NoSQL – An Introduction

Introduction

- NoSQL is an approach to databases that represents a shift away from traditional relational database management systems (RDBMS).

Relational Databases & their Challenges

- RDBMS has its own set of problems when applied to massive amounts of data.
- The problems relate to efficient processing, effective parallelization, scalability, and costs.

ACID

- ACID (Atomicity, Consistency, Isolation, Durability) is very important for concurrency, safe sharing of data by transactions.
  - Atomicity
    - All of the operations in the transaction will complete, or none will.
    - No partial transaction is allowed
  - Consistency
    - Transactions never observe or result in inconsistent data.
  - Isolation
    - The transaction will behave as if it is the only transaction being performed upon the database though many transactions are running
  - Durability
    - Upon completion of the transaction, the effect of transaction will be permanent in nature

Features of relational databases make them "challenging" for certain problems

- **Fixed schemas**
  - The schemas must be defined ahead of time, changes are difficult.
  - Assumption
    - Data is dense and is largely uniform
  - **Solution**
    - Get rid of the schemas!
    - Is it really necessary to do design first?
    - Allow non-uniform data!
- **Complicated queries**
  - SQL is declarative and powerful but may be very tedious.
  - **Solution**
    - Simple query mechanisms.
- **Cost**

- ♠ Cost is high specially if amount of data is huge
- ♠ **Solution**
  - ϖ Open source???????

δ **Scalability**

- ♠ Relational databases may not scale sufficiently to handle high data and query loads or this scalability comes with a very high cost.
- ♠ **Solution**
  - ϖ Scalability may be kept as high priority and may be inbuilt
  - ϖ Users can scale a relational database by running it on a more powerful and expensive computer.
  - ϖ To scale beyond a certain point, it must be distributed across multiple servers.
  - ϖ Relational databases don't work easily in a distributed manner because joining their tables across a distributed system is difficult,
  - ϖ Relational databases aren't designed to function with data partitioning, so distributing their functionality is a tedious task.

Solution to the challenges of RDBMS

δ There need to be a database which provides

- ♠ Possibility to store unstructured data – schema-less
- ♠ Less complicated language than SQL
- ♠ Partitioning to support distributed system
- ♠ High performance
- ♠ Reliability through replication
- ♠ Simple queries
- ♠ Open source!

Motivation of NoSQL

δ Four interrelated megatrends

- ♠ Big Users
- ♠ Internet of Things,
- ♠ Big Data
- ♠ Cloud
  - ϖ are driving the adoption of NoSQL technology.

NoSQL vs. RDBMS

δ NoSQL

- ♠ Fast and simple, but has little to none structure to enforce constraints on data.

δ RDBMS

- ♠ Satisfies all ACID properties, keeping your data safe and clean.
- ♠ Performance goes down rapidly as traffic and data set size grow.

What does NoSQL stand for?

- NoSQL was more striking term than NoRDBMS or Norelational.
- A few others proposed that NoSQL is actually an acronym that expands to "Not Only SQL."
- NoSQL is used today as an umbrella term for all databases and data stores that don't follow the popular and well-established RDBMS principles and often relate to large data sets accessed and manipulated on a Web scale.
- NoSQL is an approach to databases that represents a shift away from traditional relational database management systems (RDBMS).
- NoSQL encompasses a wide variety of different database technologies that were developed in response to the demands presented in building modern applications:
  - Developers are working with applications that create massive volumes of new, rapidly changing data types — structured, semi-structured, unstructured and polymorphic data.
  - Long gone is the twelve-to-eighteen-month waterfall development cycle.
    - Now small teams work in agile sprints, iterating quickly and pushing code every week or two, some even multiple times every day.
  - Applications that once served a finite audience are now delivered as services that must be always-on, accessible from many different devices and scaled globally to millions of users.
  - Organizations are now turning to scale-out architectures using open source software, commodity servers and cloud computing instead of large monolithic servers and storage infrastructure.
  - Relational databases were not designed to cope with the scale and agility challenges that face modern applications, nor were they built to take advantage of the commodity storage and processing power available today.

