Computer tools in education, 2016
№ 2: 3–12
UDC: 004.652
http://ipo.spb.ru/journal

DBMS: A CONTEMPORARY LANDSCAPE IN HISTORICAL PERSPECTIVE

Natalia Grafeeva, Elena Mikhaylova, Boris Novikov

Abstract

The article is a review of the main trends in the modern DBMS. It looks at the main features and capabilities of traditional relational databases and describes the way they are adapted to meet the challenges that data management systems are currently faced with. The article also discusses NoSQL databases and their key characteristics, as well as their capabilities in comparison with relational DBMS.

Keywords: *DBMS, NoSQL, DB-Engines Ranking, Integrity, Durability, Consistence, Availability, Partition tolerance.*

If geometric axioms touched the interests
of people, they would be denied.
Thomas Hobbes

How did database management systems come into existence? Initially, nobody was even going to create a system for managing databases, but there were practical applications that required processing relatively large amounts of data [2]. Having created far more than two or three applications, developers gradually came to notice that, in essentially different applications, they programmed the same functionality related to storing, searching and analyzing data. Besides, in the middle 1960s, there appeared direct access data storage devices of relatively large capacity which were capable of storing data of any structure and allowed direct access to this data. The provision of access to such structures, however, required (as it still does now) quite sophisticated programming, with only highly qualified programmers being up to this task. But the number of such programmers was significantly lower than the number of applications in development. As a consequence, quite a few applications were far from being good-quality programming products. In addition, multiple realizations of similar functionality varied in quality and made the final product considerably more expensive. A good departure from all the above was the idea to separate the functions of an application itself from those involved in data processing. This idea led to another one that envisaged creating centralized systems ensuring access to data, that is, highly efficient and reliable systems that were not focused on the needs of individual applications, but aimed exclusively at data processing. It suffices to create this kind of systems only once. Highly qualified programmers can be involved in developing such a system, and it can further be used multiple times for creating a variety of applications. This idea made it possible to significantly improve the quality, and reduce the cost of, future application development. The basic principles of such systems were first formulated in CODASYL DBTG Report in 1969. This document served as the guidelines for developing database management systems

for many years afterwards. The gist of those principles is provided below. The purposes of data management systems are:

- Reducing development costs and improving application quality by moving data processing function to DBMS.
- Centralizing storage management and data access management.
- Supporting complex logical structures.

The functions of data management systems are:

- Ensuring the mutual independence of data and applications.
- Ensuring data integrity.
- Supporting data consistency.
- Protecting databases from unauthorized access.
- Differentiating access rights.
- Supporting efficient high-level query languages.

The meaning and content of most of these functions have greatly changed in modern database management systems. Some of these functions have been fully or partially transferred to other types of application systems, whereas the importance and presence of other functions depend on the applied architecture of the information system and the type of the DBMS used. In general, however, the requirements above did set the direction of the development of database systems for many years to come. Let us now look at the main functions of a modern DBMS and begin with the functions included in the list above. Ensuring the mutual independence of data and applications was initially considered as the most important element of data management systems. The mutual independence of data and applications implies the following:

- The same data can be used for a variety of applications.
- New data requirements (for example, adding new fields, tables, business logic of data behavior, etc.) does not affect the existing applications.
- A valid asynchronous implementation of new application versions is possible (this is of particular importance for “thick” clients).

For DBMS to meet the requirements listed above, the following steps were taken: special programming languages were invented, which described the structure of data and the business logic of data behavior; there were data dictionaries introduced, which served as a tool for presenting such information. Data dictionaries and data definition languages are, in this or that manner, included in nearly all modern traditional DBMS. Most systems strictly adhere to the principles of data independence; however, due attention should be paid to the recent appearance of DBMS that fail to follow what seem to be unquestionably useful principles. Violating the first principle of data independence means that there appear data warehouses that are limited to a particular kind of data and the functionality of a single application. This would seem very irrational, if it were not for the fact that such applications are of exclusive kind and in demand by millions of users. The violation of the second and third principles of data independence has a negative effect on the development, maintenance and support of applications that interact with such systems.

