What Can NoSQL Serve an Enterprise

Helena Palovska

Faculty of Informatics and Statistics
University of Economics Prague

palovska@vse.cz

DOI: 10.20470/jsi.v6i3.240

Abstract: Strengths and limitations of relational database model and SQL databases are rethought. Universality of SQL knowledge, “universal database” engine and symmetric querying are found to be the main strengths of SQL databases. Need of centralized database and thus limited data volume manageable, relative stability of database schema and limited set of query types processed efficiently are seen as the main limitations. As well, limitation of ACID (atomicity, consistency, isolation, durability) architecture is expounded; this found to be the poor performance in case of distributed data processing. Formed types of NoSQL databases are listed, with their features and scope of strength outlined. Finally, recommendations in what cases different types of NoSQL stores to apply are summarized.

Key words: data administration, data management, SQL, NoSQL, key-value model, document store, column-family, graph database

1. Introduction

Enterprise information systems are usually built upon an SQL database. While the reasons for this seem to be widely understood, a critical analysis of this phenomena finds the author missing. During last decades, NoSQL databases arose, and a question what substantial they brought can be asked. Not all movements and trends in database world (cf. eg. Pokorny, 2010) are covered in this paper, only NoSQL databases are focused on, because they are considered substitutes or successors of SQL databases.

This author will first focus on the advantages and limitations of SQL databases, to find the issues for the NoSQL solutions. The author tries to avoid the usual argument; instead she re-examines the history and exceptionality of the relational model and also reviews some common characteristics of databases that are reevaluated in NoSQL solutions.

Subsequently, an overview of properties and advantages of NoSQL solutions is given, and recommendations are made when to use these solutions.

2. Relational History

Circumstances and reasons for the relational model, period when the SQL database prevailed, and important accompanying database features are outlined.

2.1 Birth of the Relational Idea

In 1960ies, data management was extremely cumbersome because of the approach that data would be organized and manipulated in accordance with access paths. This approach made processing of the main tasks effective, but other tasks could be performed very poorly. Computer programmers suffered from fighting structure designed for one task to serve other task given. Predominantly used technology seemed not to allow other way.

Edgar Frank Codd, an employee of IBM, thought of a logical layer above the physical, in which the programmers' work will reside. This logical layer should certify symmetry in exploitation of data, which means that all information requirements would be of the same difficulty to express (Codd, 1970, p.382). As a form of such a layer, he thought of the relational model (Codd, 1970).

At that time, hardware technology existed mature enough to implement this idea – direct-access persistent storage – but was not generally affordable. Data organization and manipulation logic to support the relational model had to be invented yet – many researchers tried hard to invent physical
data structures and data manipulation methods which could serve to implement it. This effort was worth enough because of attractiveness of the relational idea and its symmetry in exploitation of data; the logic of the relational model seemed potentially comfortable for programming.

2.2 Reign of the Relational Model

During 1970ies, relational databases came to existence. In 1980ies, relational databases became industrially applied. In 1990ies, relational model was predominant in databases – SQL databases, although (Codd, 1990) criticized SQL databases not to be truly relational.

By this author's meaning, the reason for this was the universality of SQL databases, as it is explained in part 3 of this paper.

2.3 ACID Processing

Although not genuinely a part of the relational model, during the period when SQL databases grew, a frequent request to perform specific series of data operations, called database transactions, “all or no”, to meet business logic resulted in a technology (Grey, 1981) called ACID (Haerder, Reuter, 1983). This technology proved to be very efficient when all operations of one transaction are to be performed by a single database engine. Grey developed a technology to cope with ACID requirements in distributed database architecture (Grey, Reuter, 1993), but application of distributed ACID transactions showed less efficient then was expected – this is more explained in section 4.1.

3. The Universality of SQL Databases

For a period, SQL databases became “universal databases”. This author explains this universality by pointing to the three following phenomena. First, the relational model and SQL became universal knowledge for database specialists; it became universal database conception and language. Second, an SQL engine became universal database management system (DBMS) for all enterprise applications. Third, when data model reflected the area of interest, all information requests upon this area of interest could be formulated in SQL.

These phenomena are more explained in the following.

3.1 Universal Database Knowledge

Database designers and programmers could reuse their knowledge gained. Database design became semi-routine under condition that information analysis produced some ER model and the target database model was the relational model. As to SQL, differences evolved between different SQL dialects, and language reference has to be consulted when the SQL dialect (Wikibooks, 2015) was to be changed, but the core of the language is universal. Successive versions of SQL language standard supported for this.

