modeling is what to model for an application. The *ANT+ message protocol and usage* document guides us to work with sensor nodes and to identify the device's messages in general [134]; but for more detailed understanding of each device type's data semantics we should follow the message format principles described in specific *ANT+ Device Profile* documents, such as *Environment* profile for temperature sensor [135], *Stride Based Speed and Distance Monitor* profile for footpod sensor [136] and *Heart Rate Monitor* profile for heartrate sensor [137]. A device profile is a defined standard, which confines all the vendors to manufacture their devices according to the rules and principles defined by ANT+ Alliance. Therefore, if a profile indicates about transmitting a message it means that all the vendor devices will transmit it, hence we can perform in general similar operations to the data of a specific type of all the sensors that meet a particular profile. ANT+ devices transmit monitoring data in the form of *data messages* which are specific to a profile type. Next we discuss the software engineering process of data modeling for the ANT+ sensors. We shall be rational during the time-series sensor data modeling for the MongoDB document oriented database, and shall also discuss the relational modeling (i.e., denormalization) with respect to our healthcare use case for the comparison purpose. The following descriptions expose the attributes of the chosen sensors.

Temperature It transmits current, minimum (in previous 24 h) and maximum (in previous 24 h) temperature in Centigrade (C°). With each measurement it also transmits the transmission information, timestamp and the number of event counts—which increments with each measurement.

Footpod A footpod measures speed, cadence, distance, duration, strides accumulated and distance accumulated in meters/second, strides/minute, meters, seconds, number of strides and meters as measurement units respectively.

Heart rate monitor It measures and transmits the heart rate in beats per minutes (bpm), with heart beat variability, event count and timestamp.

If we consider the profile type as an entity then such data messages will be the attributes of that actually. The three sensors are actually the instances of the device entity, but should be treated separately because they do not share attributes. So we allocate a separate MongoDB *collection* to each sensor category and define its schema for the

document storage. It is to note that a sensor will always have a unique serial number, manufacturer and model number, although the devices can share manufacturer ids, device types etc.

However, there are messages which most devices transmit, and those are not specific to a profile standard; therefore they can be treated as separate common entity with a referenced instances to the relevant sensors. Such common messages are called as *common data pages* or *background messages* [138]. For example devices transmit their information such as battery status, message rate, manufacturer or hardware information etc. We treat such messages differently during processing and storage depending on the schema model. Our storage server accepts two types of JSON messages from the sensors: (i) per-minute timestamped holding sensor data, and (ii) per-hour timestamped holding device information and other healthcare data depending on the use case.

Because a user can use more than one sensor at a time, so user ID will be an attribute of each sensor instance. Based on these guidelines an entity relationship diagram is depicted in Fig. 5, which shows the possible attributes, relationships, relationship types and primary keys. Then a relational model is drawn in Fig. 6 in 1st and 2nd Normal Form (1NF) [103].

Discussion w.r.t RDBMS In context to the RDBMS, this model although supports sensor data storage but it has many flaws, like many joins are required while querying the data for a patient at a particular time. There will be many null values because sensors will not transmit all the attribute values altogether. If we want to resolve this issue then we will end-up only with three columns in each of the sensor table i.e. *SensorTable(timestamp, value, event count)*. This is similar to saving every event message, but this will not be robust for SQL as well as for NoSQL because there will be insertion and update issues besides the computation processing issues, such as aggregates etc.

Discussion w.r.t MongoDB Considering each entity as a possible document collection will require joins between them which document-oriented databases do not support natively. However it is possible either within the application or by using data processing analysis tools such as aggregation framework [100], MapReduce [101, 102], Hadoop [14, 19] etc. With document-oriented databases we do not have problems with the null values because of schema flexibility, however interestingly, in such a case we will again end-up with the same every event storage sort of structure, as discussed in the previous

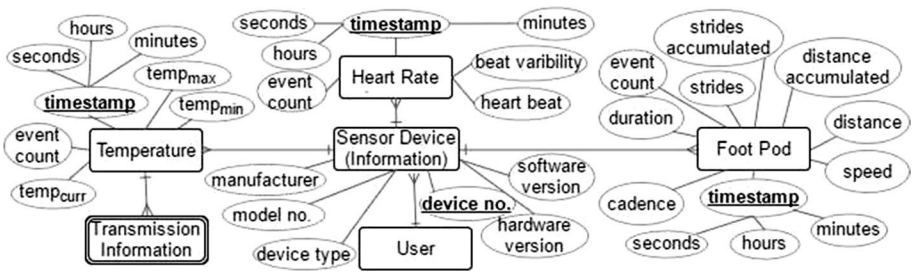


Fig. 5 Entities and their attributes with respect to the discussed use case

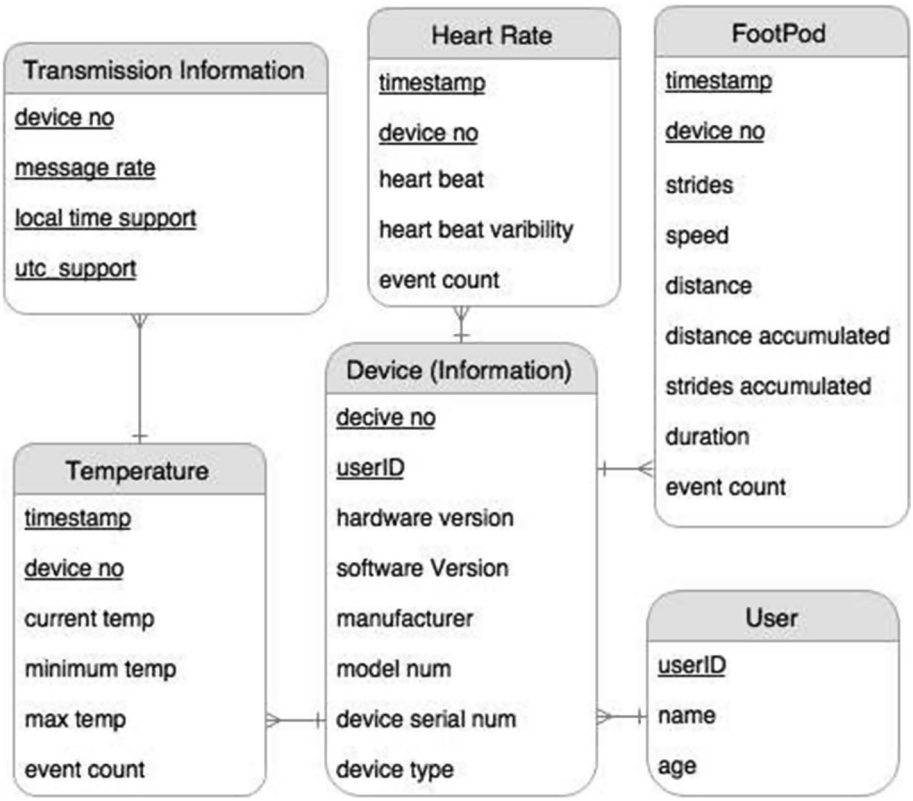


Fig. 6 Data model with entities and their attributes for the use case

discussion. Such a scenario although supports the event message storage but is not robust with respect to processing and efficiency. We shall discuss it in the next sections.

