MŰEGYETEM 1782

Budapest University of Technology and Economics
Department of Measurement and Information Systems
Fault Tolerant Systems Research Group

Critical Architectures Laboratory
Spring Semester 2016/2017

Distributed Data Storage in NoSQL Databases

Syllabus
v1.4

Author: Gábor Szárnyas
(szarnyas@mit.bme.hu)

March 1, 2017

Contents

1 Introduction

This document contains the theoretical and practical background for the *Distributed Data Storage in NoSQL Databases* session of the *Critical Architectures Laboratory* course. The purpose of this session is to demonstrate the practical side of replication and distributed data storage (lecture notes in Hungarian are available in [15]).

1.1 Big Data and the NoSQL Movement

Since the 1980s, database management systems based on the relational data model dominated the database market. Relational databases have a number of important advantages: precise mathematical background, understandability, mature tooling and so on. However, due to their rich feature set and the strongly connected nature of their data model, relational databases often have scalability issues [13]. They are typically optimized for transaction processing, instead of data analysis (see *data warehouses* for an exception). In practice, these render them impractical for a number of use cases, e.g. running complex queries on large data sets.

In the last decade, large organizations struggled to store and process the huge amounts of data they produced. This problem introduces a diverse palette of scientific and engineering challenges, called *Big Data* challenges.

Big Data challenges spawned dozens of new database management systems. Typically, these systems broke with the strictness of the relational data model and utilized simpler, more scalable data models. These systems dropped support for the SQL query language used in relational databases and hence were called *NoSQL databases*[1] [4]. Because relational databases are not suitable for large-scale model-driven applications, we experimented with numerous NoSQL databases.

2 Concepts

2.1 Consistency in a Distributed System

2.1.1 The CAP-theorem

In 1999, Eric Brewer, a professor at Berkeley University published a set of informal requirements for a distributed system, called the CAP properties. Next year, in the keynote speech of PODC (Principles of Distributed Computing), he presented the CAP-conjecture [8]. The conjecture states that in any given moment, a web service can only guarantee two of the following properties: consistency, availability, partition tolerance. The concepts are roughly defined as follows [9]:

Consistency Consistency means if and how a system is in a consistent state after the execution of an operation. A distributed system is typically considered to be consistent if after an update operation of some writer all readers see his updates in some shared data source.

[1] The community now mostly interprets NoSQL as ”not only SQL”.

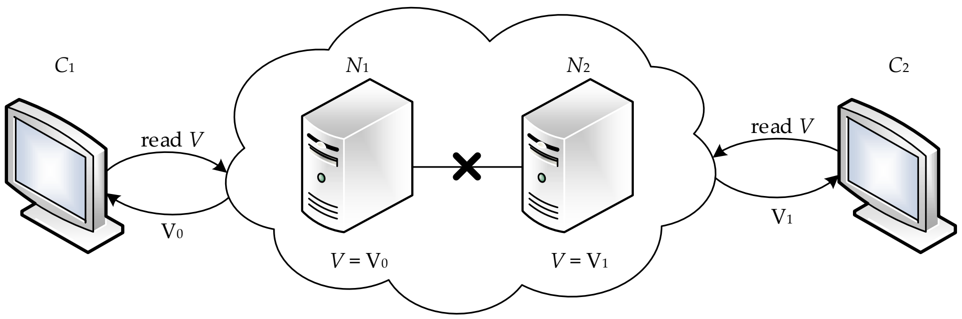

Figure 1: The V data item is not consistent, clients C_1 and C_2 see different values

(Nevertheless there are several alternatives towards this strict notion of consistency as we will see below.) See Figure 1.