In addition to describing data structure, a data management system has to ensure that stored data fully satisfies a number of conditions. For example, a grade that a student receives for their performance at an examination has be in the range from 1 to 5; the subject in which such a grade is given must be on the list of courses, etc. Such conditions are called *integrity constraints*, and they are described with the help of special language constructions. It is a DBMS that is responsible for making sure that all these conditions are satisfied in full. Any operation is rejected

if it attempts to violate these restrictions. Typical integrity constraints existing in most of today's major DBMS include the following: referential integrity, null values prohibition, duplicate values prohibition, and value range control. However, there are DBMS where integrity rules are either non-existent or only partially present. It should be noted that, if the nature of data is such that it requires certain restrictions to be taken into consideration, all of them will still have to be taken into consideration at this or that level, and, if it is technically impossible to take them into account at the database level, developers will inevitably have to do so at the level of the application in development.

Consistency support is related to the state of data. Some of database states are referred to as being consistent. The task of a DBMS is to provide mechanisms that allow to transfer the database from one consistent state to another. A set of operations that transform a database from one consistent state to another is called a transaction. The supporting by a DBMS of data consistency implies that, upon a successful or unsuccessful shutdown of an application, the database will, in either case, be in a consistent state, that is, all of the application transactions will either have been completely executed or have left no traces in the database. In the latter case, it is the DBMS that will ensure the rollback of uncompleted transactions. This requirement is called the atomicity of a transaction. After a transaction is committed, relevant changes in the database become permanent, this being called the durability of a transaction. Consistency support becomes much more complicated if the DBMS allows for parallel (or concurrent) execution of transactions. In traditional DBMS, this complication is handled with the help of a variety of sophisticated algorithms for executing transactions in such a manner that an illusion of isolation is created for all users that are working with the database simultaneously. DBMS that support all of these features are referred to as having ACID *(Atomicity, Consistency, Isolation, Durability)* properties. In reality, however, significantly different levels of isolation may be masked under such properties. According to the SQL-92 standard, there are 4 levels of isolation:

- *Read uncommitted;*
- *Read committed;*
- *Repeatable read;*
- *Serializable.*

Read uncommitted provides the weakest isolation level that allows a user to read the data of uncommitted transactions — the so-called "dirty reading". Each successive isolation level includes all the previous ones, with *Serializable* providing the strongest isolation level. Most traditional DBMS use *Read commited* as the default level. This isolation level guarantees protection against dirty reading, but, upon repeated reading of the same dataset, the results may be different, if, in the interval between the readings, the data was changed by a successfully committed transaction. The *Repeatable read* level does not allow to modify the data that has been read by the active transaction, but it does not prevent other transactions from adding new data. The strongest isolation level, *Serializable*, is geared towards creating an absolute illusion of autonomous work, as if no other transactions were taking place. It is only this isolation level that provides protection against so-called "phantom reading". In fact, some industrial databases use the Snapshot Isolation level instead of the *Serializable* one. The Snapshot level is slightly weaker than *Serializable*, but stronger than *Repeatable read*. A transaction operating at this level "sees" only those data changes that were committed before its launch, in other words, it behaves as if, upon launching, it received a snapshot of the database and works with it. It is clear that supporting the *Serializable* isolation level is the most difficult, which is why most systems use the lower levels by default. Systems with the *Serializable* isolation level and concurrent execution of billions of transactions also exist, though.

The algorithms supporting the above-mentioned isolation levels do an excellent job of creating an illusion of user isolation. It is hard to imagine that, in the recent past, it was up to developers to create such an illusion at an application level. In recent years, however, there have appeared DBMS that oppose themselves to traditional DBMS and declare the non-use of the *Serilizable* isolation level as one of their basic principles [13]. Such a declaration looks strange, to say the least, given the fact that few applications indeed use this isolation level.