Timestamped based cardinalities

A message rate is measured in cycles per second and its unit for frequency is Hertz (Hz). We discuss the data messages of the three device profiles with respect to their allowable message rates. The discussion provides us to calculate that how many instances of a certain sensor *data message* will appear against a time value, such as seconds, minutes and hours; and Table 6 summarizes the discussion.

Temperature This environmental device broadcasts only at minimum 0.5 Hz or the maximum 4.0 Hz message periods therefore once per 2 s or once per 0.25 s respectively.

Footpod A device can receive data at the full rate (4.03 Hz) or at half of this rate, therefore once per 0.248 s or once per 0.498 s respectively.

Heart rate monitor It transmits at the full rate (4.06 Hz) or at one half or one quarter of this rate; therefore the data can be received four times per second, twice per second, or once per second.

Common messages or *background messages* appear after 64 *data messages* of each device profile type [138]. In context to the time it depends upon the message rate, for example, heartrate with a message rate of 2.03 Hz will transmit almost 2 messages per second, therefore in this case a *background message* will appear after 30 seconds

Table 6 Data messages cardinalities with respect to allowable message rates

Device profile	Message rate (Hz)	Seconds	Minutes	Hours
Temperature	0.5	0	30	1800
	4.0	4	240	14,400
Foot pod	2.01	2	120	7236
	4.03	4	241	14,508
Heart rate	1.02	1	61	3672
	2.03	2	121	7308
	4.06	4	244	14,616

approximately, whereas with the message rate of 1.02 Hz (which is the minimum possible message rate) we shall never receive two background messages in a single same minute roughly. It is also to notice that such information do not change rapidly, so we may not require to store every received message, hence it is better to store it either after every 10, 15 or 20 minutes or only once per hour.

Normalization or embedding: How good is the model?

Designing a better model is the main goal, and its process encounters the tradeoff between normalization and denormalization. For this we observe the examples for three different approaches to model the time-series data in MongoDB, i.e. document-per-event, document-per-minute and document-per-hour. All these three approaches are different in context of document embedding, and result in different storage space and hence data retrieval. We have already discussed the document-per-event scenario previously in the same section during the discussion w.r.t RDBMS. We learned that in such a case the outcome will be the three columns in each of the sensor table i.e. *SensorTable (timestamp, value, event count)*.

Data Stream Management Systems (DSMS) are prone to the message arrival rates, and need adjustments upon any change in the rate. The current prototype does not support this automatic adjustment. Therefore, to be simple in the current prototype the least message rates of the sensors are used. We used the document-per-hour approach with the nested minute documents carrying 0-59 second documents. This approach requires one insert initially for the hour document, and then only updates are required for each new second value. To update the last second it requires maximum $59 + 59$ steps. Next section presents the Mongoose based schema model based upon the chosen approach for the heart rate profile.

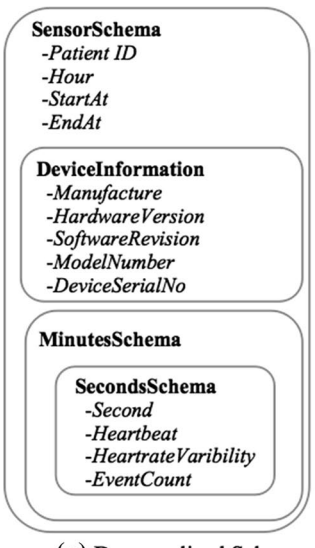
Mongoose doc-per-hour schema for heart rate sensor

Since the above three versions of the embedded document storage differentiate the data into hour, minute and seconds, therefore before going to present the Mongoose schema, the first step is to separate the hourly, minute and second level data from the heart rate sensor payload. As stated already, *background messages* mostly relate with the hardware information of the sensor device itself, and appear only after 65 other data messages [139], therefore we keep them at the hourly level for the initial prototype. Since hour-level document will be a separate document therefore it will also keep a reference of the patient, as well as the start and end timestamps.

The minutes document will keep seconds' sub-document inside it. Each second's sub-document will have the heart rate data, i.e heart beat, variability and event count. The ERD in Fig. 5 and data model in Fig. 6 depict these attributes. The Fig. 7a shows an in general denormalized model for the heart rate sensor, in which the main SchemaSensor (for hours) contains the SchemaMinute (for minutes), which further contains SchemaSecond as the second's sub-document schema. Alongside the Fig. 7b shows an in general Mongoose schema pseudocode for the denormalized schema. We can observe that most of the values will be at the seconds level, i.e, Heart beat, heart rate variability and event count as the measurements.

Mongoose schema base temporal data storage and evolution

We require to define a finite set of sequenced steps, to store a JSON message instance related to an ANT+ profile type. They must perform logically correct updates to a database, leaving it consistent and concurrent after the manipulations. Following is the description of the storage algorithm which accepts JSON formatted sensor data messages to store and evolve the chosen schema model for the MongoDB. This is based upon the Mongoose object mapping middleware, which is first used to define the time-series schema model, as in Fig. 7 depicting the rationality and storage structure. The algorithm will use the object of the Mongoose schema model to perform the inserts and the updates to the document database, and such manipulations must be performed in a manner that the new data integration would result into a storage formation as defined and desired in the model. For this purpose the manipulations need to be done in a controlled logical manner and require an algorithm for this purpose. For the above stated Mongoose schema i.e, SchemaSensor, the set of tasks which the *NodeJS Data Server* will execute once for each new specific sensor event, for the JSON formatted sensor payload, is provided in the flowchart depicted in Fig. 8. This flowchart shows the logic of the algorithm and emphasises the individual steps and their interactions in a controlled manner