However, for database administrators this universality was not available. Database administration has to specialize in particular DBMS; administration knowledge is not generally portable to a system of another producer.

3.2 Universal Database Engine

All enterprise data could be managed by single database engine, because all enterprise applications could store their data in the relational model. Until relational databases showed to be cumbersome or less efficient for some enterprise applications, a relational database was an appropriate choice. Even for data warehousing an SQL database is frequently applied as an implementing layer because of the cost of properly multidimensional solutions.

During the decades, efficiency of individual SQL engines increased. By opening of some algorithms and some code to the public this efficiency has been shared.

A problem arose when there was a need for data distribution. In that case high efficiency of a single SQL database engine confronted low speed and reliability of network communication. The overall architecture increased the probability of fault because more nodes had to coordinate in a hard-state system, as well.
3.3 Universal Querying

Upon an SQL database storing facts from some area of interest, all logically correct queries can be expressed in SQL. Moreover, the most common queries are formulated in a quite simple manner. Usual views of the area of interest are of similar difficulty/simplicity.

Declarative nature of SQL allows to formulate queries at the logical level of facts only, out of the realm of access paths, which are thus encapsulated at a lower level.

Declarative query language in turn becomes an obstacle when a purely logical construction is found to be tricky. Also, when complex objects are required, the formulation of queries becomes cumbersome.

4. Limitations of SQL Databases

Over the decades, SQL database technology has evolved into very effective, upon some specific conditions. These are: that data are stored in one centralized database that database schema is stable, and that frequent information requirements are simple. These conditions are described in more detail in the following.

4.1 All Data in a Single Database Engine

When messaging and data transfer between different SQL database engines are incorporated in the data processing architecture, distributed queries as well as distributed transactions rapidly slow down the overall performance. This is essentially because of the low speed of communication between node engines and the probability of its failure. Moreover, probability of a failure of one node multiplies the system’s failure probability. It follows that we obtain best performance in the case of all the data being processed by a single database engine. This is not always the case, however.

Volume of the data manageable by single SQL database engine is limited. Nowadays demands on data volume needed by one application can exceed this limit.

In other cases it may be requisite to distribute data due to geographical plurality of their origin.

4.2 Stable Database Schema

Predominantly inbuilt efficient SQL database technologies depend upon fixed database schema. A schema change in SQL databases leads to a major database reconstruction accompanied by operation cut-off. In addition to that, due to an application design style expecting given schema, a schema change can cause additional necessity of redesign of the applications.

4.3 Simple Frequent Queries

Storage structures, data access methods, processing algorithms and query optimizers were developed with limited set of query types in mind. This limited set of types grew out of experience in previous decades of enterprise information systems. New tasks and applications challenge this settled set of solutions – examples being OLAP, complex objects, XML data, geographical data, temporal data, network analysis...

5. What brought NoSQL

Several database models have been created since the relational, each aiming to overcome some specific limitations. Object oriented databases, OLAP databases, XML databases are examples created in the 90ies. During the last one-and-half decade models emerged which collectively won the name NoSQL. Categories of them have been formed by data structures types that corresponding database can store: key-value stores, document stores, column-family stores and graph databases (Sadalage & Fowler, 2012).

In different NoSQL-database products, basic constructs described below can be enriched or amended, or more models can be put together.
5.1 Aggregate-oriented data stores

The first three of NoSQL database types – key-value stores, document stores, column-family stores – allow for effective distribution of data. Data volume manageable by those databases can be effectively horizontally scaled while preserving high data availability. The data structure analysis strives to recognize data chunks, called aggregates, requested by the applications at a stroke, and to classify such aggregates. Load can be distributed among multiple database network nodes by fractionation, called sharding, of the classes of aggregates. Distributed data processing spanning multiple shards can apply map-reduce algorithms (Lämmel, 2008), (Dean & Ghemawat, 2010).

ACID compliance is not primarily aimed, instead versioning is applied to avoid update conflicts. For achieving high availability, replication in clusters of database nodes is deployed, either of master-slave, possibly with moving master, or peer-to-peer architecture. Instead of ACID processing, rather eventual consistency is sought which means that though the user can read obsolete data, after some time the update reaches him, after rereading.