Almost any DBMS needs to protect data from unauthorized access and differentiate levels of access. To this end, there have been developed a number of tricks aimed at hiding real data from users (for example, views) and differentiating access rights (roles, privileges, mandate level access, etc.). The latest versions of some commercial DBMS even provide for a variety of methods of data display, including those involving deliberate distortion of displayed data (as done in ORACLE 12C). Another forward-looking approach to data protection is to isolate the administrators of database servers from the data itself. In most of today's DBMS, a database server administrator can read and modify any data in the database. New means of data protection limit what administrators can do: They can still perform all operations involved in managing a database, but they are unable to read and modify the data (as in ORACLE 12C). It should be noted, however, that, despite an abundance of techniques for protecting data within DBMS, a two-tier data protection is implemented in a lot of modern applications. The two-tier protection involves first verifying user rights at the level of an application and, then, verifying unified user rights at the DBMS level. Another recent tendency is for large companies to create a three-tier data protection systems that start by verifying users at the operating system level. In most cases, moving user management functions from DBMS and applications to a centralized system indeed simplifies user management across the organization and helps automate management processes.

The presence of a high-level and effective query language is probably the most important feature of DBMS. To this end, high-level declarative languages for manipulating data were implemented in the framework of the various management systems, one of which, SQL, deserves special attention. The elegance and independence of this language from DBMS specifics, as well being supported by leading database manufacturers, made SQL a basic standard for data processing. As important as the elegance and independence of the language are, they do not ensure its efficiency, that is, an ability to promptly execute queries without putting a strain on resources. That is exactly why query optimizers have appeared in almost all DBMS. The main function of a query optimizer is to make a number of possible query execution plans and select the optimal one. As a rule, the best plan is the one which takes the least time to execute, execution time being the selection criterion in this case. In some cases, however, the choice of criteria may depend on application requirements. For example, you want to minimize the time of receiving the first query rows or the execution time of the full query. Optimizers choose plans on the basis of explicitly or implicitly defined cost function. There are two modes optimization: optimizing on the basis of the rules, and optimizing based on the statistical characteristics of the actually stored data. At present, the optimization algorithms are well developed and implemented in industrial databases, and it is only on rare occasions that they require manual tuning (that is, writing hints to the optimizer) or rewriting queries completely in order to improve their performance.

It seemed to many of us that there would only be further developments in this direction, that is, the emergence of new declarative constructions, new methods of optimization, etc. In the past decade, however, there have appeared so-called NoSQL systems that abandon many of the principles laid in the basis of the traditional DBMS, with some of the new systems going as far as giving up the declarative query languages [9]. Non-use of high-level query languages leads to a number of obvious consequences. In particular, application developers are forced to formulate low-level queries that require significantly more time at the design stage, which ultimately

causes the development of an application to take a lot more time and cost a lot more. A developer who uses a low-level query language is like a racing car driver who has to use a scooter instead of a high-performance sports car. In fact, there are two paths for application developers to follow: using an expensive DBMS with a high-level query language in order to quickly create an application, or using an inexpensive, or even free, DBMS with a low-level query language, which will inevitably make it much more time-consuming and expensive to create an application. There are advantages to each of the two paths. The first one guarantees high quality of application queries at low development costs; the second one guarantees long-term employment for a large number of developers. As strange as it may seem, the first advantage does not prove convincing to some companies. In the IT industry today, there are a number of companies whose services are used by millions of users worldwide. Such companies can afford an enormous staff of highly skilled developers. The applications created by such companies often utilize data management systems with low-level query languages. The success and popularity of these applications create the illusion of the simplicity of "churning out" applications. As a matter of fact, these applications are the result of joint effort by teams of highly qualified developers, testers, analysts, administrators, etc. Developers at small companies should be aware that the technology for creating such applications is of exclusive nature and cannot be scaled.