(a) Denormalized Schema

```
var SchemaSecond = new mongoose.Schema({
                second : Number,
             heartbeat : Number,
          hrvaribility : Number,
            eventCount : Number});

var SchemaMinute = new mongoose.Schema({
    minute:Number, seconds:SchemaSecond});

var SchemaSensor = new mongoose.Schema({
    patientID:String,
    deviceInfo:{ manufacture:String,
                 hardwareRevision:String,
                 softwareRevision:String,
                 modelnum:String,
                 deviceserial:String },
       hour : Date,
    startAt : Date, endedAt:Date,
    minutes : SchemaMinute
});
```

(b) Mongoose Schema Code

Fig. 7 The denormalized schema and the Mongoose schema for the heart rate sensor

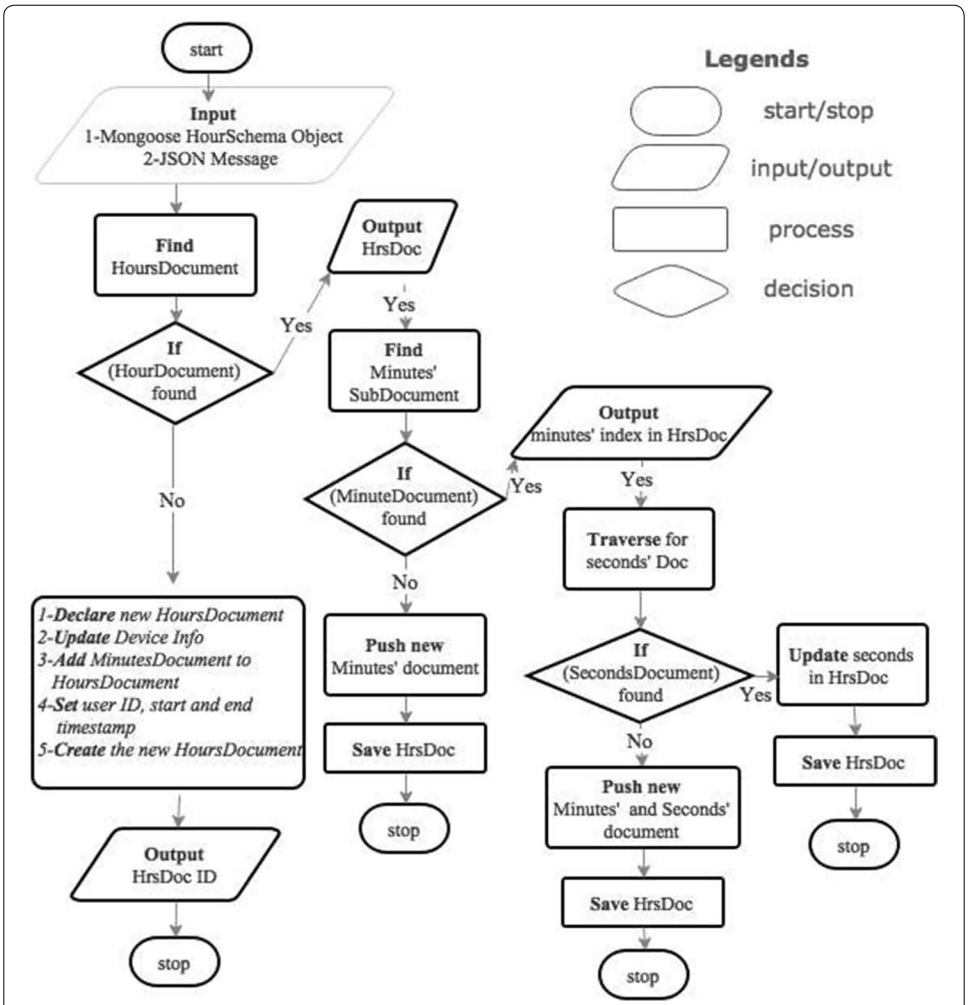


Fig. 8 The flowchart of the Mongoose based schema evolver algorithm

from one action to another. This also depicts how the data flow will take place within the doc-per-hour schema based upon the Mongoose object mapping middleware. Figure 8 is preceded with JavaScript code presented in listing , which is the implementation of the data integration and schema evolver algorithm defined in the flowchart.

Listing 1: Document-Per-Minute Algorithm

```
        INPUT
 Schm  – is a Mongoose Schema object as in Figure 8
 args  – is a variable with set of arguments in JSON
   rq  – is a request object
qryHr  – is an args based query criteria to find hour document
hrDoc  – is a JSON object representing Hour-Doc

qryMin  – is a query criteria to find minutes sub-doc within hour-doc
minDoc  – is a JSON object representing minute's sub-document
secDoc  – is an array of JSON objects representing second's sub-document

        RESULT
HrDoc  – is an embedded Document-per-Hour JSON

 1 require 'mongoose'
 2 Schm = require('sensorschema.js')
 3 Procedure  DocMinuteStorage (rq)
 4          args  = parse (rq) for JSON
 5         Schm.findOne (qryHr, hrDoc)
 6         if(hrDoc not NULL)
 7                Schm.findOne(qryMin, minDoc)
 8                 if(minDoc not NULL)
 9                      traverse for secDoc
10                       if no secDoc add it and return
11                       else secDoc
12                           secDoc.push (sensor second data)
13                           save hrDoc and exit
14                  else
15                       hrDoc->minutes as array in minArray
16                       minArray.push (sensor minutes data)
17                       save hrDoc and exit
18         else
19          create new nHrDoc using Schm
20          if(args is NULL)
21                    nHrDoc.device = NULL
22          else
23                    nHrDoc.device = {
24                        hwrevision : args.HWRevision, swrevision : args.SWRevision,
25                        manufacture : args.ManufacID, modelnum : args.ModelNum,
26                        deviceserial: args.DeviceSerial}
27                    nHrDoc.minutes = {minute : minNum,
28                         seconds : [{ secnd : datetime.seconds(), sensor second's data ]}
29          set patient and time info in nHrDoc
30          Schm.create(nHrDoc)
```

The temporal storage algorithm will use a list of input variables, such as Schm: a Mongoose Schema; args: a set of JSON arguments; rq: the request object; qryHr: the query criteria to find the hour document; hrDoc: the JSON object representing the Hour document object; and similarly the criteria for minutes along the minutes and second sub-document variables. The sensor schema in JavaScript will be imported first and will initialize the main schema object variable i.e., Schm (lines 1–2). The Schm object will be the mapper between the language constructs and the persistency of data in the MongoDB, and will help also in evolving the database according to the desired temporal schema logic. It first queries the database for the hours-level document based upon the time-interval provided in the request object (i.e, rq in line 4). The (qryHr) is a MongoDB query (in line 5) having the parameter and the projection parameters to fetch the relevant hour's document. For example the following code constructs the query parameter to fetch the document where the hrmHour is equal to dateHours and patient is userid.

```
var userid = store.get(‘curruser’);
var query ={ "hrmhour" : dateHours, "patient" : userid};
```

In Asynchronous JavaScript programming the functions return the results in the last argument i.e., the callback; therefore the result will be returned in the hrDoc object. The listing is self explanatory and is having correspondent with the flowchart. If the hrDoc is

found in line 6, the algorithm will go on searching for the minutes document, as in line 7; otherwise will move to line 18 for a new declaration and creation of the hours document.

