Annales Univ. Sci. Budapest., Sect. Comp. 42 (2014) 173–198

A COMPARATIVE EVALUATION OF NoSQL DATABASE SYSTEMS

László Dobos (Budapest, Hungary)
Balázs Pinczel (Budapest, Hungary)
Attila Kiss (Budapest, Hungary)
Gábor Rácz (Budapest, Hungary)
Tamás Eiler (Budapest, Hungary)

Dedicated to Professor András Benczúr on the occasion of his 70th birthday

Communicated by János Demetrovics

(Received June 1, 2014; accepted July 1, 2014)

Abstract. With the quick emergence of the largest web sites in the mid-2000s, and the adoption of the cloud-based computing model, traditional relational database systems could not keep up with the requirements of high throughput and distributed operation. As a result, major web companies developed their own, inherently distributed, lightweight solution to act as a database back-end for their services. These developments spun interest in the open source world and numerous products appeared under the term NoSQL – not only SQL. In the present work we compare a few NoSQL systems (MongoDB, Cassandra, Riak) according to a wide set of aspects. We conclude, that while NoSQL systems offer much less functionality than traditional relation database management systems, especially in transaction isolation and scan operations, they can be successfully used when complex database logic is not, but large-scale, distributed operation is an objective.

Key words and phrases: NoSQL, Cassandra, MongoDB, Riak
1998 CR Categories and Descriptors: H.2.4 [Database Management]: Systems – Distributed databases
This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013). It was also supported by the Hungarian grant OTKA-103244.
https://doi.org/10.71352/ac.42.173

1. Introduction

The enormous growth of database sizes during the last decade made monolithic database systems struggle to keep up with today's requirements. While relational database management systems (RDBMS) provide a very comprehensive set of functionality and are widely-used in almost every field where reliable data handling is necessary, they lack the out-of-the-box support for multi-machine scale-out. Although some products based on the well-known RDBMS traits are available today, applications that require data and/or functional partitioning, either because of the sheer size of the data or for the purpose of load balancing, have to rely on custom-built solutions or utilize alternative database systems. As the broad functionality and large code base of SQL database systems make implementing distributed partitioning a real challenge, with the high demand for multi-machine databases a whole set of NoSQL (not only SQL) products have emerged to fill in the niche left open by RDBMS.

The common features of NoSQL products are the divergence from the relational data model, the simplification of the transactional model and transaction processing, and most importantly, the shift to the imperative programming model from the declarative-style SQL language. As a consequence to these simplifications – which make it possible to implement NoSQL systems with a relatively small footprint, at least compared to the mainstream SQL products – a multitude of different NoSQL products are available with a very broad spectrum of different data models, functionality, API, storage models, programming models, as well as licensing and support options.

In this work we briefly review the most important aspects of NoSQL products, paying attention to the simplicity of their application in existing systems. First we establish a characterization scheme according to which NoSQL products can be evaluated and classified. In Section 2 we touch upon the topics of logical and physical data models, distributed storage abilities, and network application layer implementation and topology. Section 3 is devoted to the details of transaction processing in distributed systems. Properties of NoSQL systems from the aspect of system integration are discussed in Section 4. In Section 5, we evaluate three products (MongoDB, Cassanra and Riak) according to the points we establish earlier. We conclude our paper in Section 6.

2. Data models

NoSQL is an umbrella term for a rather diverse set of products. Any comparative evaluation of them should start with their classification according to the implemented data model, storage model and indexing support. While the logical data model and query capabilities together determine the usage scenarios of a system, the physical data model is primarily responsible for system performance. In comparison with relational databases, NoSQL products are usually very immature in both query capabilities and physical storage optimization. On the other hand, they support distributed operation intrinsically, which, when installed in cloud environments, often makes them an inevitable choice over traditional SQL products.

2.1. Logical data models

NoSQL systems are designed with easy sharding in mind, and with no requirement to support joins. As a consequence, products with a great selection of data models are available. Data models are designed to be quite flexible in order to support the storage needs arising from applications dealing with highly heterogeneous data. Also, the wide-spread use of dynamically typed scripting languages has made less strictly structured background storage system favourable. While a highly generic data model looks reasonable from the aspect of the client, efficient server-side processing makes certain restrictions on the data model necessary. As a result, many NoSQL systems offer semi-structured models and list-like data types. A primary objective of NoSQL database systems is to evenly distribute data among shards. As this is achieved via key hashing, any NoSQL data model is necessarily some kind of derivative of the most generic key-value stores. In the following, we discuss some data models supported by some NoSQL products starting from the most generic ones.

Records and collections: To distinguish them from the table concept of relational databases, we will refer to sets of records of NoSQL databases as collections. Records within a collection are identified by keys that must be unique and sortable. The existence of a unique key is a basic requirement of a database system that does not support scan operations or does sharding. In the former case keys are evidently necessary to address records, while in the latter case the definitions of the shard boundaries depend on a unique, possibly uniformly distributed key with well-defined ordering. The most basic data models use simple keys either automatically assigned by the database system, or provided by the user. Automatic key generation has the advantage of the ability to enforce the uniformity of the distribution of key values which is necessary to uniformly distribute data over the shards (assuming a well-behaving distribution of the data entries). Automatically generated keys, on the other hand, make scan operations (or at least mimicking scan operations) harder. NoSQL data models may differ in what restrictions on the format and size of the data field (or fields) of the records are made. In contrast to relational databases, values associated with keys often store complex data structures, like sets or lists. In case of NoSQL products which support versioning, a time-stamp is also associated with each record in addition to the key. Key lookups then made by either key–time-stamp pairs or default to the most recent records.