Until quite recently, it was possible to classify DBMS types by supported data models: relational, hierarchical, object-oriented, XML, etc. Today, this classification makes sense only for some NoSQL systems that, as rule, have narrow specialization. At the same time, most industrial DBMS, having originally been positioned as relational ones (e.g., ORACLE and DB2), are declared to support, and do support, a variety of data models, including unstructured data. Whatever the features of the model of a particular DBMS can be, the relevant database consists of objects. This means that object identification is one of the most important functions of a modern DBMS. The main classes of identification methods are the following:

- *Identification by properties*. This type of identification is based on the values of some attributes of objects to be identified. What is good about this method is that it is life-like and resembles object identification by signs in the natural environment, for example, identification of a person by fingerprints.
- *Positional identification*. This identification method is based on the information about the position of an object in space or in relation to other objects. This class may include geographical coordinates, all kinds of addresses, and relative instructions (for example, "immediately after the third bridge").
- *Surrogate identification*. It is based on the attributing to an object of a non-existent identifier that is in no way associated with the properties of this object. This identifier is generated upon the first appearance of the object in the system and never changes.

Another aspect of data processing is how data is reviewed or searched. In modern systems, there are two types of data reviewing:

- Navigation by link.
- Associative data search by values.

The latter method is life-like and typical of traditional systems where it is possible to search for an object (or objects) when there is information about some of its (or their) characteristics, for example, finding all ginger cats with green eyes. How many units of data can be processed in a single operation is yet another facet of data processing. There are two approaches to data processing:

- Processing of individual objects.
- Bulk processing.

The SQL language, for example, was designed for bulk data processing. A large number of rows can be processed in a single operation. Moreover, one of distinctive features of all popular SQL implementations within traditional DBMS is that bulk processing operations are significantly more efficient than single object processing: It is more efficient to process 10,000 objects in a single operation than to process individual objects 10,000 times. Notably, however, there are systems where query languages are focused on processing individual objects, and, moreover, such systems are being actively developed and widely discussed.

In the past decade, there have arisen new data management systems that are grouped under a rather ambitious slogan “the NoSQL movement”. These systems position themselves as alternative systems that can store and process enormous quantities of data [11]. There is a wide variety of NoSQL systems in use, and it is difficult to understand what these systems really are. And it is even more difficult to give recommendations on their use for particular applications. Let us try to describe this NoSQL movement with the help of examples of real systems. Most NoSQL systems were created for easy scaling and use on clusters of inexpensive computers in a cloud environment. As is well known, transaction support and strict data consistency in distributed systems require a considerable number of synchronous interactions. This does not only reduce the response time of the system, but also lowers its reliability. In addition, there are a number of applications where there is no objective need to support transactions. A good example of such applications are analytical ones, in which there are no regular updates, and no information arriving in the past hours, minutes or seconds is of significance. It may seem that the existence of such kind of tasks is a sufficient argument in favor of creating systems without transaction support. In practice, however, it all turned out differently.

Amid the discussion of performance issues and fault tolerance, there appeared an empirical statement by Eric Brewer that a distributed system cannot provide consistency, availability and partition tolerance simultaneously. Published in the article [4], it later became known as "CAP theorem" (*Consistency, Availability*, *Partition tolerance*). Nothing like a mathematical proof can be found either in the original article, or in any other; therefore, it is not a theorem unless there is a proof. In fact, the term "CAP theorem" has become nothing but a catchy term for an empirical statement. Some developers of new data management systems may not have even read Eric Brewer’s original article, but they have used the "CAP theorem" as a formal rationale for creating a variety of data storage systems based on tradeoffs between these three properties. The theorem is cited on every hand today, and an article on NoSQL storage where it is not cited will be hard to come by . Moreover, there is even a classification of new data management systems made on the basis of this theorem [2]:

- CA — meets the requirements of Consistency and Availability,
- CP — meets the requirements of Consistency and Partition tolerance,
- AP — meets the requirements of Availability and Partition tolerance.

It is more than evident that the ACID properties will not be supported at all in the above cases, whereas such properties will be supported, if to a varying degree, by applications based on traditional DBMS. Nevertheless, the problems that existed in traditional DBMS and were related to handling large amounts of data, the appearance of new methods of managing distributed systems, as well as the ripple of excitement caused by the “CAP theorem”, led to the emergence of a new class of systems that would later be called NoSQL ones.