History

- Such databases have existed since the late 1960s, but it did not obtain the "NoSQL" term until a surge of popularity in the early twenty-first century,
- It was triggered by the needs of Web 2.0 companies such s Facebook, Google and Amazon.com
- The term NoSQL was used by Carlo Strozzi in 1998 to name his lightweight, Strozzi NoSQL open-source relational database that did not expose the standard SQL interface, but was still relational.
- Strozzi suggests that it should have been called more appropriately 'NoREL'", referring to 'No Relational', since the current NoSQL movement departs from the relational model altogether

- Johan Oskarsson of Last.fm (**Last.fm** is a music website, founded in the United Kingdom in 2002) reintroduced the term *NoSQL* in early 2009 when he organized an event to discuss "open source distributed, non-relational databases".
- The name attempted to label the emergence of an increasing number of non-relational, distributed data stores, including open source clones of Google's BigTable / MapReduce and Amazon's Dynamo.
- Most of the early NoSQL systems did not attempt to provide atomicity, consistency, isolation and durability guarantees, contrary to the prevailing practice among relational database systems.
- Based on 2015 popularity rankings, the most popular NoSQL databases are MongoDB, Apache Cassandra, and Redis.

Mile Stones

- MultiValue databases created by Dick Pick at TRW in 1965.
- DBM is released by AT&T in 1979.
- Lotus Domino (server software) released in 1989.
- Carlo Strozzi used the term NoSQL in 1998 to name his lightweight, open-source relational database that did not expose the standard SQL interface.
- Graph database Neo4j is started in 2000.
- Google BigTable is started in 2004. Paper published in 2006.
- CouchDB is started in 2005.
- The research paper on Amazon Dynamo is released in 2007.
- The document database MongoDB was started in 2007 as a part of a open source cloud computing stack and first standalone release in 2009.
- Facebooks open sourced the Cassandra project in 2008.
- Project Voldemort (distributed key-value storage system) started in 2008.
- The term NoSQL was reintroduced in early 2009.
- HBase is a BigTable clone for the Hadoop project while Hypertable is another BigTable type database also from 2009.

Features of NoSQL

- Not Only SQL" or "Not Relational".
  - Handle big data (huge amount)
  - Scale horizontally "simple operations"
  - Replicate/distribute data over many servers
  - Simple call level interface (contrast with SQL)
  - Weaker concurrency model than ACID
  - Flexible schema

- NoSQL databases are useful for several problems not well-suited for relational databases with some typical features
  - Variable data: semi-structured, evolving, or has no schema
  - Massive data: terabytes or petabytes of data from applications like web analysis, sensors, social graphs
  - Parallelism: large data requires architectures to handle massive parallelism, scalability, and reliability
  - Simpler queries: may not need full SQL expressiveness
  - Relaxed consistency: more tolerant of errors, delays, or inconsistent results ("eventual consistency")
  - Easier/cheaper: less initial cost to get started