Key-value data model: The simplest, and most generic case of NoSQL data models, providing key-based access to data is the key-value data model. While keys have to be unique and well-ordered, no restrictions on the data itself are imposed (except usually some size limit), thus giving great flexibility to the developer to store virtually anything in a key-value store. In this case the interpretation of raw data is entirely up to the client program which makes interoperability and server-side programming an issue. If the database provides support for handling semi-structured values, it is sometimes called a document store. The very generic nature of values can make server side processing, such as indexing of collections a challenge, even when using standardized formats like XML or JSON. When storing complex and big data in a document store, it becomes necessary to provide partial access to the documents, either in forms of random access streams, or by more sophisticated ways, for instance, benefiting from the hierarchical structure of JSON documents. Certain key-value stores provide functionality to store and handle special types of values as data fields, such as arrays, sets, lists, hash tables, ordered sets etc. of various types. The big advantage of providing such API for the developers is that processing required to handle these data structures can be taken from the client-side to the server nodes reducing network traffic and harnessing the processing power of the servers.

Composite key-value data model: As NoSQL solutions are intended to be used in highly distributed environments, certain features of the data model are necessary to support physical organization of the data according to the use cases. For instance, certain parts of the data have to be stored on a dedicated set of servers, higher level of consistency must be enforced on them, or access restrictions apply to them, etc. The composite key-value data model addresses this issue by introducing a key consisting of multiple parts. Under this data model each collection can have an arbitrary number of predefined, seldom modified field families. Each field family can have any number of dynamically created, weakly-typed fields (or columns). Fields in this case are defined on the record level only, and not on the collection level: in practice, the collections are implemented in a way similar to sparse tables, with composite keys, where composite keys consist of the key of the record, the name of the field family and the unique name of the field. Sharding is done by the record key but not by the field family name.

Sparse table data model: Under the table data model, we understand the well-known tables of relational databases with the additional requirement of a primary key. Tables consist of statically typed columns, consequently they have the properties well-known from the SQL world, but most often lack a schema, i.e. a predefined set of columns. In many implementations, columns to tables can be added dynamically, quite often organized by column families following the concept of Google's BigTable [6]. As more NoSQL products tend to provide a limited set of the SQL language as the primary programming interface, they converge more towards the table data model and abandon the concepts of semi-structured documents and list data types. It is also very important to emphasize here, however, that NoSQL products do not support join operations, which makes it much harder to map complex data structures to tables as opposed to the case of relational database systems.

2.2. Physical storage models

As in many other aspects, the physical data storage models of NoSQL products all diverge from the SQL world. While most mainstream SQL products use disk-based storage and B-trees to store row data and optimize for random key-lookup, in-place updates and scans, NoSQL products are often much less sophisticated.

As most NoSQL products are much more flexible than the strict table model of relational databases, physical storage optimization, especially for in-place updates is a bigger problem. In case of document stores, for example, abandoning B-trees is an obvious choice as the size of the documents can vary significantly. Data is most often laid out on the disk according to key ranges, but the organization and access of the individual records within those key ranges changes from product to product. A recurring feature of NoSQL products is copy-on-write inserts and updates. This eliminates the complex logic of index page splits, tricky locking scenarios and independent transaction logging, but might lead to significant overhead with read operations when records are updated often. This is probably one of the main reasons, besides sharding, that scan operations are rarely supported, or if implemented, are significantly restricted compared to the SQL world. In heavy update scenarios, copy-on-write operation makes data files fragmented thus regular data compaction necessary, which we consider a serious drawback. Because of the flexible logical data model and data format of most products, the storage size overhead, which is often significant, is an important factor when considering a product for a certain application.

2.3. In-memory operation

With easier access to big memory hardware, in-memory, OLTP-oriented databases are becoming more common, consequently we were interested how well NoSQL products can be used for in-memory data storage. Many products can be configured to use the main memory for data storage primarily, but this sometimes requires tricks, like RAM-disks, using third-party back-ends or memory-mapped files. When tricking NoSQL products into in-memory operation via RAM disks or memory mapped files, the significant storage overhead of the systems instantly becomes a serious bottleneck. Also, because of the copy-on-write operation, frequent memory compaction would be necessary. A few products are designed to act as distributed in-memory cache and support cache expiration. Certain advanced SQL products, most importantly high performance OLTP systems and data warehouses, are already optimized for in-memory data locality (for better CPU cache hit rate) and non-uniform memory access (NUMA), an unavoidable property of big memory machines. Although NoSQL products are much less sophisticated in this respect, but they are admittedly targeted towards clouds of commodity servers rather than big iron machines.

2.4. Transaction logs, back up and restore

During standard operation, NoSQL databases intend to provide durability via replication to multiple, often geographically distributed server nodes. Journaling is most often achieved via copy-on-write storage implementation. When disk-based durability is required, the few in-memory NoSQL products often support snapshots, check-pointing and journals written to the disk. NoSQL code bases are usually developed according to the urgent needs of a few main customers. Because of this and because systems are almost always replicated, back up features are usually rather immature. This often manifests in very long back up and/or recovery times. Our experience is similar with large-scale data provisioning: initial data loading times are often surprisingly long (hours for a few millions of records) and no bulk-load feature is implemented.

2.5. Sharding and replication

One fundamental design feature of NoSQL databases is that they run on a medium or large cluster of machines. While they are often mentioned as data stores for the cloud, we should distinguish the loosely bound set of machines termed “the cloud” from the precisely configured database clusters that most

NoSQL systems require to yield sufficient performance. Nevertheless, certain software vendors claim that their NoSQL solutions can scale out to thousand of machines. As scale-out capabilities and performance depend significantly on the underlying network topology and the organisation of shards and replicas over the cluster nodes, we need to discuss this topic in detail. Also, there is a significant variation from product to product regarding the data layout which makes it an important decision factor when choosing one product or the other. Another important aspect of the sharding strategy is how difficult it is to add and remove cluster nodes on-line. Because key lookups are the most fundamental operations in the NoSQL world, sharding is always done based on the key, or on a hash of the key. Certain systems may or may not support custom hash algorithms.

Consistent hashing: When shard splits are common, i.e. the system is often extended with additional nodes when the data size is growing, the use of a consistent hash [11] to define shards is a common practice, e.g. as implemented in Amazon Dynamo [8]. Consistent hashing has the property of minimising the need of remapping when the originally K keys are reshuffled into m buckets. Contrary to ordinary hashing where almost all keys are to be moved, consistent hashing requires moving only K/m keys. When mapped to the hardware, shards (continuous hash ranges) are organized around a ring by wrapping the hash value around at maximum to zero. Server nodes are also imagined to be around a ring, hence adjacent shards can be assigned to adjacent nodes. Replication is done similarly, the replicas of a shard are assigned to the next few adjacent nodes. Moreover, certain products allow the customization of the mapping of shards and replicas to the available nodes. By customising the mapping, cluster nodes can intentionally be unbalanced, if required.