Schema for foot pod and temperature and in general approach

The same kind of schema as we saw for the heart rate sensor is possible for the foot pod and temperature sensor. The difference will be mainly for the second's level values. The denormalized schema design for the foot pod sensor, the seconds' level documents will hold the values regarding the distance, duration, speed and cadence; where as in the case of temperature this will hold the values for low, high and current temperature. Similarly the data integration and schema evolving algorithm will also have the similar kind of sequence of steps for these sensors, with some modifications in context to the seconds' level documents and other profile specific values.

Issues

In MongoDB schemas are sometimes prone to update problems related with the deep nested arrays. For example, the "seconds" dimension contains deep nested arrays of JSON objects. When this is periodically updated upon receiving values, the algorithm evolution sometimes result in an error. The author tried to investigate the problem and found that this is because of the limits in the MongoDB engine in context to updating the deep nested schemas having arrays. The engine currently does not support several positioning operator (i.e, "$") based updates to the nested arrays.[1] Only a limited number of positional operators are currently being supported. During the prototype testing the schema and relevant algorithm has worked successfully, and only misses few values because of this. The problem is discussed in an online thread,[2] and has solutions that either to avoid deep nested schemas, or to wait for a fix. Speaking in general the approach presented in this paper is still useful and valuable for usage to define rational document-oriented schemas and to develop relevant algorithms for data integration and evolution. The next we discuss the related work regarding temporal aspects modeling for sensor data.

Related work

Some related work in context of data modeling for the NoSQL databases is already pointed out very briefly during the previous subsections. A lot of data modeling studies has already been conducted by different researchers for NoSQL databases, but non has provided in depth study for the ANT+ sensor data especially to preserve the data based on temporal properties, therefore this research is novel in context to the schema flexible time series of the ANT+ data. The authors in [140] and [8] make models for the NoSQL databases, such as MongoDB; and present both relational and non-relational database queries to have a comparison between them. While using simple select queries, the former researchers present that joins are not required during NoSQL based retrieval, whereas they are required during SQL based approach. More using MongoDB queries the data is stored in single document or if needs to store in different documents then

[1] https://jira.mongodb.org/browse/SERVER-831.

[2] http://stackoverflow.com/questions/14855246/multiple-use-of-the-positional-operator-to-update-nested-arrays.

documents are related by using reference fields. The latter approach present a schema modeling case study for both relational and non-relational databases. They explain that each NoSQL database has its own query language, such as CQL (Cassandra Query Language) for Cassandra, MongoDB Query Language for MongoDB, Cypher Query Language for Neo4j etc. The latter research presents more general results and show query syntax for three databases, i.e PostgreSQL for relational database, MongoDB query language for MongoDB and Cypher query language for Neo4j. They explain that these NoSQL databases may require extra storage space, because of denormalized data, but results in overall improvements in performance, flexibility and scalability. However they do not deal with modeling for the temporal aspects.

Similarly Bugiotti et al. [106] design for a selected NoSQL framework, is based on best practices and guidelines. They provide a methodology, independent of the specific target system, which depends upon the initial activities of software design. They make novel data model for NoSQL databases, named as NoAM (NoSQL Abstract Model). After outlining the commonalities of various NoSQL systems, they specify a system-independent representation of the application data. They treat collections separately as abstract model or table, as in this system. However we have discussed and used the approach to model time series schema design in MongoDB with different variants during the research and the prototype development [128]. Same approach is used for different application domains in [141] and [142].

Vera et al. [143] proposes a general data modeling standard in the form of ERD diagrams for document-oriented databases. Parker et al. [144] compare the performance of NoSQL database, MongoDB, with one of the relational database, SQL Server. Their study shows that for a modest amount of data the performance of MongoDB is equal or better than the relational database. During this study they consider the three main aspects for performace i.e.,insert speed, update speed, and select operation speed. In [12] the authors perform a comparison of 14 different NoSQL Databases based on their data models, query possibilities, partitioning, and replication opportunities. They recommend to use NoSQL databases for fast operations over very large datasets. Following section guides us how we can improve our work in future.

Future work

For the temporal storage of the ANT+ sensor data messages there is a lot of potential for one to extend this work, especially in context to the adjustment of the different arrival rates and by offering different different storage models to each of them. This may require different schema models for each message rate. The other schema techniques also need attention (i.e., doc-per-minute, etc.) along the algorithm development for the data integration and the schema evolution. In future, we plan to have a comparison between the the exact storage measurements with respect to different schema models. For both relational and NoSQL models, such query based data measurement and comparison would reveal valuable results in context to the optimized data storage, retrieval and performance measurements. In the NoSQL domain and while using denormalized schema modeling, the temporal factor is a significant dimension that has not been addressed by the researchers too much. The approach presented in the form of schema model and algorithm is not limited to the ANT+ sensor data but is applicable to other domains

also. Using this fundamental study and the referenced information we can define in general modeling standards for the document-oriented databases.

Summary

The data-driven monitoring applications rapidly and continuously transmit the sensor data, which is meaningful only when is processed and analysed while considering the temporal characteristics. Traditional RDBMS has given in managing such variety of prolific data and the new technologies, such as big data has promised to offer robust management frameworks to handle such continuous data streams. There are different NoSQL management frameworks, such as key-value, column-oriented, graph and document-oriented. A way to manage this is by defining out of the box optimized storage schemas and then to store the data into NoSQL databases while abiding by the format principles. This paper presents the usage of Mongoose middleware, to define document oriented hierarchical schemas for the temporal modeling of the ANT+ sensor data. The NodeJS data server uses these schemas to run a sequence of particular operations to be executed upon a specific event for the data integration and schema evolution. The algorithm automatically integrates the sensor data into a hierarchical structure based upon the temporal properties. This out of the box schema is modeled for NoSQL using the traditional ERD data modeling techniques. There are many possible schema variants, such as (i) document-per-event, (ii) document-per-minute, and (iii) document-per-hour. In this research, we define denormalized schema to have a document for each hour, which contains minutes as sub documents containing sensor data in an array of seconds' sub-documents. The normalization and denormalization of the document hierarchy decides the quality of a schema with respect to number of reads, updates and storage space utilization.