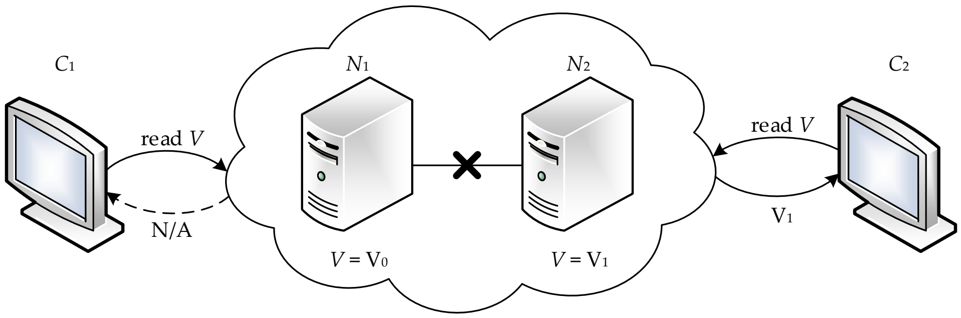

Figure 2: The V data item is not available for client C_1

Availability Availability means that a system is designed and implemented in a way that allows it to continue operation (i.e. allowing read and write operations) if e.g. nodes in a cluster crash or some hardware or software parts are down due to upgrades. See Figure 2.

Partition tolerance Partition tolerance is the the ability of the system to continue operation in the presence of network partitions. These occur if two or more ”islands” of network nodes arise, which (temporarily or permanently) cannot connect to each other. Some people also understand partition tolerance as the ability of a system to cope with the dynamic addition and removal of nodes (e.g. for maintainance purposes; removed and again added nodes are considered an own network partition in this notion).

The conjecture was formalized and proved in 2002 by two researchers in MIT (Massachusetts Institute of Technology), Seth Gilbert and Nancy Lynch [11]. In this form, the CAP-theorem defines a limitation for all distributed systems.

In short, the CAP-theorem can be stated as follows: in a distributed system in the case of network partition, the operations of the system will not be atomic and/or the data elements will be unavailable.

Formally: in a distributed system running on an asynchronous network, the system cannot guarantee the following properties:

- availability,
- atomic consistency.

2.2 Replication

Replication, consistency and availability are strongly correlated. We use the following notations [16]:

- N – the number of nodes that store a replica of the data.
- W – the number of replicas that need to acknowledge the receipt of the update before the update completes.
- R – the number of replicas that are contacted when a data object is accessed through a read operation.

Quorum protocols enforce that the following inequalities holds:

1. $W > N/2$
2. $W + R > N$

The first condition guarantees that any two write quorums have a mutual node, which makes it possible to preserve the order of subsequent writes. The second guarantees that each read quorum and write quorum overlap, so that we will not read stale data.

2.3 Sharding

Sharding is the process of determining the location of each data item in a distributed system. Sometimes this process is called *partitioning* or *segmenting*.

3 Apache Cassandra

Figure 3: The logo of Apache Cassandra

Cassandra is one of the most widely used NoSQL databases [2]. Originally developed by Facebook [14], Cassandra is now an Apache project. Cassandra is a column family database with advanced fault-tolerance mechanisms. It allows the application to balance between availability and consistency by allowing it to tune the consistency constraints. Cassandra is used mainly by Web 2.0 companies, including Digg, Netflix, Reddit, SoundCloud and Twitter. It is also used for research purposes at CERN and NASA [5].

3.1 Data Model

Cassandra uses the *column family data model*. A column family is similar to a table of a relational database: it consists of rows and columns. However, unlike in a relational database's table, the rows do not have to have the same fixed set of columns. Instead, each row can have a different set of columns. This makes the data structure more dynamic and avoids the problems associated with NULL values.

3.2 Serialization and Client Options

Cassandra uses the Apache Thrift framework [3] for serializing data. Like Cassandra, Thrift was originally developed by Facebook, but unlike Cassandra it is still used there.

The most straightforward option is to use the command-line *CQL Shell for Apache Cassandra* provided by the `bin/cqlsh` executable. While new Cassandra projects are encouraged to use CQL (the official Java driver is available at [10]), there are also Thrift-based ORM libraries available (e.g. [12]).