Replication strategies: Although several slightly different replication strategies exist, they most often store a single data entry on at least three nodes. When replicating, it is very important to ensure that the replicas of the same shard reside on different *physical* machines, which is not always a trivial task in cloud environments where many virtual machines might run on a single server. Replication is usually done on the shards level, multiple replicas belonging to the same shard. It is usually a good practice, therefore, to set the number of shards as a multiple of the number of available physical nodes the following way $s = r \cdot n$, where s is number of shards, r is the replication factor and n is the number of available physical machines. The number of replicas is most often chosen to be an odd number, so that majority voting never results in a tie, at least not during normal operation. By setting the number of shards as specified above, each machine will contain r different shards. Combined with consistent hashing, this strategy results in an automatically balanced system. Certain systems support location aware replication which can take network bandwidth limitations and latency into account. This feature is particularly useful for multi-rack and geo-redundant installations.

3. Transaction processing

Distributed systems are inherently subject to network failures and are more likely to suffer from single node hardware failure. Moreover, NoSQL systems are targeted towards cloud applications which usually run on a large number of cheaper servers which makes the hardware layer even less foolproof. As a consequence, reliability is implemented in the software layer via replication of the data to usually at least three separate servers. Reliability comes with a price: the replicas of the data may become inconsistent after a partial system failure. Consequently, NoSQL systems are essential to be classified according to the implementation of distributed transaction processing and conflict resolution strategies.

3.1. Consistency, availability and partition tolerance

NoSQL developers implemented several existing and newly-invented protocols to handle replication and resolve conflicts among inconsistent replicas. These algorithms relax the strict ACID system the following way. A minimum consistency level has to be defined for each read and write operation, so that the NoSQL system knows how many replicas need to be considered and be consistent to give a valid answer to a query. Furthermore, being able to relax write consistency has the very beneficial side-effect of also being able to increase the availability of the system. A write query can report completed once the changes are made to the required number of replicas instead of waiting for all servers to replicate the data. In theory, this should also increase the throughput of the system. To achieve total consistency, as it was shown by Gifford [9], one should set the sum of number of replicas written (W) and number or replicas read (R) to be larger than the number of all replicas: $R + W > r$. NoSQL systems relax this equation and allow the systems to work inconsistently, at least within a given time period, the so called consistency window.

Brewer [5] conjured, later Gilbert & Lynch [10] proved that hundred percent consistency, availability *and* partition tolerance of a distributed database system cannot be guaranteed under any circumstances at the same time. Brewer's CAP theorem is rather theoretical but one can view it quite practically. Higher consistency requires comparing more replicas which obviously decreases the availability (and throughput) of the system, but the problem comes when partition tolerance is necessary, too. In a partition tolerant system, data and services are available at many entry points and parts of the system remain operational even when all network connections between two partitions are lost after a failure. In this case, clients may talk to separate partitions (but always the same partition for the same client) of the distributed system at the same time. The different partitions will not be aware of the changes made to other partitions until network connections are restored and the potential conflicts are resolved. As the conflicts between replicas of the data are natural in partition tolerant systems, and some conflicts might be unresolvable, we lose hundred percent consistency. There is, however, much room to balance among the three properties: partitioning can be minimized by using reliable hardware while consistency and availability can be tuned by the number of servers, consistency settings and the replication factor used.

3.2. Data manipulation

Compared to relational database management systems, NoSQL solutions provide a restricted set of data manipulation capabilities, including restricted query abilities. In case of many products, the lack of server-side processing capabilities is a serious issue, when implementing complex transaction logic is an objective. Manipulating data often requires multiple request-response rounds between the client and the server which results in multiplied network latency.

Search capabilities: In order to efficiently deal with difficulties arising from the distributed nature of the systems, most NoSQL databases focus on the simplest search operations: key lookup, and – if indexes are supported – index lookup. Search capabilities beyond these vary from product to product. One watershed among the systems is the ability to perform range scans. The feasibility of scan operations depends on the storage and sharding models: if records are not stored and/or distributed according to the ordering of the key (e.g. based on a hash of the key instead), then this scan operation cannot be implemented efficiently. Even when scans are available, returning results in custom order is usually not possible. Some NoSQL solutions support aggregation and complex analytical queries, either natively, or by implementing some variant of the MapReduce [7] framework.

Availability of secondary indexes: Because keys or hashes determine the order in which data records are organized among shards, key lookups are the only feasible operations that are supported by all NoSQL products. In case when keys are not hashed, the original order of keys can be maintained (at a price of unbalancing shards) and key range scans can be implemented. Secondary indexes further extend search capabilities by maintaining a list of bookmarks corresponding for each index value. They, consequently, allow for seek-like and range-scan operations, however these are an order of magnitude more complex as the bookmarks might point to data scattered across all shards. As a consequence, range scans are usually implemented by NoSQL systems in a MapReduce approach, and often only intended for the purposes of data analytics instead of transaction processing.

The ability to create and maintain secondary indexes is useful when the data is frequently accessed by some other attributes than the key. NoSQL products support this functionality at different levels. Some products completely lack this feature, requiring the developers to redesign the data model (using the frequently accessed attributes as keys), or to introduce separate, manually maintained index collections. Triggers (if supported) can facilitate the latter approach by initiating an update of the index on every update of the data. Other products have built-in support for indexes and encourage the use of them, some even to the extent that certain search operations can only be carried out in the presence of the corresponding index, i.e. the engine refuses to scan a whole collection, requiring an index lookup instead.