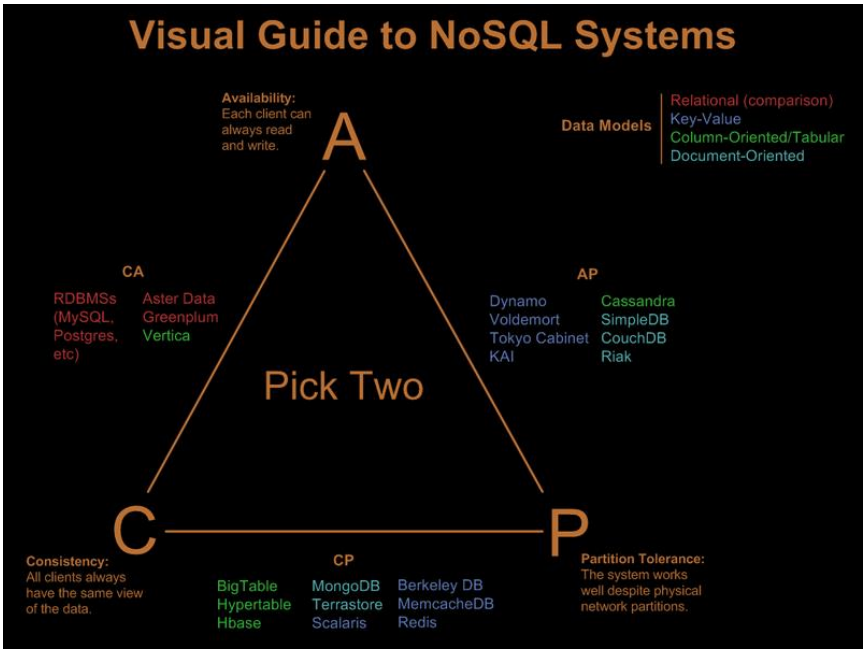
Big Data

- In a 2001 research report and related lectures, META Group (now Gartner) analyst Doug Laney defined data growth challenges and opportunities as being three-dimensional, i.e. increasing volume (amount of data), velocity (speed of data in and out), and variety (range of data types and sources).
- Gartner, and now much of the industry, continue to use this "3Vs" model for describing big data.
- In 2012, Gartner updated the definition as follows: "Big data is high volume, high velocity, and/or high variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization."
- The 3Vs have been expanded to other complementary characteristics of big data:
  - Volume: Large amount of data
  - Velocity: high frequency of data generation
  - Variety: different kind of text, images, audio, video etc.
- Machine Learning: explores the study and construction of algorithms that can learn from and make predictions on data.
- Digital footprint: big data is often a cost-free by-product of digital interaction

CAP Theorem

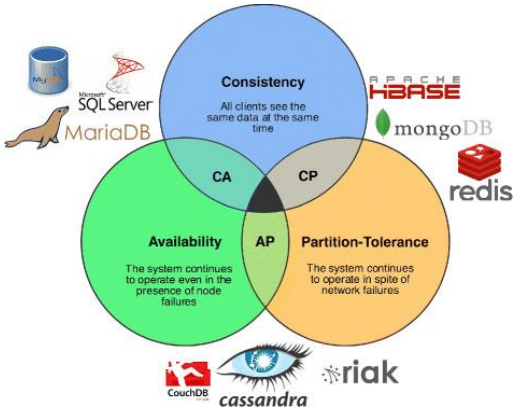
- The CAP Theorem (proposed by Eric Brewer) states that there are three properties of a data system
  - Consistency
    - each client always has the same view of the data
  - Availability
    - all clients can always read and write
  - Partitions
    - the system works well across physical network partitions

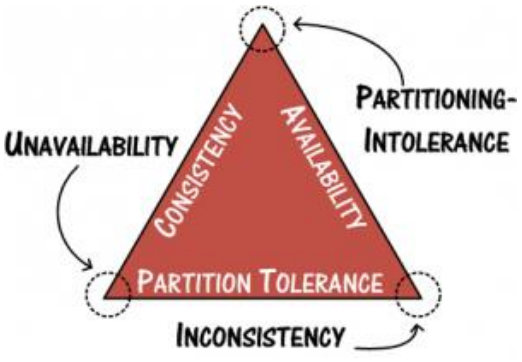
NoSQl in a snapshot



CAP Theorem – NoSQL

- δ **Consistency** (all nodes see the same data at the same time)
- δ **Availability** (a guarantee that every request receives a response about whether it was successful or failed)
- δ **Partition tolerance** (the system continues to operate despite arbitrary partitioning due to network failures)
  - ♠ At most two of the three properties at a time can be available in any DBMS.
  - ♠ Since scaling out requires partitioning, many NoSQL systems sacrifice consistency for availability. (or availability for consistency)
- δ One of the primary goals of NoSQL systems is to bolster horizontal scalability.
- δ To scale horizontally, you need strong network partition tolerance which requires giving up either consistency or availability.
- δ NoSQL systems typically accomplish this by relaxing relational abilities and/or loosening transactional semantics.