Authors' contributions

NM did the primary literature review, contributed, developed and implemented the idea, designed the experiments,drafted and wrote the manuscript. RC provided the experimental conceptions and contributed during the acquisitionof the ANT+ sensor data. LM played a pivotal role by guiding throughout the research work, especially in contextto the research directions for real-time systems. All authors read and approved the final manuscript.

Authors' information

Nadeem Q. Mehmood is a Ph.D. student at the Department of Computer Science, University of Camerino, Italy. Hiscurrent research work is in Smart Environments, Artificial Intelligence, Emerging Technologies, Real-time Web, BigData and Internet of Things (IoT). He has an M.Sc. in Computer Science from PUCIT, Pakistan and a M.S. degreein Data and Knowledge Engineering from Otto-von-Guericke University, Germany in 2004 and 2008 respectively.

Prof. Rosario Culmone is a researcher at the University of Camerino. He teaches programming languages, databasesand software engineering. He has been interested in web interaction patterns and more recently on programminglanguages based on constraints for Ambient Assisted Living and smart grid.

Prof. Leonardo Mostarda is an associate professor at the Department of Computer Science, University of Camerino,Italy. He is also a visiting professor at the Department of Computer Science, Middlesex University, England. Hisresearch interests include Distributed Systems, Sensor Networks, IT Security, Energy Efficient Wireless EmbeddedSystems, Big Data and Real-time Ubiquitous systems.

Acknowledgements

This work is part of the EUREKA (XXVIII Cycle) project which results from the cooperation between ServiliComputers s.r.l., Italy; the University of Camerino (Department of Computer Science), Italy; and the statedepartment Regione Marche, Italy. I am thankful to the Software Industry; Ministry of Education and Research; andthe University, Italy, for the financial support of this project.

Competing interests

The authors declares that the grant, scholarship and/or funding mentioned in "Acknowledgements" donot lead to any competing interests. Additionally, the authors declares that there is no competing interestsregarding the publication of this manuscript.

Availability of data and materials
The software implementation of the algorithm is uploaded for downloading and customization at GitHub [145].

Consent for publication
I give my permission for the following material to appear in the print, online, and licensed versions of Journal of BigData to grant permission to third parties to reproduce this material. I understand that my name will not bepublished but that complete anonymity cannot be guaranteed. I declare that I have read the manuscript or a generaldescription of what the manuscript contains and reviewed all photographs, illustrations, video, or audio files (ifincluded) in which I am included that will be published.

Ethics approval and consent to participate
Not applicable, since no patient was involved directly or indirectly to collect the data. The author(s) show(n) theconsent to participate and was/were the subject(s) of this study.

Received: 19 July 2016 Accepted: 15 March 2017
Published online: 31 March 2017

References

1. Mauri R. A new generation of data requires next-generation systems. 2015. www.wired.com/insights/2015/01/a-new-generation-of-data-requires-next-generation-systems. Accessed 20 Oct 2016
2. Padhy RP, Patra MR, Satapathy SC. RDBMS to NoSQL: reviewing some next-generation non-relational database's. Int J Adv Eng Sci Technol. 2011;11(1):15–30.
3. Michael M, Moreira JE, Shiloach D, Wisniewski RW. Scale-up x scale-out: a case study using nutch/lucene. In: Parallel and distributed processing symposium, 2007. IPDPS 2007. New York: IEEE International; 2007. p. 1–8.
4. Buneman P. Semistructured data. In: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems. New York City: ACM; 1997. p. 117–21.
5. Abiteboul S. Querying semi-structured data. In: International conference on database theory. Berlin: Springer; 1997. p. 1–18.
6. Blumberg R, Atre S. The problem with unstructured data. DM Rev. 2003;13(42–49):62.
7. Keller AM, Jensen R, Agarwal S. Persistence software: bridging object-oriented programming and relational databases. In: ACM SIGMOD record. vol. 22. New York CIty: ACM. 1993. p. 523–8.
8. Kaur K, Rani R. Modeling and querying data in NoSQL databases. In: 2013 IEEE international conference on big data. New York: IEEE; 2013. p. 1–7.
9. Scherzinger S, Klettke M, Störl U. Managing schema evolution in NoSQL data stores. 2013. arXiv preprint arXiv:1308.0514.
10. Gudivada VN, Rao D, Raghavan VV. NoSQL systems for big data management. In: 2014 IEEE world congress on services (SERVICES). New York: IEEE; 2014. p. 190–7.
11. Zikopoulos P, Eaton C. Understanding big data: analytics for enterprise class hadoop and streaming data. New York: McGraw-Hill Osborne Media; 2011. http://www.bdvc.nl/images/Rapporten/ibm-understanding-big-data.pdf. Accessed 27 Mar 2017.
12. Hecht R, Jablonski S. NoSQL evaluation: a use case oriented survey. 2011.
13. Li Y, Manoharan S. A performance comparison of SQL and NoSQL databases. In: 2013 IEEE pacific rim conference on communications, computers and signal processing (PACRIM). New York: IEEE; 2013. p. 15–9.
14. Pokorný J. New database architectures: steps towards big data processing. In: Palma Dos Reis A, Abraham AP, editors. Proc. of IADIS European conference on data mining (ECDM'13). IADIS Press; 2013. p. 3–10.
15. Bajaber F, Sakr S, Batarfi O, Altalhi A, Elshawi R, Barnawi A. Big data processing systems: state-of-the-art and open challenges. In: 2015 international conference on cloud computing (ICCC). New York: IEEE; 2015. p. 1–8.
16. Grolinger K, Hayes M, Higashino WA, L'Heureux A, Allison DS, Capretz MAM. Challenges for mapreduce in big data. In: 2014 IEEE world congress on services (SERVICES). 2014. p. 182–9. doi: 10.1109/SERVICES.2014.41.
17. Sakr S, Liu A, Fayoumi AG. The family of mapreduce and large-scale data processing systems. ACM Comput Surv. 2013;46(1):11.
18. Chen CP, Zhang C-Y. Data-intensive applications, challenges, techniques and technologies: a survey on big data. Inf Sci. 2014;275:314–47.
19. Hu H, Wen Y, Chua T-S, Li X. Toward scalable systems for big data analytics: a technology tutorial. IEEE Access. 2014;2:652–87.
20. Ribeiro A, Silva A, da Silva AR. Data modeling and data analytics: a survey from a big data perspective. J Softw Eng Appl. 2015;8(12):617.
21. Panigati E, Schreiber FA, Zaniolo C. Data streams and data stream management systems and languages. In: Data management in pervasive systems. Berlin: Springer. 2015. p. 93–111.
22. Gregersen H, Jensen CS. Temporal entity-relationship models—a survey. IEEE Trans Knowl Data Eng. 1999;11(3):464–97.
23. Ozsoyoilu G, Snodgrass RT. Temporal and real-time databases: a survey. IEEE Trans Knowl Data Eng. 1995;7(5):513–32.
24. Cuzzocrea A. Temporal aspects of big data management: state-of-the-art analysis and future research directions. In: 2015 22nd international symposium on temporal representation and reasoning (TIME). New York: IEEE; 2015. p. 180–5.
25. Bonnet P, Gehrke J, Seshadri P. Towards sensor database systems. In: International conference on mobile data management. Berlin: Springer; 2001. p. 3–14.