3.3. Transaction support

One key feature of NoSQL systems is the relaxed transactional model. NoSQL databases are built on the BASE* transaction model which allows fine tuning the consistency requirements when reading and writing data. In an ACID framework, data is only considered written, when all distributed components have the same copy of the data, the changes are successfully written to the transaction log and the transaction is committed on all servers. This is almost always done using two-phase commits. Because ACID writes put the system from one consistent state to another, properly isolated reads return the same data, regardless where they are being read from. Even in the ACID model, relaxing the isolation level might allow dirty reads. NoSQL systems allow balancing the consistency level and system availability (i.e. transaction processing bandwidth) by specifying the number of server nodes, all containing the same set of replicated data, that are required to have the same copy of the data after a read or write operation completes. When the consistency level is set to majority, the BASE transaction model is theoretically the equivalent of the ACID model. In practice, significant difference appears when the system is failing and full consistency cannot be achieved.

The eventually consistent transaction isolation model allows for fine tuning the consistency level at which read and write operations are performed. Most system provides functionality to set the number of replicas that are required

*Basically Available, Soft state, Eventual consistency to agree in a certain value of a record during reads (denoted R) and that are required to report completion during writes to commit a transaction (denoted W). Typically, one wants at least $\lceil \frac{r+1}{2} \rceil$ replicas to take part in the voting, a so called *quorum*, to ensure majority. Many applications, however, can benefit from dirty reads and quick writes. Another way to balance consistency is to tune for read-heavy or write-heavy workloads. One might require quick writes (requiring, let us say only one replica to commit before reporting the transaction complete) and then require a quorum to agree on read values and resolve possible conflicts at read. Consistency can be tuned towards write-heavy scenarios similarly.

Transaction atomicity in NoSQL systems is realized only for single-record reads and writes. This can be a serious limitation, especially when implementing complex business logic, that needs to be thoroughly considered before choosing NoSQL systems for these kind of projects over SQL databases. Certain systems, however, support simple, synchronized batch execution of data modifying instructions, or atomic read-then-modify operations.

Transaction isolation is usually supported only on the record level. This is not surprising, as it is usually the case in the SQL world, but certain NoSQL systems (like document stores) can contain complex, mostly schema-free or hierarchical data as record values, in which case sub-record transaction isolation would be desirable. In the relational data model, correct isolation is solved by normalizing the schema and storing complex data types “flattened-out” in auxiliary tables. This is not always possible in NoSQL systems where join queries are not supported.

Some products, usually only their enterprise editions, support location-aware transaction processing. These systems can take network bandwidth and latency limitations into account when selecting a quorum to answer a request.

3.4. Conflict resolution

Even distributed systems with a single entry point are prone to inconsistency because certain parts of the system can go off-line for indefinitely long periods of time. Multiple entry points guarantee conflicts, thus any distributed system requires additional logic for conflict resolution. There are various ways to approach the problem. The simplest case is a “last-write-wins” scenario, in which records are time-stamped – note, however, that timer synchronization is an issue – and in case of a conflict, always the record with the latest time-stamp is chosen. Another way of dealing with conflicts is to detect them, but leave it to the client to resolve them. This is a particularly useful, though inefficient and cumbersome solution, especially for users. Because many data models allow for storing complex data in records, last-write-wins is not a good strategy and often individual data fields might need to be merged to resolve conflicts. The best time for conflict resolution is at quorum reads, as this is when differences among replicas can be detected.

3.5. Request routing in distributed environments

Typical NoSQL configurations run on many machines which instantly raises the question of the whereabouts of the network entry point. This question is of extremely high importance, as, by default, clients are not supposed to have detailed information about the topology of the database cluster. Various strategies exist: a) a central controller (or redundant controllers) might forward transactions to the appropriate cluster nodes or b) they can redirect clients to the nodes so that network communication between the client and the server containing the data is done directly. In a highly distributed system a single entry point would be a waste of resources as the overall performance would be limited by the throughput of a single coordinator node. To avoid the controller, a peer-to-peer approach can be implemented in which any cluster node can answer client requests and they either c) forward transactions to the nodes containing the requested data or d) redirect the client to talk to the node directly. Smarter clients can cache redirection information after the first request and send all subsequent requests to the appropriate machine directly. Real smart clients may be able to query the system topology and route requests according to it. Because NoSQL products implement only one of these strategies, but other aspects of the systems dominate transaction throughput, finding the best request routing strategy is a challenge.

3.6. Load balancing over replicas

As the preferred way of achieving durability and redundancy in NoSQL systems is real-time replication (as opposed to back ups), we can always count with more than two available replicas of the same data, on multiple machines, at any time. In most cases, replicas are not distinguished from each other but certain systems define a master replica. By default, when the hash function is uniform, transactions are supposed to target all shards uniformly. If it is not the case, the multiple available replicas open room for load balancing. From the perspective of the client, load balancing can follow many strategies. There exists a set of rather generic load balancing strategies that can be used in distributed systems [2]. Some of these (round-robin, least-connection, latency-aware) are sometimes implemented in NoSQL clients by default, other vendors suggest using third-party schedulers. In highly-distributed large systems with hierarchical network infrastructure, the optimal load balancing strategy must also observe locality, and schedule replicas for transaction processing accordingly. Similarly to request routing, load balancing logic can be implemented either in a controller, or in a smart client.

3.7. Smart client capabilities

In the NoSQL world, multiple servers contain multiple (eventually consistent) copies of the same data. As a consequence, client requests need to be routed to the appropriate servers, either due to sharding, or because of load balancing. We have to distinguish two fundamentally different approaches to message routing: controller-based and smart-client-based. In the controller-based scenario, all incoming client requests are sent to a central controller machine (which, in this simple case, is a single point of failure). Smart clients, on the other hand, are aware of the system topology or, at least, implement logic to figure it out and cache topological information. Having knowledge about shard boundaries and locations of the replicas, a smart client is able to route requests to the appropriate servers directly, without talking to a central controller. This approach can significantly improve message routing, but can also introduce important difficulties, especially on the client side.

3.8. Congestion due to failing nodes

The most important problems are caused by failing cluster nodes, a common event in cloud environments. Smart clients have to not only know where requests need to be sent but also should be able to handle node failures and reroute requests if necessary. As low latency is one of the main objectives of near-real time databases, clients must remain responsive, even when a single node fails. This means that node failures should be detected as quickly as possible, otherwise requests might queue up increasing overall system latency significantly. Network time-out detection is the usual way of revealing failing nodes, but setting network time-out too low might also plague the system. Heartbeat monitoring of nodes is an option only if the number of clients is limited to a few. Because node failures are common, clients must also support asynchronous routing of requests, so no failing node can retard other transactions. Limiting the number of outstanding requests sent to a server node is a good way of handling this situation.