BASE

- **B**asically **A**vailable **S**oft state **E**ventual consistency
- Updates will eventually propagate through all nodes in the system which will then be consistent.
- Basically Available
  - System does guarantee availability in terms of CAP Theorem
- Soft State
  - State of the system may change over time even without input.
  - This is because of the eventual consistency model.
- Eventual Consistency
  - System will become consistent over time given that the system does not receive input during that time.

ACID Vs. BASE

- Leads to abandonment of ACID requirements, moving toward BASE
  - Atomicity
    - Implementing a transaction fully or not at all
  - Consistency
    - Guaranteeing integrity of the database
  - Isolation
    - Transactions are carried out independent of simultaneously running transactions
  - Durability
    - After commitment of the transaction, changes are saved permanently
- Changed to
  - **Basically available**
    - System is generally available, availability is not guaranteed though
  - **Soft state**
    - System state can change over time even without data entry
  - **Eventual Consistency**
    - Temporary inconsistency of the database is accepted, consistency is restored with a delay
  - When no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent

Types of DMS

- In addition to CAP configurations, another significant way data management systems vary is by the data model they use
  - Relational

- Key-value
- Column-oriented
- Document-oriented

- **Relational** systems
  - The databases we've been using for a while now.
  - RDBMSs and systems that support ACIDity and joins are considered relational.
- **Key-value** systems
  - Support get, put, and delete operations based on a primary key.
- **Column-oriented** systems
  - Use tables but have no joins (joins must be handled within your application).
  - Obviously, they store data by column as opposed to traditional row-oriented databases.
    - This makes aggregations much easier.
- **Document-oriented** systems
  - Store structured "documents" such as JSON or XML but have no joins (joins must be handled within your application).
  - It's very easy to map data from object-oriented software to these systems.

Types of NoSQL

- Column Stores
- Key Stores
- Document Stores
- Graph databases
- Multi model

Column Stores

- Represent data in columns rather than rows.
- Cassandra, Hbase

```
10:001,12:002,11:003,22:004;
Smith:001,Jones:002,Johnson:003,Jones:004;
Joe:001,Mary:002,Cathy:003,Bob:004;
40000:001,50000:002,44000:003,55000:004;
```

Key-value stores

- These databases pair keys to values.
- CouchDB, Dynamo, Oracle NoSQL Database, OrientDB, Redis
- Example-1
  - "person0:firstname" = "Lekha"
  - "person1:firstname" = "Isha"
- Example-2

- ♠ "person0:account_type" = "Saving"
- ♠ "person0:balance" = "255443.871"
- ♠ "person1:account_type" = "Current"
- ♠ "person1:balance" = "16254.56"

Document stores

- δ Similar to key-value stores except value is a document in some form (e.g. JSON)
- δ MongoDB, Apache CouchDB, Couchbase, OrientDB, DocumentDB, Lotus Notes, MarkLogic

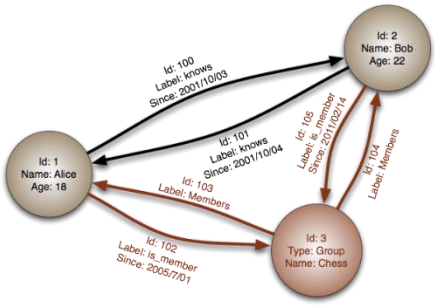
δ

```
Bson:
  \x16\x00\x00\x00              // total document size
  \x02                          // 0x02 = type String
  hello\x00                     // field name
  \x06\x00\x00\x00world\x00     // field value (size of value, value, null terminator)
  \x00                          // 0x00 = type EOO ('end of object')
```