26. Gilbert AC, Kotidis Y, Muthukrishnan S, Strauss M. Quicksand: quick summary and analysis of network data. Technical report. 2001.
27. Chen J, DeWitt DJ, Tian F, Wang Y. Niagaracq: a scalable continuous query system for internet databases. In: ACM SIGMOD record. vol. 29. New York City: ACM; 2000. p. 379–90.
28. Zhu Y, Shasha D. Statstream: statistical monitoring of thousands of data streams in real time. In: Proceedings of the 28th international conference on very large data bases. Toronto: VLDB Endowment; 2002. p. 358–69.
29. Agrawal R, Somani A, Xu Y. Storage and querying of e-commerce data. VLDB. 2001;1:149–58.
30. Law Y-N, Wang H, Zaniolo C. Query languages and data models for database sequences and data streams. In: Proceedings of the 13th int. conference on very large data bases. vol. 30. VLDB '04. Toronto: VLDB Endowment. 2004. p. 492–503. http://dl.acm.org/citation.cfm?id=1316689.1316733. Accessed 27 Mar 2017.
31. Golab L, Özsu MT. Issues in data stream management. SIGMOD Rec. 2003;32(2):5–14. doi:10.1145/776985.776986.
32. Akulakrishna PK, Lakshmi J, Nandy SK. Efficient storage of big-data for real-time gps applications. In: 2014 IEEE fourth international conference on big data and cloud computing (BdCloud). 2014. p. 1–8. doi: 10.1109/BDCloud.2014.49.
33. Ediger D, McColl R, Poovey J, Campbell D. Scalable infrastructures for data in motion. In: 2014 14th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGrid). 2014. p. 875–82. doi: 10.1109/CCGrid.2014.91.
34. Mehmood NQ, Culmone R. An ANT+ protocol based health care system. In: 2015 IEEE 29th international conference on advanced information networking and applications workshops (WAINA). New York: IEEE; 2015. p. 193–8.
35. Perumal T, Ramli AR, Leong CY, Mansor S, Samsudin K. Interoperability among heterogeneous systems in smart home environment. In: IEEE international conference on signal image technology and internet based systems, 2008. SITIS '08. p. 177–86. doi: 10.1109/SITIS.2008.94.
36. ThisisANT: ThisIsANT: the wireless sensor network solution. http://www.thisisant.com. Accessed 27 Mar 2017.
37. Li T, Liu Y, Tian Y, Shen S, Mao W. A storage solution for massive iot data based on NoSQL. In: 2012 IEEE international conference on green computing and communications (GreenCom). New York: IEEE; 2012. p. 50–7.
38. MongoDB for GIANT ideas. https://www.mongodb.com. Accessed 27 Mar 2017.
39. Bray T. The javascript object notation (json) data interchange format. 2014.
40. Mehmood N, Culmone R. A data acquisition and document oriented storage methodology for ANT+ protocol sensors in real-time web. In: The 30-th IEEE international conference on advanced information networking and applications (AINA-2016). Crans-Montana: Centre de Congrès le Régent; 2016.
41. Babcock B, Babu S, Datar M, Motwani R, Widom J. Models and issues in data stream systems. In: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems. New York City: ACM; 2002. p. 1–16.
42. Hesse G, Lorenz M. Conceptual survey on data stream processing systems. In: 2015 IEEE 21st international conference on parallel and distributed systems (ICPADS). New York: IEEE; 2015. p. 797–802.
43. MongoDB University I. NoSQL Vs relational databases. 2016. https://www.mongodb.com/scale/nosql-vs-relational-databases. Accessed 27 Mar 2017.
44. Ali S. Comparisons of relational databases with big data: a teaching approach. 2016. www.asee.org/documents/zones/zone3/2015/Comparisons-of-Relational-Databases-with-Big-Data-a-Teaching-Approach.pdf. Accessed 27 Mar 2017.
45. Cattell R. Scalable SQL and NoSQL data stores. ACM SIGMOD Rec. 2011;39(4):12–27.
46. Sagiroglu S, Sinanc D. Big data: a review. In: 2013 international conference on collaboration technologies and systems (CTS). New York: IEEE; 2013. p. 42–7.
47. Strozzi C. NoSQL—a relational database management system. Lainattu. 1998;5:2014.
48. Redis is an open source in-memory data store. http://redis.io. Accessed 27 Mar 2017.
49. Project Aerospike. 2016. http://www.aerospike.com. Accessed 27 Mar 2017.
50. Project Voldemort. 2013. http://www.project-voldemort.com/voldemort. Accessed 27 Mar 2017.
51. List of NoSQL databases. http://nosql-database.org. Accessed 27 Mar 2017.
52. Neo4j. http://neo4j.com. Accessed 27 Mar 2017.
53. Beyer M. Gartner says solving "big data" challenge involves more than just managing volumes of data. Gartner. Archived from the original on 10, 2011.
54. Snow, D.: Adding a 4th V to BIG Data-Veracity. http://dsnowondb2.blogspot.cz/2012/07/adding-4th-v-to-big-data-veracity.html. Accessed 06 Mar 2016.
55. Memcached. http://memcached.org. Accessed 27 Mar 2017.
56. MemcacheDB. http://memcachedb.org. Accessed 27 Mar 2017.
57. Apache Cassandra database. http://cassandra.apache.org. Accessed 27 Mar 2017.
58. Jedis-small and sane Redis java client. https://github.com/xetorthio/jedis. Accessed 27 Mar 2017.
59. r3-Map-Reduce engine for Redis Python client. http://heynemann.github.io/r3/. Accessed 27 Mar 2017.
60. Basho Data Platform for Riak. http://basho.com/products. Accessed 27 Mar 2017.
61. Basho Riak KV. http://basho.com/products/riak-kv. Accessed 27 Mar 2017.
62. DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W. Dynamo: amazon's highly available key-value store. In: ACM SIGOPS operating systems review. vol. 41. New York City: ACM; 2007. p. 205–20.
63. Basho Riak TS. http://basho.com/products/riak-ts/. Accessed 27 Mar 2017.
64. Project Voldemort Design. 2013. http://www.project-voldemort.com/voldemort/design.html. Accessed 27 Mar 2017.
65. Amazon DynamoDB. 2012. https://aws.amazon.com/dynamodb. Accessed 27 Mar 2017.
66. Oracle Berkeley DB. http://www.oracle.com/us/products/database/berkeley-db/overview. Accessed 27 Mar 2017.
67. Tokyo Cabinet: a modern implementation of DBM. http://fallabs.com/tokyocabinet. Accessed 27 Mar 2017.
68. Tokyo Tyrant. http://fallabs.com/tokyotyrant. Accessed 27 Mar 2017.
69. Scalaris, a distributed transactional key-value store. https://code.google.com/archive/p/scalaris. Accessed 27 Mar 2017.