We emphasize one important problem with load balancing when near real-time operation is an objective. Since distributed systems most often detect node failures from network time-out, and network time-outs are usually set an order of magnitude longer than the expected transaction processing time, many requests might end queued up on the client side when a system node goes off line. One can imagine active algorithms that can detect and report node failures to the controller or directly to the client. Reporting to the clients is only feasible when the number of clients is low. In middle-ware application this is typically the case, hence there is high interest in researching such algorithms in the future.

3.9. Asynchronous request processing

Some use cases rely heavily on the availability of asynchronous operations in order to achieve good read or write performance, or to eliminate glitches and non-standard behaviour in partial failure situations. When implemented in the client library, asynchronous data manipulation often relies on operation system threads (basically synchronous communication with the servers) instead of call-back functions triggered by responses from the server via the network protocol. This can easily lead to filling up the thread pool and jamming the client, or, if the thread pools are unbound or the number of sets set to a high value, can cause significant kernel times due to thread synchronization.

4. System integration

A rare occasion is when a system can be rebuilt entirely from scratch, it is more often that the storage back-end of existing systems needs to be replaced for performance and reliability reasons. Because database software usually run on dedicated servers, it is query capabilities and client libraries that matter primarily from the perspective of system integration.

4.1. Query language and server side programming

Traditional relational database systems provide a more or less standardized way of formulating queries and data modification operations: the SQL language. NoSQL products used to define themselves specifically as databases that do not use SQL, which is likely to change in the near future. NoSQL products often lack a declarative query language and implement transactions as data manipulation function in the client API. A significant disadvantage of this approach is that functionality is often limited by the availability of certain functions in the API designed for a specific platform. It is quite frequent that a Java client of a product offers much more query and DML functionality than the C implementation or *vice versa*. If the database defines a query language (either imperative or declarative), then the capabilities of client drivers depend less on the implementation. Several products are moving from a custom query API to a significantly simplified version of SQL (e.g. CQL for Cassandra).

Because SQL products usually implement a full set of query operators, and there are more or less standard ways of mapping certain logical data models to the relational model, programmers are not much concerned about the completeness of the query language. The situation in the NoSQL world is significantly different. We have to emphasize that the data modelling capability of a NoSQL system is determined by both the logical data model it implements and its query language features. For example, using NoSQL systems as a store for objects from an OOP language is usually straightforward but dealing with a simple hierarchical or graph data model might pose serious problems.

When executing consecutive data manipulation operations, the network overhead of sending the requests one-by-one could be significant. This network overhead also appears when complex transactions require a value to be sent back to the client side in order to make a decision about the next step of the transaction. Traditional SQL systems solve these situations by using stored procedures. Many NoSQL products support some kind of instruction batching in which case multiple operation are sent to the server, or queued up at the server side and executed coherently. It does not always mean, however, that the multiple instruction are executed as an atomic transaction. A few products offer the opportunity to write server-side triggers that are executed atomically with write operations.

4.2. Client libraries and language support

The capabilities and characteristics of the client libraries can have a major impact on the usability of a NoSQL product. The ability to choose from a wide variety of client libraries designed for different languages is important for various reasons. Obviously, if one intends to use a NoSQL database as storage back-end for an application written in a specific language, having drivers in more languages increases the possibility of finding one that can be integrated seamlessly into the application. NoSQL systems usually target a wide range of applications, consequently they should support the three main development platforms (native C, Java, .Net). Products that use an imperative programming model – as opposed to the declarative style of SQL – usually adopt existing procedural languages to formulate data manipulation operation. Certain languages, especially scripting languages, are more appropriate for these tasks. JavaScript, Erlang, etc. are widely used for this purpose.

4.3. Documentation quality

When choosing a product for a certain system as a database back-end, beside performance and platform considerations, many practical aspects are need to be considered. Among these, we mention the quality of the documentation of the product, the simplicity and documentation of installation and licensing options. Probably because many vendors make their living not from selling the software itself, but by supporting it, documentations are usually not of the same quality as it is common in the SQL world. While the documentation of most products cover the general usage scenarios, important details on internal operation can often only be found in blog entries and web forums.

4.4. Customizability and extensibility

Under extensibility, we understand the means of changing server or client-side behaviour of the system by implementing user-defined extension logic. Since all reviewed products are open source, theoretically any behaviour can be customized. However, we concentrate here on customizability offered directly by the products API. The particularly interesting issues are the following. a) The ability to control sharding, either via custom hash functions or via customizing node assignment to shards and replicas. b) The ability to customize request routing to fine-tune load balancing and handle failure. c) The ability to control connection pooling and synchronization of request on the client-side.

5. The NoSQL products reviewed

Based on the aspects defined in Sections 2-4, we now briefly review three leading open-source NoSQL products: MongoDB, Cassandra and Riak.

5.1. MongoDB

MongoDB is claimed to be the most popular open source NoSQL software. Written in C++, it implements a document store.

Logical data model: MongoDB implements a document store. Hierarchical JSON documents are associated with keys of any basic data type and organized into named collections identified by a string. MongoDB neither enforces nor validates the schema of the documents. It supports, however, server-side access to the individual data fields of the hierarchical documents and can automatically build indexes on a given value of a sub-document.