3.3 Sharding in Cassandra

3.3.1 Consistent Hashing

To distribute the data across the cluster, Cassandra uses a partitioner mechanism. The basic partitioners distribute the rows evenly based on their key's hash value using *consistent hashing*.

In a cluster with n nodes, the naïve method for determining the location for a row with key x is computing $h(x) \bmod n$, where $h(x)$ is the hash function. Currently, Cassandra provides partitioners based on the MD5 and the Murmur3 hash functions[2]. However, this approach has a serious limitation: if we remove or add nodes to the cluster, we have to recompute the hash values and possibly relocate almost all rows in the cluster. To avoid this, Cassandra uses a special kind of hashing called *consistent hashing*.

The ring is divided into ranges equal to the number of nodes, with each node being responsible for one or more ranges of the data. Before a node can join the ring, it must be assigned a *token*. The token value determines the node's position in the ring and its range of data. The ring is walked clockwise until it locates the node with a token value greater than that of the row key. Each node is responsible for the region of the ring between itself (inclusive) and its predecessor (exclusive). With the nodes sorted in token order, the last node is considered the predecessor of the first node; hence the ring representation[3] [7].

For example, consider a simple four-node cluster, where all of the row keys managed by the cluster are in the range of 0 to 100. Each node is assigned a token that represents a point in this range. In this example, the token values are 0, 25, 50 and 75. The first node (with

[2] From Cassandra 1.2, the default partitioner uses the Murmur3 hash function. Although the MD5 hash function has important cryptographic properties, these are irrelevant for distributing data. Hence, the MD5-based partitioner is no longer recommended.

[3] Note that *consistent* here is different from both the idea of consistency in *data consistency* and in the *ACID (atomicity, consistency, isolation, durability)* properties guaranteed by transactions. It refers to the fact that tries to map the same rows to the same machine, even if the number of machines (n) changes over time slightly.

token 0), is responsible for the wrapping range (76+), the second node (with token 25) is responsible for the data range $1 - 25$, and so on [7]. Figure 4 shows the token ring for this cluster.

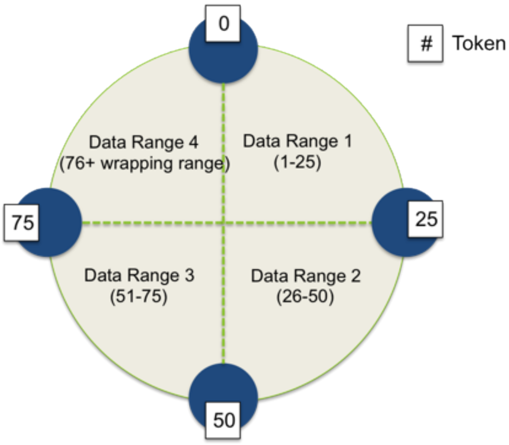

Figure 4: Cassandra's ring for data partitioning [7]

3.3.2 Virtual Nodes

In older versions of Cassandra (prior to version 1.2), you had to calculate and assign a single token to each node in a cluster. Each token determined the node's position in the ring and its portion of data according to its hash value. Although the design of consistent hashing used in older versions (compared to other distribution designs), allowed moving a single node's worth of data when adding or removing nodes from the cluster, it still required substantial effort to do so.

Starting in version 1.2, Cassandra changed this paradigm from one token and range per node to many tokens per node. This paradigm is called *virtual nodes* (vnodes). Vnodes allow each node to own a large number of small partition ranges distributed throughout the cluster. Vnodes also use consistent hashing to distribute data but using them doesn't require token generation and assignment [1].

3.4 Cassandra Query Language

Cassandra provides the Cassandra Query Language (CQL). The reference for CQL3 is available in [6]. CQL's goal is to offer a query language similar to the Structued Query Language (SQL) used in most relational databases.

CQL allows the user to set the data consistency on a per operation basis. The desired level can be set with the `CONSISTENCY [ANY | ONE | TWO | THREE | QUORUM | ALL | ...]` command. The `CONSISTENCY` command displays the current consistency level.