Graph databases

- δ Represent data as graphs
- δ Neo4J, OrientDB, Allegro, InfiniteGraph

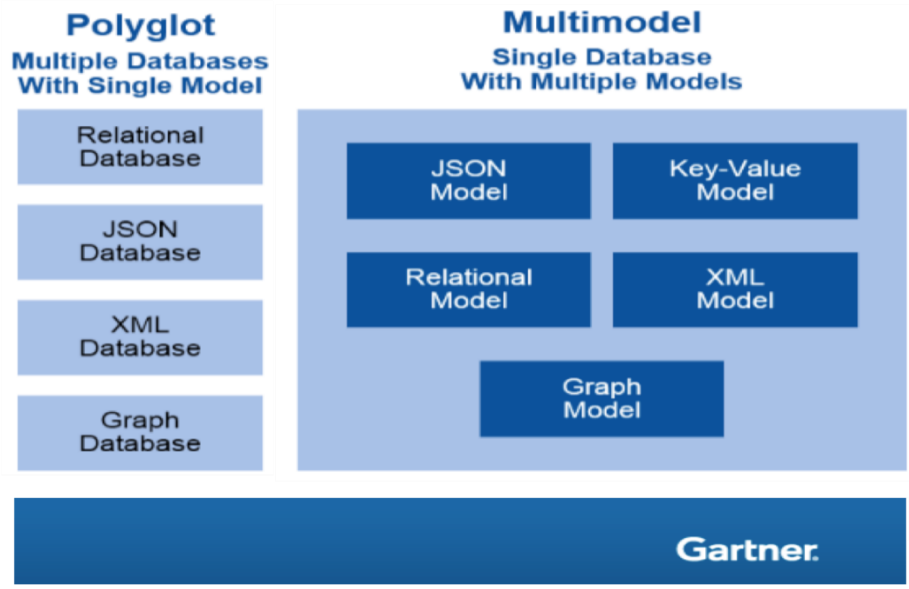
δ



Multi-model

- δ MarkLogic, OrientDB

δ



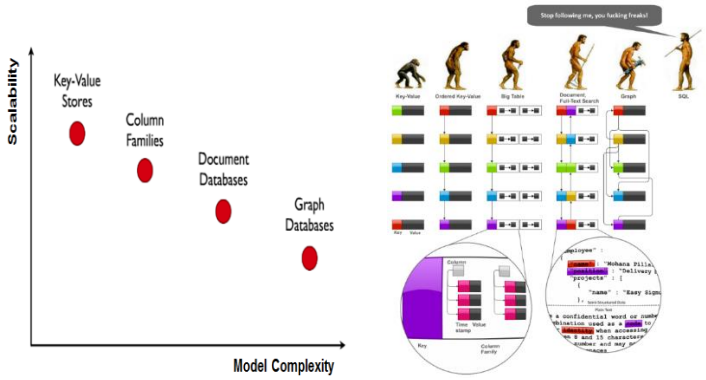
CAP Configuration and Systems

- **Consistent, Available (CA) Systems**
  - They have trouble with partitions and typically deal with it with replication.
  - Examples of CA systems include
    - Traditional RDBMSs like Postgres, MySQL, etc (relational)
  - Vertica (column-oriented)
  - Aster Data (relational)
  - Greenplum (relational)
- **Consistent, Partition-Tolerant (CP) Systems**
  - They have trouble with availability while keeping data consistent across partitioned nodes.
  - Examples
    - BigTable, Hypertable, HBase (column-oriented/tabular)
    - MongoDB, Terrastore (document-oriented)
    - Redis, Scalaris, MemcacheDB, Berkley DB (key-value)
- **Available, Partition-Tolerant (AP) Systems**
  - Achieve "eventual consistency" through replication and verification.
  - Examples
    - Dynamo, Voldemort, Tokyo Cabinet, KAI (key-value)
    - Cassandra (column-oriented/tabular)
    - CouchDB, SimpleDB, Riak (document-oriented)

NoSQL vs. Relational Database

- Supports very simple query language.
  - Supports powerful query language.
- No fixed schema.
  - It has a fixed schema.
- It is only "eventually consistent".
  - Follows ACID (Atomicity, Consistency, Isolation, and Durability).
- Does not support transactions.
  - Supports transactions.