The authorship of the NoSQL term is attributed to Johan Oskarsson who used it at the conference on non-relational databases in 2009 [11]. The acronym NoSQL stands for “Not Only SQL”. It is an umbrella term for a whole range of data storage systems, many of which are not based on the relational data model. Such systems tend to be narrowly focused on highly specialized data

models, for example, graphs. As diverse as NoSQL systems are, the following set of characteristics is often associated with them [7]:

- Supporting simple and flexible non-relational data model intended for a wide range of tasks. Modern NoSQL data models are usually divided into four categories [7], and namely: storage of key-value type, document storage, column storage, and graph-oriented storage.
- Horizontal scaling. Some NoSQL storage systems provide scalability for data storage, while others focus on scaling read/write data operations.
- Ensuring high availability of data. Many NoSQL data storage systems are intended for distributed data usage scenarios. High availability of data is ensured at the expense of data consistency, which has led to the appearance of solutions of AP type (Availability, Partition tolerance).
- Not supporting ACID transactions. Unlike traditional DBMS, NoSQL systems do not typically support ACID transactions. Such systems are sometimes called BASE systems (*Basically Available*, *Soft state*, *Eventually consistent*) [8]. In this acronym, Basically Available means that data is always provided upon a user's query, even if part of this data may not be available at this moment. Soft state means that data may not be in a consistent state for some time, and Eventually consistent means that the data in a storage system will inevitably and eventually come into a consistent state. However, there are some NoSQL data storage systems, such as CouchDB [3] , that provide support for ACID transactions.
- Utilizing the MapReduce technology [8]. This technology breaks up data processing into two steps — Map (1) and Reduce (2). The Map step is when preliminary data processing takes place; in the Reduce step, the results obtained in the previous step are combined into a single result. The main advantage of the MapReduce technology is high fault tolerance, that is, the ability to continue working in case of equipment or power failures. This technology is especially useful in those applications where absolute accuracy is not needed.

Data models and data processing methods in NoSQL systems are extremely diverse. As was noted earlier, their characteristic feature is the focus on a narrow class of tasks. What follows is a description of the main categories of NoSQL storage systems [7] and their most popular representatives. According to the ratings of the key-value systems issued by DB-Engines Ranking in February 2016 [5], the most popular systems are the following: Redis, Memcached, Amazon DynamoDB (multi-model system), and Riak. These systems are declared to guarantee high performance, ease of scalability, and effective search of objects by a unique key, with such requested objects being either a sequence of bytes or having a more complex structure. The claims about high performance which are made by those who develop such systems cause some doubt as to their objectivity, for an objective comparison of the key-value systems and traditional DBMS ones can only be based on comparable operations. Such operations as those performed in a traditional DBMS with the help of a single high-level request will probably take dozens, or even hundreds, of low-level operations in key-value systems. Making a comparison on the basis of small key-value operations does not seems fair because it is bulk processing that is the strength of high-level queries.The authors of this article did not find either any results of such a comparison, or ideas of how to do it. Therefore, there are no objective grounds for stating that key-value storage systems are better than traditional DBMS ones in terms of performance. In addition, the complete absence of relationships between objects seems to be a serious disadvantage of key-value systems, which means that a key-value storage management system is unable to control the integrity of relationships, and the corresponding functionality is all passed to the application. However, this is not a serious problem for a certain class of applications, and these function quite efficiently (for example, systems for storing video files, images, etc.).

As for document storage systems, DB-Engines Ranking ranks the following ones the highest: MongoDB, CouchDB, CouchBase and the above-mentioned multi-model system Amazon DynamoDB. Within the framework of the documentary model, these systems store objects (documents) in JSON format (Java Script Object Notation) or BSON (Binary JSON). The JSON and BSON formats allow the use of attributes of simple types, arrays and nested objects. These systems support indexes in fields of documents and allow to build complex queries. ACID transactions are not supported either. However, update operation on the level of one document are usually atomic. Such data stores are effectively used in content management systems, publishing, documentary search, etc.