Physical storage model: As the JSON format is rather verbose, MongoDB stores documents in the binary BSON format. While BSON offers a more compact storage format and fast scanning using size prefixes, it is still very tedious compared to fixed-schema storage models as field names of the hierarchical documents need to be stored for each and every document. MongoDB accesses disk-based data via memory-mapped files thus memory management (the equivalent of page pool management) is delegated to the operation system entirely. This can sometimes result in weird behaviour, especially when multiple concurrent MongoDB processes run side by side. For example, large database files are being read into memory right after startup, and it can take tens of minutes before the system reaches stable operation. During high, update-heavy workloads, MongoDB tends to show effects similar to heavy thrashing due to the high number of hard page faults associated with the memory mapped files. As there is no mechanism to serialize disk access, thrashing results in random reads and writes which can very easily jam the IO system. MongoDB reuses space in data files which can, similarly to SQL systems, lead to heavy fragmentation. Data files can be manually defragmented using the so called compact functionality. Because of the use of memory-mapped files, MongoDB can simulate complete in-memory operation as long as data file sizes fit into the memory. Data files, however, have a significant storage overhead which prohibits building economic in-memory system using the product. MongoDB ensures durability via writing a journal to the disk.

Sharding and replication: MongoDB supports two fundamental approaches to sharding: range-based or hash-based. Hash-based sharding can uniformly distribute data among server nodes but mix the order of records, thus prohibits effective scan operations over ranges of data keys. Sharding is implemented with two granularity levels: each shard consists of a set of chunks, also defined by hash or key ranges. Chunks are automatically split into halves as data size grows and are automatically migrated to different shards (different machines) when it is necessary to maintain the balance of the cluster. In case of range-based sharding, key ranges can be manually associated with certain shards – and in turn, servers storing them – via a process called tag-aware sharding.

Transaction processing: MongoDB supports transaction isolation on the single row level, though multi-document inserts and updates are possible, the latter can be also synchronized within single shards, but not across the whole collection. Interestingly, newer version of the product has started providing two phase commit functionality for cross-shard transactions. Single row isolation is provided via a collection-level lock. The level of consistency of data modification statements can be set via a write concern parameter (specific number of shards or majority). Although this synchronizes access to the collection, the documentation claims that it is not an issue as most MongoDB deployments are heavily sharded and locks are per shard. MongoDB supports scans at an isolation level similar to read committed.

A rather puzzling limitation of MongoDB is that it offers not way to enforce a time-out on single-record data modifying operations. In a write-heavy workload, many queries can get queued up due to the collection-level lock used for synchronization and can stay in the queue indefinitely. One can attempt to kill queued up operations after a certain amount of time but the kill operation can either happen before or after the write itself has completed. As a result, the client cannot be sure about the outcome of the transaction which we consider a serious issue.

Query capabilities: MongoDB implements its own mixed imperative-declarative programming interface in the form of JSON objects with a great selection of operations. Single and multi-document data modification operations are supported, as well as scans, aggregation, MapReduce operations and geo-spatial searches. As it was mentioned above, multi-document operations are usually not atomic, nor isolated.

Cluster architecture: A single MongoDB process is responsible for storing a single shard of a larger database. Multiple replicas of the same shard require multiple running processes. Information about the system configuration, when sharded and replicated, is stored in a central database, also stored in a MongoDB instance. Sharding and replication is supervised by a controller process, which forms a single entry point for cluster-wide queries, but individual shards can also be queried directly. Multiple controller instances can be run in parallel for fail-over and load balancing purposes.

Query language and server-side programming: MongoDB has its own query language, operations are described and parametrized as JSON documents. It is a convenient way of writing queries in string format, but composing JSON documents anagrammatically can be cumbersome. A very powerful feature of MongoDB is its support of server-side execution of JavaScript for scan and MapReduce operations.

Client libraries: MongoDB, by default, is accessible via a console interface, making the testing of the system very easy. It offers official drivers to a wide variety of platforms including native C and manages libraries for both Java and .Net. As MongoDB communicates with the client via JSON documents, all server features are available to all types of clients.

Usability and extensibility: MongoDB is a relatively well-documented, easy-to-install system. Its documentation, however, lacks the discussion of many details that are important from the aspect of system integration, namely limitations. MongoDB comes with a large set of tools for system configuration and monitoring. Its server-side extensibility is limited, probably by the fact that it is implemented in a native environment.

5.2. Cassandra

Cassandra is a sparse table store developed in Java and features the programming language CQL (Cassandra Query Language) with similar syntax to SQL but with rather limited functionality.

Logical data model: Cassandra uses the sparse table data model, organising columns into *column families*. Two types of column families are distinguished based on their intended ways of use. A *static* column family has the same set of columns (fixed at creation time) for each row, with occasional missing values – much like an ordinary relational table, while a *dynamic* column family can have arbitrary columns for each row. The internal representation of the values is the same for both types of columns, but they are distinguished from the aspect of query language. Columns are typed, using one of the numerous available data types, such as text, boolean, numeric types of various precision, uninterpreted binary data, universally unique identifier, atomic counter, etc. To store somewhat more complex structures, collections (lists, sets and maps) can be used, however, collections of collections are not allowed. Cassandra provides built-in support for automatically-maintained secondary indexes of static columns. Since search operations generally prohibit scans, the existence of an appropriate index is required in order to impose a filter condition on a column.

Physical storage model: The storage model is based on concepts familiar from Google's BigTable [6]. For each column family, Cassandra maintains an in-memory structure called a *memtable*, which stores recently modified row fragments that can be looked up by key. Once a memtable reaches the specified size limit, it gets flushed to disk as an *SSTable*, an immutable persistent storage unit that stores row fragments sorted by the hash of the key. To serve a read request, the database server has to combine row fragments from the memtable of the column family as well as potentially multiple SSTables that may contain relevant data according to a Bloom filter [4]. To reduce the number of data files that have to be searched, a background process regularly performs compaction, merging row fragments from old SSTables to form a single new one, while removing stale versions of data in the process. Cassandra provides durability via a continuously written commit log, though logging latency can be as high as a few milliseconds.

Sharding and replication: Data distribution uses the consistent hashing method, with tokens assigned to either virtual or physical nodes, and multiple options for the hash function (including an order preserving one, and the possibility to provide custom functions via implementing a Java interface). With a replication factor of r, replicas of a row are placed on the r consecutive nodes of the ring, starting with the node which owns the appropriate hash range. If topology information is present, a more advanced replica placement strategy can be chosen, which avoids placing more replicas on the same rack, as nodes in the same rack might become inaccessible at the same time with higher probability. Tuning between availability and consistency is achieved through individual R and W values for each request, representing the number of replicas required to participate in fetching and committing a value, respectively.