Data Modelling



- NoSQL data modeling often starts from the application-specific queries as opposed to relational modeling
  - Relational modeling is typically driven by the structure of available data.
    - The main design theme is "**What answers do I have?"**
  - NoSQL data modeling is typically driven by application-specific access patterns, i.e. the types of queries to be supported.
    - The main design theme is **"What questions do I have?"**
- NoSQL data modeling often requires a deeper understanding of data structures and algorithms than relational database modeling does.
- Data duplication and denormalization are first-class citizens.
- Relational databases are not very convenient for hierarchical or graph-like data modeling and processing.
  - Graph databases are obviously a perfect solution for this area, but actually most of NoSQL solutions are surprisingly strong for such problems.

Basic Principles behind Data Modeling

- **Denormalization**
  - Denormalization can be defined as the copying of the same data into multiple documents or tables in order to simplify/optimize query processing or to fit the user's data into a particular data model.
  - It is helpful for the following trade-offs
    - Query data volume or IO per query VS total data volume
      - Using denormalization one can group all data that is needed to process a query in one place.
      - This often means that for different query flows the same data will be accessed in different combinations.
      - Hence there is a need to duplicate data, which increases total data volume.
    - Processing complexity VS total data volume.

- Modeling-time normalization and consequent query-time joins obviously increase complexity of the query processor, especially in distributed systems.
- Denormalization allow one to store data in a query-friendly structure to simplify query processing.

- **Applicability**
  - Key-Value Stores, Document Databases, BigTable-style Databases
- **Aggregates**
  - All major genres of NoSQL provide soft schema capabilities in one way or another
    - Key-Value Stores and Graph Databases typically do not place constraints on values, so values can be comprised of arbitrary format.
      - It is also possible to vary a number of records for one business entity by using composite keys.
      - For example, a user account can be modeled as a set of entries with composite keys like UserID_name, UserID_email, UserID_messages and so on.
        - If a user has no email or messages then a corresponding entry is not recorded.
  - BigTable models support soft schema via a variable set of columns within a *column family* and a variable number of *versions* for one *cell*.
  - Document databases are inherently schema-less, although some of them allow one to validate incoming data using a user-defined schema.

Cassandra

- Apache Cassandra is an open source, distributed and decentralized/distributed storage system (database), for managing very large amounts of structured data spread out across the world.
- It provides highly available service with no single point of failure.
- It is a column-oriented database.

CouchDB

- CouchDB is an open source database developed by Apache software foundation.
- The focus is on the ease of use, embracing the web.
- It is a NoSQL document store database.
- It uses JSON, to store data (documents), java script as its query language to transform the documents, http protocol for API to access the documents, query the indices with the web browser.
- It is a multi-master application released in 2005 and it became an apache project in 2008.

HBase

- HBase is a distributed column-oriented database built on top of the Hadoop file system.
- It is an open-source project and is horizontally scalable.
- HBase is a data model that is similar to Google's big table designed to provide quick random access to huge amounts of structured data.
- It leverages the fault tolerance provided by the Hadoop File System (HDFS).

Neo4j

- Neo4j is one of the popular Graph Databases and Cypher Query Language (CQL).
- Neo4j is written in Java Language.

OrientDB

- OrientDB is an Open Source NoSQL Database Management System.
- OrientDB is the first Multi-Model open source NoSQL DBMS that brings together the power of graphs and flexibility of documents into a scalable high-performance operational database.