70. Abadi DJ, Madden SR, Hachem N. Column-stores vs. row-stores: how different are they really? In: Proceedings of the 2008 ACM SIGMOD international conference on management of data. New York City: ACM; 2008. p. 967–80.
71. Sarkissian A. Wtf is a supercolumn? An intro to the Cassandra data model. 2009. http://arin.me/blog/wtf-is-a-supercolumn-cassandra-data-model. Accessed 3 Aug 2011.
72. Apache HBase. http://hbase.apache.org. Accessed 27 Mar 2017.
73. Apache Accumulo. https://accumulo.apache.org. Accessed 27 Mar 2017.
74. Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE. Bigtable: a distributed storage system for structured data. ACM Trans Comput Syst. 2008;26(2):4.
75. Ghemawat S, Gobioff H, Leung S-T. The google file system. SIGOPS Oper. Syst. Rev. 2003;37(5):29–43. doi:10.1145/1165389.945450.
76. Burrows M. The chubby lock service for loosely-coupled distributed systems. In: Proceedings of the 7th symposium on operating systems design and implementation. Seattle: USENIX Association; 2006. p. 335–50. http://dl.acm.org/citation.cfm?id=1298455.1298487. Accessed 29 Mar 2017.
77. Lakshman A, Malik P. Cassandra: structured storage system on a p2p network. In: Proceedings of the 28th ACM symposium on principles of distributed computing. New York City: ACM; 2009. p. 5.
78. Hypertable. http://www.hypertable.com. Accessed 27 Mar 2017.
79. AllegroGraph—Graph Database. http://allegrograph.com. Accessed 27 Mar 2017.
80. ArangoDB. www.arangodb.com. Accessed 27 Mar 2017.
81. OrientDB. http://orientdb.com. Accessed 27 Mar 2017.
82. Montag D. Understanding neo4j scalability. Neotechnology: White Paper. 2013.
83. Prud'Hommeaux E, Seaborne A, et al. SPARQL query language for rdf. W3C recommendation. vol. 15. 2008.
84. neo4django : an Object Graph Mapper. http://neo4django.readthedocs.io. Accessed 27 Mar 2017.
85. DB-Engines. http://db-engines.com. Accessed 27 Mar 2017.
86. Weinberger C. Native multi-model can compete with pure document and graph databases. https://www.arangodb.com/2015/06/multi-model-benchmark. Accessed 27 Mar 2017.
87. Fowler A. NoSQL For Dummies. New York: Wiley. 2015.
88. Apache CouchDB. http://couchdb.apache.org. Accessed 27 Mar 2017.
89. Apache CouchBase. http://www.couchbase.com. Accessed 27 Mar 2017.
90. Rethink DB. https://rethinkdb.com. Accessed 27 Mar 2017.
91. IBM Cloudant. https://cloudant.com. Accessed 27 Mar 2017.
92. Polymorphism MongoDB. http://mongodb.github.io/mongo-csharp-driver/2.0/reference/bson/mapping/polymorphism. Accessed 27 Mar 2017.
93. Moniruzzaman A, Hossain SA. NoSQL database: New era of databases for big data analytics-classification, characteristics and comparison. 2013. arXiv preprint arXiv:1307.0191.
94. Creating a basic custom schema type. http://mongoosejs.com/docs/customschematypes.html. Accessed 27 Mar 2017.
95. NoSQL---MongoDB vs CouchDB. http://stackoverflow.com/questions/3375494/nosql-mongodb-vs-couchdb. Accessed 27 Mar 2017.
96. Mardan A. Boosting your node.js data with the mongoose orm library. In: Building real-world scalable web apps: practical node.js. Berlin: Springer; 2014. p. 149–72.
97. Morphia - MongoDB object-document mapper in Java. https://github.com/mongodb/morphia. Accessed 27 Mar 2017.
98. Membrey P, Plugge E, Hawkins D. The definitive guide to MongoDB: the NoSQL database for cloud and desktop computing. New York City: Apress; 2011.
99. MongoDB: Mongodb architecture guide 3.2—a mongodb white paper.
100. MongoDB Aggregation. https://docs.mongodb.org/manual/aggregation. Accessed 27 Mar 2017.
101. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. Commun ACM. 2008;51(1):107–13. doi:10.1145/1327452.1327492.
102. Meijer E, Bierman G. A co-relational model of data for large shared data banks. Commun ACM. 2011;54(4):49–58.
103. Elmasri R, Navathe S. Fundamentals of database systems. 6th ed. Boston: Addison-Wesley; 2010.
104. Katsov, I.: NoSQL data modeling techniques. 2012. https://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques. Accessed 27 Mar 2017.
105. Patel, J.: Cassandra data modeling best practices. 2012. www.ebaytechblog.com/2012/07/16/cassandra-data-modeling-best-practices-part-1. Accessed 27 Mar 2017.
106. Bugiotti F, Cabibbo L, Atzeni P, Torlone R. Database design for NoSQL systems. In: 33rd international conference on conceptual modeling. Berlin: SPringer; 2014. p. 223–31.
107. Wei-ping Z, Ming-Xin L, Huan C. Using mongodb to implement textbook management system instead of MySQL. In: IEEE 3rd international conference on communication software and networks (ICCSN). New York: IEEE; 2011. p. 303–5.
108. Kanade A, Gopal A, Kanade S. A study of normalization and embedding in mongodb. In: 2014 IEEE international on advance computing conference (IACC). New York: IEEE; 2014. p. 416–21.
109. Chen PP-S. The entity-relationship model-toward a unified view of data. ACM Trans Database Syst. 1976;1(1):9–36.
110. Codd EF. A relational model of data for large shared data banks. Commun ACM. 1970;13(6):377–87. doi:10.1145/362384.362685.
111. Codd EF. Does your dbms run by the rules? Comput World. 1985;21:11.
112. MongoDB Drivers. https://docs.mongodb.org/ecosystem/drivers. Accessed 27 Mar 2017.
113. MongoDB, Java and Object Relational Mapping. http://www.infoq.com/articles/mongodb-java-orm-bcd. Accessed 27 Mar 2017.
114. mongoose:elegant mongodb object modeling for node.js. http://mongoosejs.com. Accessed 27 Mar 2017.
115. Iridium—a high performance MongoDB ORM for Node.js. https://github.com/SierraSoftworks/Iridium. Accessed 27 Mar 2017.