Transaction support: Support for transactions is provided only at single row level, atomic and isolated execution of transactions affecting more than one rows is not guaranteed. Since Cassandra 2.0, lightweight transactions have been introduced, which help to avoid race conditions of *read-then-modify* style updates using a conditional construct (`INSERT` ... `IF NOT EXISTS` and `UPDATE` ... `IF` ...). Also, atomic incrementation is possible with the counter data type.

Query capabilities: Cassandra has its own query language called CQL, with data manipulation and data definition features, using an SQL-like syntax. For performance reasons, search capabilities are limited to key lookup and index lookup in most cases, although some scan operations can be performed by explicitly permitting potentially expensive queries. Result rows can be sorted only by the column key. Support for querying a key range is limited to the case when the hash used for data distribution is an order preserving one.

Query language: The latest versions of Cassandra use the CQL language as primary query language. CQL is largely based on SQL, although it does not support multi-table constructs. CQL distinguishes static and dynamic column families: static columns can simply be listed in the select list while dynamic columns can only be accessed via a composite-key syntax (basically moving the column reference into the where clause). We consider this distinction of column family types by the query language a major drawback. Although the CQL language appears to be attractive at first, its hidden differences from SQL do not make the learning curve of the language less steep than those of completely different languages or APIs. Familiar SQL constructs may not work the expected way, or may not work at all. Also, the abstract representation of the queries does not reflect the internal storage structure well, making the task of figuring out efficiency implications of CQL operations non-trivial. Apart from CQL, there exists a legacy API based on Apache Thrift [1], with low-level data access without abstractions, however it is considered deprecated.

Client libraries: Client libraries to Cassandra come in a wide variety of languages, although, as the product itself is developed in Java, the Java driver is considered to be the main, most feature-rich one. It offers lots of smart capabilities including node discovery, transparent fail-over, and tunable load balancing. The driver features a fully asynchronous client API and a freely customizable retry policy allowing further flexibility in tuning CAP properties.

Usability and extensibility: Apache Cassandra is open-source, licensed under the Apache License (Version 2.0), and an enterprise edition also exists with advanced functionality (such as in-memory features, management and monitoring tools) and support offered by DataStax. The products are easy to deploy and use, thanks to the comfortable installation process and comprehensive documentation. Cassandra is also outstanding in terms of extensibility: in addition to the opportunities arising from its open-source nature, several extension points exist where the server- or client-side behaviour can be modified or extended with custom logic without rebuilding the source. This is achieved by placing the customized implementation of pre-defined interfaces on the Java classpath, and configuring the system to use them instead of the built-in options. The customizable features include the partitioner (the embodiment of the hash function used in sharding) and the replica placement strategy on server side, and the load balancing policy and retry policy on client side.

5.3. Riak

Riak is a key-value store developed in Erlang. It supports a wide selection of various physical data storage back-ends.

Logical data model: Riak uses the key-value model for storing data, where keys are binary values or strings that uniquely identify objects, the units of storage. The key-value pairs are organized into virtual namespaces called buckets. Objects are composed of a bucket-key pair, a vector clock (a type of time-stamp used for conflict resolution), and a list of metadata-value pairs. Multiple metadata-value pairs may occur when more than one actor updates the same value simultaneously and the system stores multiple versions of data as siblings.

Physical storage model: Riak implements an extensible storage back-end model, so the system itself acts as a distributed processing coordinator umbrella over a set of data storage engines. Three back-ends are supported by default, namely Bitcask [12], LevelDB [3], and Memory. Bitcask is an Erlang application implementing a log-structured hash table; entries are stored in write-ahead logs on disk, and a hash table is kept in memory that maps every key to a fixed-size structure giving the file, offset, and size of the entry belonging to the key. The obvious drawback of this is that the entire keyspace must fit into the memory, and data access performance and overall throughput is directly limited by the random read and write performance of the underlying IO system. LevelDB is an open source on-disk key-value store developed at Google. It stores keys and values in arbitrary byte arrays and data is sorted by key. If the keyspace does not fit into memory, it is a good alternative to Bitcask. The Memory back-end is non-persistent storage based on Erlang's *ets tables*, providing the quickest access to data. Riak can further be extended by third-party storage back-ends and supports setting the storage engine used at the bucket level. This basically allows mixing different storage strategies within the same system.

Sharding and replication: Riak follows the consistent hashing technique in data distribution, where entries are mapped to a virtual *keyspace* based on the SHA-1 hash of the keys. The keyspace is divided into partitions which are called *virtual nodes*. Each virtual node is associated to a physical node of the cluster trying to keep the amount of data stored on each server equally balanced. With a replication factor of r, replicas of a row are placed on r consecutive partitions in the virtual keyspace, the first of which is selected based on the hash. However, it is not guaranteed that every replica is stored on a different physical machine, because the basis of replication in case of Riak is the virtual nodes and not physical machines.

Transaction support: Transactions are not supported in Riak, only writing a single key-value pair can be seen as atomic operation. However, as it is an eventually consistent system, inconsistency can occur among replicas. In that case, conflict resolution is based on vector clocks which help tracking a true sequence of events of single objects by storing time-stamps along with objects. The consistency level of read and write operations, similarly to Cassandra, can be configured via R and W values for each request independently. During normal operation, read and write requests are directed to the first R and W out of the r consecutive nodes owning a replica of the data. Riak, however, supports further options for tuning between availability and consistency via its “sloppy quorum” feature. When this is enabled, a failure situation may cause the requests to be directed to other nodes (not being one of the r original owners of a replica), which can temporarily take over the role of inaccessible machines.

Query capabilities: Riak provides three ways to query data beyond simple key-lookup, namely, secondary indexes, a full-text search feature termed Riak Search, and a MapReduce framework. The availability of these depends on the actual storage back-end and client library used. Secondary indexes can only be built on additional attributes stored alongside the documents. The data type of these attributes is limited to integers and strings. A third, special type of attribute is a link to another key. Links establish a one-way relationship between keys and the relationships can be tagged with a label. Secondary indexes can be queried by equality and range queries.