MongoDB

- MongoDB is an open-source document database and leading NoSQL database.
- MongoDB is written in C++.
- MongoDB is a cross-platform, **document-oriented** database that provides, high performance, high availability, and easy scalability.
- MongoDB works on concept of collection and document.

Advantages of MongoDB

- **Schema less**
  - MongoDB is a document database in which one collection holds different documents.
  - Number of fields, content and size of the document can differ from one document to another.
- Structure of a single object is clear.
- No complex joins.
- Deep query-ability. MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.
- Tuning.
- **Ease of scale-out**
  - MongoDB is easy to scale.
- Conversion/mapping of application objects to database objects not needed.

- Uses internal memory for storing the (windowed) working set, enabling faster access of data.

Why Use MongoDB?

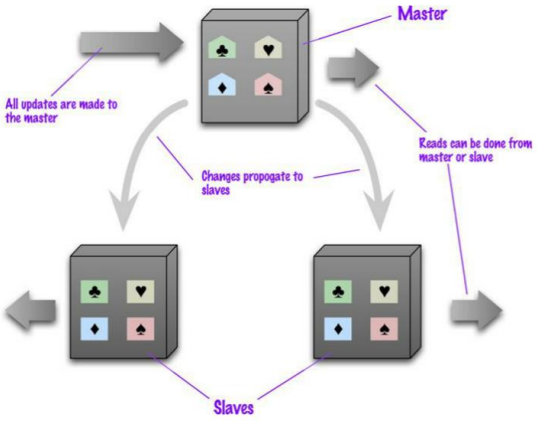
- Document Oriented Storage
  - Data is stored in the form of JSON style documents.
- Index on any attribute
- Replication and high availability
- Auto-sharding
- Rich queries
- Fast in-place updates
- Professional support by MongoDB

Replication

- Data replication is the concept of having data, within a system, be geo-distributed; preferably through a non-interactive, reliable process.
- It is of two kinds
  - Master-Slave
  - Peer-to-Peer

Master-Slave Replication

- Master
  - is the authoritative source for the data
  - is responsible for processing any updates to that data
  - can be appointed manually or automatically
- Slave
  - A replication process synchronizes the slaves with the master
  - After a failure of the master, a slave can be appointed as new master very quickly.



Master-Slave Replication – Pros

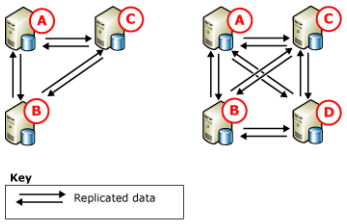
- δ More read requests
  - ♠ Add more slave nodes
  - ♠ Ensure that all read requests are routed to the slaves
- δ Should the master fail, the slaves can still handle read requests
- δ Good for datasets with a read-intensive dataset

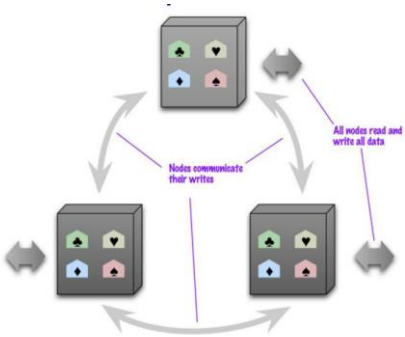
Master-Slave Replication – Cons

- δ The master is a bottleneck
  - ♠ Limited by its ability to process updates and to pass those updates on
  - ♠ Its failure does eliminate the ability to handle writes until
    - ϖ the master is restored or
    - ϖ a new master is appointed
- δ Inconsistency due to slow propagation of changes to the slaves
- δ Bad for data sets with heavy write traffic

Peer-to-Peer Replication

- δ All the replicas have equal weight, they can all accept writes.
- δ The loss of any of them doesn't prevent access to the data store.