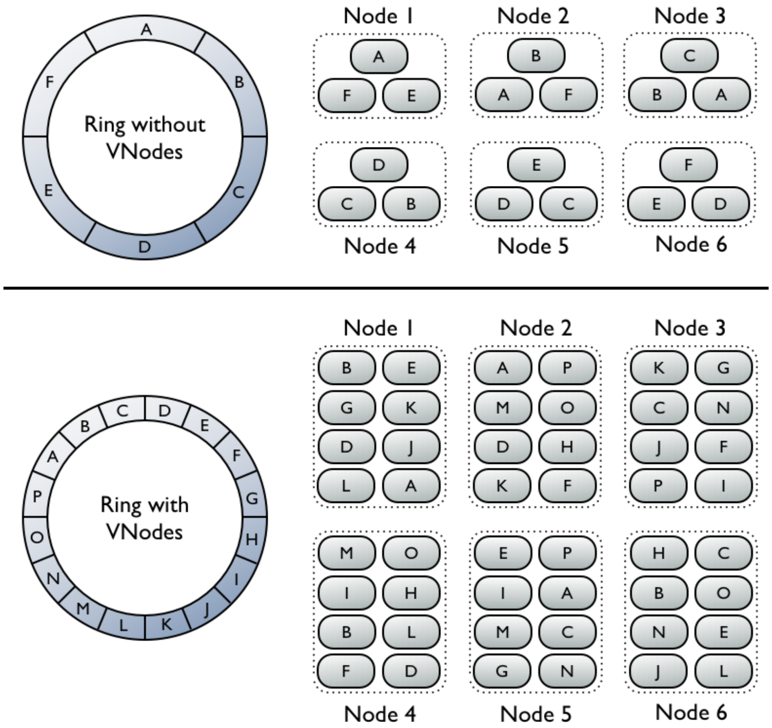

Figure 5: Virtual nodes in Cassandra

4 Exercises

4.1 Setting up the Environment

The topology consists of three machines (Figure 6), based on the same VMware virtual machine image. The virtual image contains the following software components:

- Xubuntu 16.04 (64-bit)
- Apache Cassandra 3.10
- ClusterSSH
- tmux

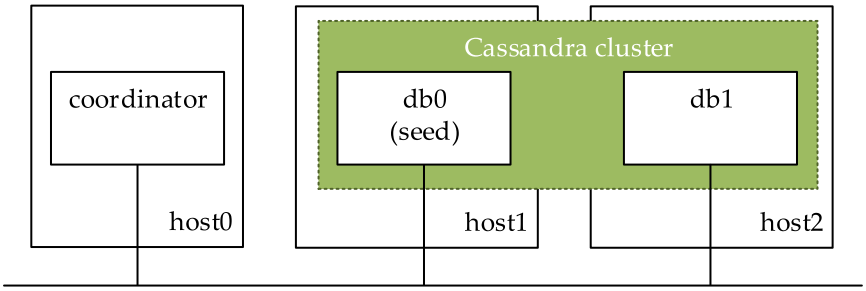

Figure 6: Topology

4.1.1 Hostnames

Set hostnames on the database nodes Because the machines are based on the same image, they have the same hostname (`cassandra-vm`). To distinguish between them, change the hostnames in the `/etc/hostname` file to different ones, e.g. `coordinator`, `db0`, `db1`. Also change the hostname associated with the 127.0.0.1 IP address in the `/etc/hosts` file as well[4].

Set hostnames on the coordinator To allow easier access (e.g. when using `ssh`), add the hostnames to the *coordinator's* `/etc/hosts` file.

```
10.0.0.100 db0
10.0.0.101 db1
```

Listing 1: `hosts` file

This way, we connect to all database servers using Cluster SSH:

```
cssh db0 db1
```

[4] This is due to a quirk in Debian-based Linux distributions. For details, visit `http://qref.sourceforge.net/quick/ch-gateway.en.html#s-net-dns`.