Riak Search is a distributed full-text search engine. The full-text indexes are automatically updated at write time using “commit hooks”. Commit hooks extract and analyse data depending on its mime type; several mime types are supported by default such as JSON, XML, and plain text. Riak Search provides a query language over search indexes with capabilities of exact match, wildcards, inclusive/exclusive range queries, grouping and basic logical operators.

The Riak MapReduce framework allows performing ad-hoc queries composed of an arbitrary number of Map and Reduce phases implemented as Erlang or JavaScript functions. The MapReduce framework can take advantage of parallel processing and data locality, however, it is designed for batch processing and not for real time operations. As a result, it is rather slow for simple queries due to the high flat cost of initializing the framework, hence it is not a substitute for scan operations.

Query language: The primary data access interface of Riak is a REST HTTP front-end. Key-value pairs can be consequently manipulated via REST request using various HTTP verbs. Query parameters are submitted as query string parameters. MapReduce jobs can be written in Erlang or JavaScript, in the latter case in form of JSON objects. Riak also offers customizable commit hooks, functions similar to SQL triggers, which are executed before or after each write operation into a certain bucket.

Client libraries: Riak offers two significantly different interfaces for accessing data: the REST HTTP API and the so called Protocol Buffer Client interface. The former is significantly easier to use but suitable only for testing purposes due to the known limitations and overheads of the HTTP protocol. The latter protocol, however, is binary and offers much higher performance. A number of open-source client libraries for a variety of programming languages and platforms are supported: Erlang, Java, PHP, Python, Ruby. Interestingly, the availability of certain server functionality (secondary indexes, link walking) depends on the client used.

Usability and extensibility: Similarly to Cassandra, Riak is open-source and licensed under Apache License (Version 2.0). A commercial edition also exists with multi-datacenter replication feature, advanced monitoring tools, and Java Management Extensions (JMX) support. In addition, Basho offers engineering and customer support in commercial packages. Although the products are easy to deploy and use, the documentation is not complete yet and the advanced settings are rather hard to find.

6. Summary and future work

It the present work, we have reviewed three popular NoSQL products according to an evaluation method presented in Sections 2-4. The aspects of evaluation span from the physical storage model to client features and extensibility and at every level compared to features offered by mainstream SQL products. We discussed that NoSQL systems significantly lag behind SQL system in terms of programmability, data modelling capabilities and transaction isolation support, but can be scaled out to many machine configurations very easily. As a conclusion, we think that NoSQL systems, in their current form, are appropriate as data back-ends for applications where business logic is rather simple and transaction isolation is not a central issue, but high transaction throughput is a requirement, for instance, social network web sites. Due to the lack of highly optimized disk access, NoSQL systems perform well in highly distributed environments rather than on "big iron" with sophisticated RAID systems.

We reviewed three products in more detail: MongoDB, Cassandra and Riak and pointed out a few design flaws that are important to know before any decision about the adoption of the products. MongoDB turned out to a be a feature-rich product with some important caveats regarding transaction atomicity and durability. Cassandra offers a simple, yet not so obvious query language based largely on SQL and offers great customizability on both server and client-side. An important strength of Riak is its ability to use different storage back-ends. Since the NoSQL world is changing fast, and products develop independently from each other, frequent reviews in the future are necessary to keep up with features implemented in new versions.

Nevertheless, this paper covered only theoretical aspects and performance metrics are the real factors of decision when a system is selected for adoption. We have thoroughly tested the three aforementioned products and will publish our results in another paper.

References

[1] Apache Thrift `http://thrift.apache.org/`.

[2] Job Scheduling Algorithms (LVSKB) `http:kb.linuxvirtualserver.org/wiki/Category:Job\_Scheduling\_Algorithms`.

[3] LevelDB `http://code.google.com/p/leveldb/`.

[4] **Bloom, B.H.,** Space/time trade-offs in hash coding with allowable errors, *Communications of ACM,* **13** (7) (1970), 422-426.

[5] **Brewer, E.A.,** Towards robust distributed systems (invited talk), *Principles of Distributed Computing,* Portland, Oregon, 2000.

[6] **Chang, F., J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes and R.E. Gruber,** Bigtable: A distributed storage system for structured data, *Proc. 7th Symp. on Operating Systems Design and Implementation OSDI'06, Berkeley, CA, USA, 2006,* USENIX Association, 205-218.

[7] **Dean, J. and S. Ghemawat,** MapReduce: Simplified data processing on large clusters, *Proc. 6th Symp. on Operating Systems Design and Implementation OSDI'04, Berkeley, CA, USA, 2004,* USENIX Association.

[8] **DeCandia, G., D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels,** Dynamo: Amazon's highly available key-value store, *Proc. Twenty-first ACM SIGOPS Symp. on Operating Systems Principles SOSP'07,* ACM, New York, NY, USA, 2007, 205-220.

[9] **Gifford, D.K.,** Weighted voting for replicated data, *Proc. Seventh ACM Symp. on Operating Systems Principles SOSP'79,* ACM, New York, NY, USA, 1979, 150-162.

[10] **Gilbert, S. and N. Lynch,** Brewer's conjucture and the feasibility of consistent, available, partition-tolerant web services, *SIGACT News,* **33** (2) (2002), 51-59.

[11] **Karger, D., E. Lehman, T. Leighton, R. Panigrahy, M. Levine and D. Lewin,** Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web, *Proc. 29th Annual ACM Symp. on Theory of Computing STOC'97,* ACM, New York, NY, USA, 1997, 654-663.

[12] **Sheehy, J. and D. Smith,** Bitcask – A log-structured hash table for fast key/value data, `http://downloads.basho.com/papers/bitcask-intro.pdf`

László Dobos
Department of Physics of Complex Systems
Eötvös Loránd Tudományegyetem
Budapest, Hungary
`dobos@complex.elte.hu`

Balázs Pinczel, Gábor Rácz, Attila Kiss
Department of Information Systems
Eötvös Loránd Tudományegyetem
Budapest, Hungary
{vic,gabee33,kiss}@inf.elte.hu

Tamás Eiler
Ericsson Hungary
Budapest, Hungary
tamas.eiler@ericsson.com