A special place among NoSQL systems is occupied by column data stores. According to the ratings of DB-Engines Ranking, the leading column data stores are the following: Cassandra, HBase, Accumolo, and Hypertable. What unites these data storage systems is that data is presented in them as tables, and data storage and fragmentation is not effected by rows, as in traditional DBMS, but in columns. In addition, many systems of this class are characterized by the use of SQL-like high-level languages. As is known, the main principles of relational DBMS are: the presentation of data in tables and the use of high-level languages. How exactly data is stored does not matter. Therefore, technically speaking, a column data store is nothing but an ordinary relational DBMS that stores and fragments data in a different manner than existing relational DBMS do (ORACLE, DB2, Postgress, Mycrosoft SQL Server, MySQL, etc.). It seems strange, to say the least, that this group of systems is featured as part of the NoSQL movement.

Graph-oriented systems are specially designed to store graph nodes and connections between them. As a rule, such systems allow to assign sets of arbitrary attribute for nodes and links and, then, to select nodes and links by these attributes. In addition, such systems support algorithms of graph traversal and route construction. DB-Engines Ranking gives the highest rankings to the following graph-oriented systems: Neo4G, OrientDB, Titan, ArangoDB. OrientDB and ArangoDB are marketed as multi-model one. Graph-oriented systems are best used for tasks related to analyzing social networks, choice of routes, etc.

NoSQL systems tend to allow to specify various cluster configurations and thus to achieve required application properties (for example, to allow or prohibit reading from the second replica, set the number of possible replicas, etc.). In addition, if the system supports partitioning data (sharding), it becomes extremely important to choose suitable keys for distributing data between network nodes with the view to balancing the workload.

In summary, the NoSQL movement does not only include a wide variety of models (including the relational one) and processing methods, but also a lot of finely tunable configuration options that considerably affect the properties of a systems. What about traditional relational DBMS? Are they as obsolete as some authors say? Of course, not! Relational databases have existed for over 30 years now. In this time, several revolutions have been sparked in the IT industry, and each of them intended to come up with new ideas about data storage and processing. A newly-developed technology for handling databases had, now and then, been proclaimed a technological breakthrough that will help to do away with relational databases. The latter was exactly what was expected of object-related databases in the 1990s. The same expectations were about the future of XML storage systems in the first decade of the 21st century. Time has shown, however, that none of these revolutions achieved as much as destroying relational databases as a class. Yet again, there is now a new revolutionary movement rising on the basis of the new technologies united under the flag of NoSQL.

Generally speaking, what happens when such technologies come into existence? Database developers learn them, assess their viability, and then introduce the best technological solutions to traditional DBMS. Take a closer look at the recent versions of the latter! It is for quite a while

that they have supported a variety of data models (relational, object-oriented, XML, JSON, key-value, etc.), have had improved data processing methods and used parallelization techniques, etc. The most recent versions are intended for working in the cloud environment and possess all the characteristics required for it. Strictly speaking, traditional DBMS are no longer just relational DBMS, as they once were. If they are still called "relational" ones, it is nothing but a matter of tradition. To what extent traditional DBMS are popular is convincingly shown in the unified ranking of all DBMS issued by DB-Engines Ranking in February of 2016 (Figure 1).

295 systems in ranking, February 2016

Rank			DBMS	Database Model	Score		
Feb 2016	Jan 2016	Feb 2015			Feb 2016	Jan 2016	Feb 2015
1.	1.	1.	Oracle	Relational DBMS	1476.14	-19.94	+36.42
2.	2.	2.	MySQL	Relational DBMS	1321.13	+21.87	+48.67
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1150.23	+6.16	-27.26
4.	4.	4.	MongoDB	Document store	305.60	-0.43	+38.36
5.	5.	5.	PostgreSQL	Relational DBMS	288.66	+6.26	+26.32
6.	6.	6.	DB2	Relational DBMS	194.48	-1.89	-7.94
7.	7.	7.	Microsoft Access	Relational DBMS	133.08	-0.96	-7.47
8.	8.	8.	Cassandra	Wide column store	131.76	+0.81	+24.68
9.	9.	9.	SQLite	Relational DBMS	106.78	+3.04	+7.22
10.	10.	10.	Redis	Key-value store	102.07	+0.92	+2.86