The data chunks – the aggregates – form the basic constructs of these models. The difference between the three categories is in applicable inner structure of the aggregates. All three are built upon the key-value model:

The **key-value model** is the simplest one of them. The only data structure used in it is a key-value pair – with the meaning that the key is mapping to the value. This model allows for the greatest freedom in data organization design, but the least logical level data service. All data access must be designed via organization of the keys system. The values themselves are blobs, no meta-data level for the values is supported by the database. The aggregates – data chunks – are the key-value pairs. No joins are supported; data chains designed via storing “foreign” key in the value part of the key-value pair would be a matter of application programming (load a value, extract a “foreign” key, load indicated value).

**Document-store model** uses hierarchical semi-structured “documents” consisting of “fields” – name-value pairs. Documents are the data chunks – the aggregates; they are primarily accessed by their keys. The database can nevertheless further hierarchically examine the fields of the documents (CouchDB, MongoDB, 2015). For fields, indexes can be defined and used in document search. SQL-like (no JOINS, no GROUP BY) query language can be provided.

The document-store model does not impose schema definition on documents, that means each document can consist of its own fields. It is the responsibility of a programmer to create the fields meaningfully and use them consistently within a project. On the other hand, application data schema flexibility is enabled this way.

**Column-family model** decomposes records into aggregates called column families (Chang & col., 2004). Column families may be of two types: standard- and super- (Lakshman & Malik, 2009). A standard column family is a set of name-value-timestamp triples – called columns. A super-column family is a set of super-columns – named sets of name-value-timestamp triples. Super-columns so provide for further data nesting, but are not independent units. No matter if standard- or super-, column families form the units of data manipulation. Primary access method is a row key; in some implementations, indexes for non-key columns can be defined. Whether the schema should be defined up-front to populating the store with data, or not, depends upon the store product. Terminology of the data model differs between products, too.

Neither standard nor super-column families impose strict structure, their columns or super-columns are optional – in fact, sparse records are expected. In a case, (super-)columns can be added dynamically to a column family (standard- or super-), this way allowing for repeated (super-)columns – names of the (super-)columns are created at the insertion time and are thus becoming carriers of information (cf. meta-information in usual case).

In the column-family model, concept of a record is maintained by a record key. Data chunks of the same record stored in different column families are identified by the same record key. Within one column family, record keys must be unique. Among column families, record keys may be the same. So, this way, record identity is conveyed.
5.2 Graph Databases

Graph databases are aimed at efficiently traversing between entities according to the relationships between them. The model (Robinson & col., 2013) consist of nodes representing entities, connections representing relationship between the nodes, and properties of both. Typing or classification can take place.

Compared to SQL databases, the model does not need foreign keys; the connections are recorded by means of direct pointers. So, for traversing no index search is needed, making this way graph databases substantially faster. Of course, by-property search efficiency depends upon previous indexing.

The realization of connections is similar to network databases, data definition and manipulation is but quite easier, as compared to (CODASYL, 1971). Declarative query language, enabling to formulate searched patterns of nodes, relationships and properties (Neo4J, 2015), relieves the tedious and error prone work in case of procedural programming for the same task. Traversal-oriented language is supported, of course.

This feature of encoding relationships by pointers also represents a major limitation of graph databases – distribution of data is a problem. Horizontal scaling faces trade-off need. Cross-server data processing would never be as fast as by-pointer traversing inside one server database.

Residing on one server, ACID compliance can be efficiently implemented.

Graph databases aid for schema flexibility; adding new types of relationships or properties, and populating them in newly recorded facts is trouble-free.

Summary

If all application data can be managed by single SQL engine, stored data structures are stable and frequent application information needs would be met by simple queries, choice of an SQL database could be the best.

If your data need to be distributed, analysis of application data needs should be performed and with respect to that analysis data aggregates should be designed. If no data structure for these aggregates can be recognized or would be useful, a key-value store would be the right choice.

If the application would profit from referencing to separate structure parts of the aggregates, a document or column-family store could serve. When an object identity should be maintained across multiple aggregates, a column-family could serve.

When application information needs require much traversing between data records, a graph database would be appropriate.

When information needs are evolving or frequently changing, any NoSQL solution would help; better key-value, document store or graph database, then column-family.

If you think of a sparse table with very many columns, column-family is right choice.

References


CODASYL Data Base Task Group 1971: April 1971 report. ACM


Codd, E. F., 1990: The Relational Model for Database Management, Addison-Wesley Publishing Company


JEL Classification: C88, M15