116. Node ORM2—object relational mapping. https://github.com/dresende/node-orm2. Accessed 27 Mar 2017.
117. What is the killer reason for using Mongoose ORM? http://stackoverflow.com/questions/5747806/what-is-the-killer-reason-for-using-mongoose-orm. Accessed 27 Mar 2017.
118. MJORM (mongo-java-orm)—a MongoDB Java ORM. https://code.google.com/archive/p/mongo-java-orm. Accessed 27 Mar 2017.
119. Java IoT: Article Cover Story: What Is POJO Programming? http://java.sys-con.com/node/180374. Accessed 27 Mar 2017.
120. Mehta VP. Getting started with object-relational mapping. Pro LINQ Object Relational Mapping with C# 2008. 2008:3–15.
121. MongoJack. http://mongojack.org. Accessed 27 Mar 2017.
122. Query in Java as in Mongo shell. http://jongo.org. Accessed 27 Mar 2017.
123. MongoLink: an object document mapper (ODM) for Java and MongoDB. http://mongolink.org. Accessed 27 Mar 2017.
124. POCO Support in .NET Framework. https://msdn.microsoft.com/en-us/library/cc681329.aspx. Accessed 27 Mar 2017.
125. MongoDB ODM for Node.js based on ES6 generators. http://mongorito.com. Accessed 27 Mar 2017.
126. Ming:Database mapping layer for MongoDB on Python. https://sourceforge.net/projects/merciless. Accessed 27 Mar 2017.
127. BackboneORM: a polystore ORM for Node.js and the browser. http://vidigami.github.io/backbone-orm. Accessed 27 Mar 2017.
128. Parikh S, Stirman K. Schema design for time series data in mongodb. vol. 30. 2013. http://blog.mongodb.org. Accessed 27 Mar 2017.
129. MongoDB Cloud Manager. https://www.mongodb.com/cloud. Accessed 27 Mar 2017.
130. MongoDB for time series data (Webinar Series). https://www.mongodb.com/lp/webinar-series/time-series-july-2014. Accessed 27 Mar 2017.
131. MongoDB limits and thresholds. https://docs.mongodb.org/manual/reference/limits/. Accessed 27 Mar 2017.
132. MongoDB GridFS API. https://docs.mongodb.org/manual/core/gridfs/. Accessed 27 Mar 2017.
133. Mehmood NQ, Culmone R, Mostarda L. A flexible and scalable architecture for real-time ANT+ sensor data acquisition and NoSQL storage. Int J Distrib Sens Netw. 2016;2016:13.
134. Dynastream Corporation I. ANT message protocol and usage, version 2.1. www.thisisant.com. Accessed 06 Mar 2016.
135. Dynastream Corporation I. ANT+ device profile: environment, revision 1.0. www.thisisant.com. Accessed 06 Mar 2016.
136. Dynastream Corporation I. ANT+ device profile: stride based speed and distance monitor, revision 1.3. www.thisisant.com. Accessed 06 Mar 2016.
137. Dynastream Corporation I. ANT+ device profile: heart rate monitor, revision 1.13. www.thisisant.com. Accessed 01 May 2015.
138. Dynastream Corporation I. ANT+ common pages, revision 2.4. www.thisisant.com. Accessed 06 Mar 2016.
139. Ant message protocol and usage: application notes,version 2.1. online doc, Dynastream innovations Inc. 2007. www.thisisant.com. Accessed 06 Mar 2016.
140. Arora R, Aggarwal RR. Modeling and querying data in mongodb. Int J Sci Eng Res. 2013;4(7). https://pdfs.semanticscholar.org/bd01/577311001f31d93930586f5ab0ad79bb7564.pdf. Accessed 27 Mar 2017.
141. Janković O. NoSQL dokument baza podataka: prikaz skladištenja podataka sa osvrtom na podatke sa senzora. Infoteh-Jahorina, INFOTEH-JAHORINA. 14:561–6. http://infoteh.rs.ba/rad/2015/RSS-3/RSS-3-1.pdf. Accessed 27 Mar 2017.
142. Papoutsoglou E, Samourkasidis A, Tsai M-Y, Davey M, Ineichen A, Eeftens M, Athanasiadis IN. Towards an air pollution health study data management system—a case study from a smoky swiss railway.
143. Vera H, Wagner Boaventura MH, Guimaraes V, Hondo F. Data modeling for NoSQL document-oriented databases.
144. Parker Z, Poe S, Vrbsky SV. Comparing NoSQL mongodb to an SQL db. In: Proceedings of the 51st ACM southeast conference. New York City: ACM; 2013. p. 5.
145. Nadeem QM. NodeTempANT. 2016. https://github.com/nqaisar/NodeTempANT. Accessed 27 Mar 2017.