Figure 1. The most popular DBMS as ranked by DB-Engines Ranking

Although NoSQL systems are becoming increasingly popular, it is believed by the authors that traditional databases are not losing ground and are still in demand on the software market. In the near future, both kinds of systems will be used simultaneously, but they will be intended for different practical applications. The choice of a storage system to be implemented will depend, as it does now, on the nature of data itself, the estimated volume of data to be stored, as well as the manner in which it will be used.

References

1. Kuznetshov, S. & Poskonin, A. 2013, "Is cooperation between SQL and NoSQL possible?" ["Vozmozhno li sotrudnichestvo SQL i NoSQL? "], *Open systems*, no. 9, pp. 38–41.
2. Seleznyov, K. 2012, "From SQL to NoSQL and back" ["Ot SQL k NoSQL i obratno"], *Open systems*, no. 2, pp. 25–29.
3. "ApacheCouchDB" [Online], available at: http://couchdb.apache.org/ [Accessed 11 Jan. 2016].
4. Brewer, E. 2000, "Towards Robust Distributed Systems", in: *ACM Symposium on the Principles of Distributed Computing*, Portland, Oregon.
5. "DB-Engines Ranking" [Online], available at: http://db-engines.com/en/ranking/ [Accessed 26 Feb. 2016].
6. DeCandia, G, Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. & Vogels, W. 2007, "Dynamo: Amazon's highly available Key -value store", *ACM SIGOPS Operating Syst. Rev.*, no. 41, p. 205.
7. Hecht, R. & Jablonski, S. 2011, "NoSQL evaluation: A use case oriented survey", *Proc 2011 Int. Conf. Cloud Serv. Computing*, pp. 336–341.
8. Dean, J. & Ghemawat, S. 2004, "MapReduce: Simplified Data Processing on Large Clusters", in: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA.

9. Grolinger, K, Higashino, W., Tiwari, A. & Capretz, M. 2013, “Data management in cloud environments: NoSQL and NewSQL data stores”, *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 2(22).
10. Murty, J. 2008, *Programming Amazon Web Services: S3, EC2, SQS, FPS, and SimpleDB.* O’Reilly Media, Inc.
11. “NOSQL meetup. San Francisco: Eventbrite” [Online], available at: http://nosql.eventbrite.com/ [Accessed 5 Jan. 2016].
12. Pritchett, D. 2008, “BASE: An ACID Alternative”, *Queue*, no. 6, p. 48.
13. Vogels, W. 2009, “Eventually Consistent”, *Communication of the ACM*, no. 52(1), pp. 40–44.

БАЗЫ ДАННЫХ: СОВРЕМЕННЫЙ ПЕЙЗАЖ В ИСТОРИЧЕСКОЙ ПЕРСПЕКТИВЕ

Графеева Наталья Генриховна, Михайлова Елена Георгиевна, Новиков Борис Асенович

Аннотация

Проблема обработки и хранения BigData, основную часть которых составляет неструктурированная информация, привела к появлению NoSQL баз данных, которые стремительно завоевали популярность. Одно время даже высказывалось мнение, что традиционные реляционные СУБД обречены. Действительно ли это так? Решают ли новомодные системы те задачи, которые стоят в настоящее время перед системами хранения данных?

Ключевые слова: *СУБД, NoSQL, рейтинг DB-Engines, целостность данных, продолжительность транзакций, согласованность данных, доступность данных, устойчивость к разделению.*

Natalia Grafeeva,
Associate Professor, Sub-Department of Analytical Information Systems, St Petersburg University,
N.Grafeeva@spbu.ru

Elena Mikhaylova,
Associate Professor, Sub-Department of Analytical Information Systems, St Petersburg University,
E.Mikhaylova@spbu.ru

Boris Novikov,
Professor, Sub-Department of Analytical Information Systems, St Petersburg University,
borisnov@acm.org