4.1.2 Running Cassandra on a Single Node

In the provided virtual machines, Cassandra is already set up to run in a single node. This means that the appropriate directory for storing the log files (`logs/`) and the data (`data/`) are created and their permissions are set. To run Cassandra, open a terminal emulator, navigate to the Cassandra folder (`~/apache-cassandra-*/`) and run the following command:

```
bin/cassandra -f
```

The `-f` switch forces Cassandra to run in foreground mode.

4.1.3 Restarting Cassandra

If Cassandra runs in foreground mode, simply send the Ctrl+C interrupt signal to kill the process. Else, send a termination signal with the following command:

```
pkill -f cassandra
```

To delete all previous data and restart the server, issue the following command:

```
rm -rf data/*; bin/cassandra -f
```

4.1.4 Running Cassandra on a Cluster of Nodes

The next step is to configure the Cassandra cluster. The main configuration file is stored in the `conf/cassandra.yaml` file. Follow the documentation [6] and configure Cassandra to run in a two node cluster.

4.2 Data Modeling

Create a database that stores road cars. Cars have a *manufacturer*, a *type*. Each car has a *maximum performance* and a *maximum torque* value. Table 1 shows an example.

manufacturer	type	maximum performance	maximum torque
Ford	Focus	100	170
Mercedes	E class	184	270

Table 1: Column family for storing cars

4.3 Replication and Consistency

To test Cassandra's replication schema and consistency models, we define a few steps. The task is to follow these steps, observe the cluster's behaviour and try to explain it.

It is important to note that some of the steps may throw error messages as their expected behaviour. In these cases, try to determine the reason of the error message and proceed to the next step.

Between the steps, you should delete all previous data (from the `/var/lib/cassandra/` directory) and restart the server.

4.3.1 Network Partition without Replication

1. Create a keyspace with the *replication factor* set to 1.
2. Create a column family and fill it with data.
3. Query the data from both `db0` and `db1`.
4. Disconnect `db0` from the network. The easiest way to do this is to disable the network interface in VMware.
5. Query the data from `db1`.
6. Modify the data from `db1`, e.g. add new rows to the column family.
7. Reconnect `db0` and query the data from `db0`.

4.3.2 Network Partition with Replication and Weak Consistency

1. Create a keyspace with the *replication factor* set to **2**.
2. Create a column family and fill it with data.
3. Query the data from both `db0` and `db1`.
4. Disconnect `db0` from the network.
5. Query the data from `db1`.
6. Modify the data from `db1`, e.g. add new rows to the column family.
7. Disconnect `db1` from the network.
8. Reconnect `db0` from the network.
9. Also query and modify from `db0`. Modify the same rows as previously so it will *conflict* with the previous write operations.
10. Reconnect `db1` to the network.
11. Query the data from both `db0` and `db1`.

4.3.3 Network Partition with Replication and Quorum Consistency

1. Create a keyspace with the *replication factor* set to **2**.
2. Create a column family, fill it with data, query it and disconnect `db0`.
3. On `db1`, set the consistency to **`QUORUM`**.
4. Query and modify the data.

4.4 Advanced Data Modeling

Cars have different powertrains (e.g. hybrid systems [17]). Each type can be described with different parameters:

- Internal combustion engine: fuel type, displacement, maximum torque, maximum power
- Electric motor: maximum torque, maximum power
- Both: all of the above and the combined maximum torque and power values