Peer-to-Peer Replication – Pros

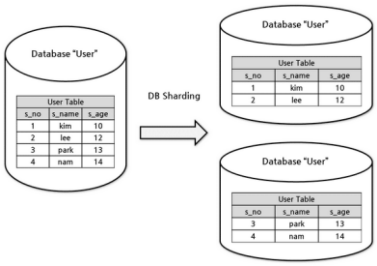
- δ Can ride over node failures without losing access to data.
- δ Can easily add nodes to improve the performance.

Peer-to-peer Replication – Cons

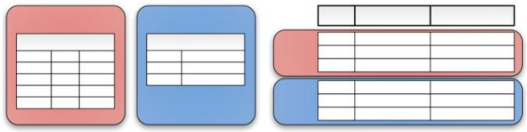
- δ Inconsistency!
  - ♠ Slow propagation of changes to copies on different nodes
    - ϖ Inconsistencies on read lead to problems but are relatively transient
  - ♠ Two people can update different copies of the same record stored on different nodes at the same time - a **write-write conflict**.
    - ϖ Inconsistent writes are forever.

Sharding

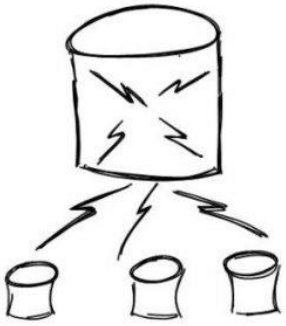
- Process of keeping different parts of the data onto different servers.
- Sharding is a type of database partitioning that separates very large databases the into smaller, faster, more easily managed parts called data shards.
- The word shard means a small part of a whole.
- Also refers to horizontal partitioning.
- Database Sharding can be simply defined as a "shared-nothing" partitioning scheme for large databases across a number of servers, enabling new levels of database performance and scalability achievable.



- The distinction of **horizontal** vs **vertical** comes from the traditional tabular view of a database.
- A database can be split
  - ♠ vertically—storing different tables & columns in a separate database,
  - ♠ horizontally—storing rows of a same table in multiple database nodes.



- Sharding provides a method for scalability across independent servers, each with their own CPU, memory and disk.
- Database Sharding is very straightforward: take a large database, and break it into a number of smaller databases across servers.



Sharding vs. Partitioning

- Vertical partitioning is very domain specific.
  - A logical split is drawn within the application data, storing them in different databases.
  - It is almost always implemented at the **application level**—a piece of code routing reads and writes to a designated database.
- Sharding splits a homogeneous type of data into multiple databases.
- Sharding can be implemented at either the application or **database level**.

Sharding – Advantages

- Improved scalability, growing in a near-linear fashion as more servers are added to the network.
- *Smaller databases are easier to manage.*
  - Production databases must be fully managed for regular backups, database optimization and other common tasks.
  - With a single large database these routine tasks can be very difficult to accomplish, if only in terms of the time window required for completion.
  - Routine table and index optimizations can stretch to hours or days, in some cases making regular maintenance infeasible.
  - By using the sharding approach, each individual "shard" can be maintained independently, providing a far more manageable scenario, performing such maintenance tasks in parallel.
- *Smaller databases are faster.*
  - The scalability of sharding is apparent, achieved through the distribution of processing across multiple shards and servers in the network.
  - What is less apparent is the fact that each individual shard database will outperform a single large database due to its smaller size.
  - By hosting each shard database on its own server, the ratio between memory and data on disk is greatly improved, thereby reducing disk I/O.
  - This results in less contention for resources, greater join performance, faster index searches, and fewer database locks.
- *Database Sharding can reduce costs.*
  - Most Database Sharding implementations take advantage of lower-cost open source databases, or can even take advantage of "workgroup" versions of commercial databases.
  - Additionally, sharding works well with commodity multi-core server hardware, far less expensive than high-end multi-CPU servers and expensive SANs.

- The overall reduction in cost due to savings in license fees, software maintenance and hardware investment is substantial, in some cases 70% or more when compared to other solutions.