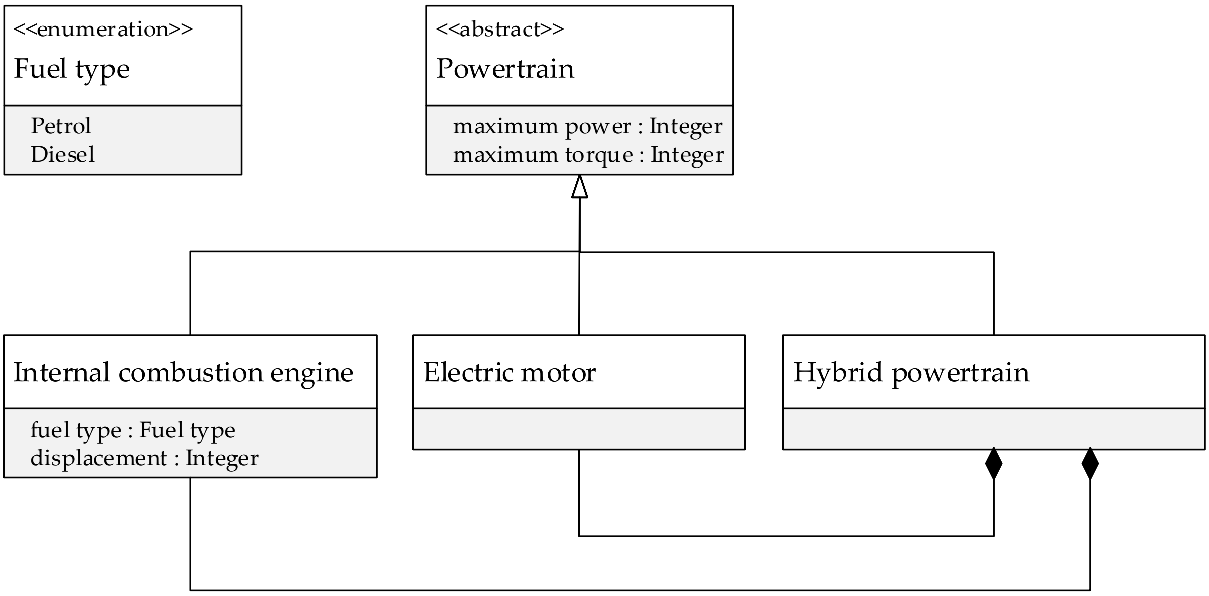

Figure 7: UML class diagram showing different powertrain types

The class hierarchy for different powertrain types are shown on Figure 7.

1. Extend the `cars` column family to store the powertrain of each car.
2. Write a query that collects the cars with an internal combustion engine.
3. Write a query that collects the cars with an internal combustion engine or an electric motor.

5 Additional Exercises

Cars have different transmissions systems. The main categories are the following:

- Automatic

  - fixed ratios: list of gear ratios, type (e.g. semi-automatic, dual-clutch, etc.)
  - continuously variable (CVT): gear ratio range

- Manual: list of gear ratios

- No gearbox: gear ratio

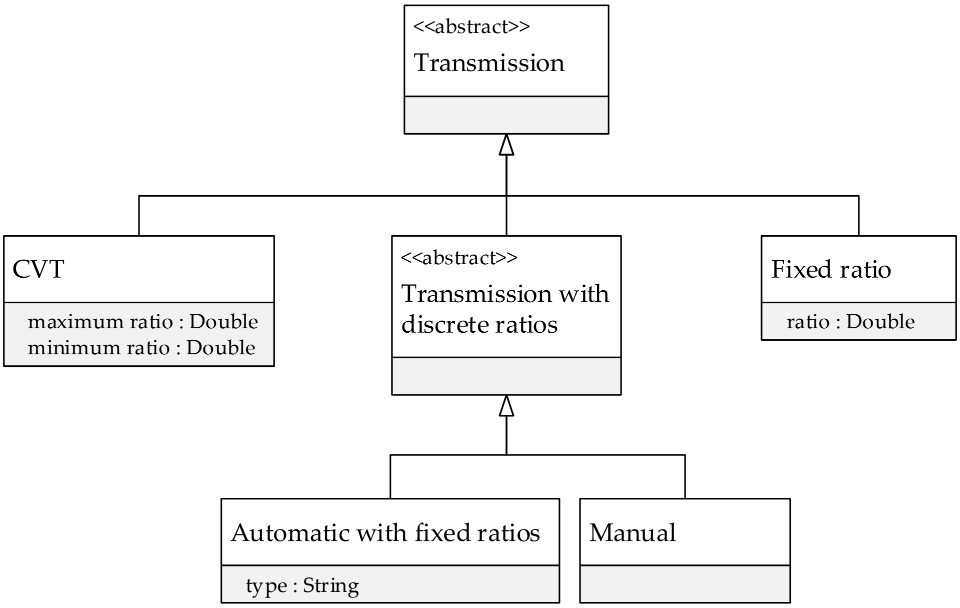

Figure 8: UML class diagram showing different transmission types

Figure 8 shows the class hierarchy for different transmission types.

Extend the `cars` column family to store the transmission for each car. To store the list of gear ratios, use CQL's `list` collection type.

6 Tips

- Set the listen address to 0.0.0.0 and the broadcast to the IP address of the machine.

- `ssh-keygen -t rsa`, `ssh-copy-id`

- VM: Bridged. To get a new IP address (in case of collision), change the adapter to something else (e.g. NAT) and change it back to Bridged.

  (Fortunately, Disconnect & Reconnect does not assign a new IP address.)

- Ctrl + Alt + T launches a terminal emulator application. You can open new tabs with the Ctrl + ⇧ + T hotkey. Ctrl + D sends and end-of-transmission character, which closes the terminal.

- Use `ifconfig` to determine a machine's IP address.

- The most important hotkeys for `mcedit` are the following:

  - F7 – search
  - ⇧+F7 – find next
  - Ctrl+0 – exit

- For `nano`, use Ctrl+W for search.

- If you use `vim`, you should already know the hotkeys ;-).

- Use the `watch` command to repeatedly run a tool, e.g.
  `watch -n 1 bin/nodetool status`

- To check if Cassandra is running in the background, run:
  `ps aux | grep cassandr[a]`
  To repeat the check later, simply run `!ps`.

- Useful Bash hotkeys: Ctrl+A jumps back to the start of the line, Ctrl+U deletes the current line.

References

[1] Virtual nodes in Cassandra 1.2. `http://www.datastax.com/dev/blog/virtual-nodes-in-cassandra-1-2`, December 2012.

[2] Apache Cassandra. `http://cassandra.apache.org/`, October 2013.

[3] Apache Thrift. `http://thrift.apache.org/`, October 2013.

[4] NoSQL Databases. `http://nosql-database.org/`, October 2013.

[5] Planet Cassandra – Companies. `http://planetcassandra.org/Company/ViewCompany`, October 2013.

[6] Apache Cassandra 3.9 documentation. `http://cassandra.apache.org/doc/3.9/`, March 2017.

[7] Nick Bailey. Introduction to Cassandra architecture. `https://www.slideshare.net/nickmbailey/introduction-to-cassandra-architecture`, June 2016.

[8] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[9] Christof Strauch. NoSQL Databases. `http://www.christof-strauch.de/nosqldbs.pdf`, October 2013.

[10] DataStax. Java Driver for Apache Cassandra. `https://github.com/datastax/java-driver`, October 2013.

[11] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[12] impetus-opensource. Kundera. `https://github.com/impetus-opensource/Kundera`, October 2013.

[13] Adam Jacobs. The pathologies of big data. *Commun. ACM*, 52(8):36–44, August 2009.

[14] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[15] Majzik István. Többpéldányos adatkezelés. `http://www.inf.mit.bme.hu/sites/default/files/materials/category/kateg%C3%B3ria/oktat%C3%A1s/msc-t%C3%A1rgyak/szolg%C3%A1ltat%C3%A1sbiztons%C3%A1gra-tervez%C3%A9s/13/SZBT-2013_EA05_tobbpeldanyos_adatkezeles.pdf`.

[16] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.

[17] Wikipedia. Hybrid vehicle drivetrain. `http://en.wikipedia.org/wiki/Hybrid_vehicle_drivetrain`, October 2013.