Introduction to No SQL Databases

pm_jat @ daiict

No SQL Databases

- What is No SQL Databases?

- Before that, what is “SQL databases”?

SQL Databases

- SQL Databases are basically “**Relational Databases**”
- Relational databases are predominantly around since its proposal by EF Codd [1970]
- In recent decades, the enterprise computing saw many changes in terms of programming languages, architectures, platforms, and processes; however one thing **remained same was “relational databases“**, till the time “No SQL” appeared!
- “No SQL” emerged (typically since 2009) as a major challenger!

Relational “Reign”

- What has been great about Relational databases that they ruled for decades?
- Let us enumerate some benefits of SQL databases!

Relational “Reign”

The “Data Model”

1. Purely a mathematical model and hides all representational details like – records, data files, etc.
2. Operations are performed on “relations” while internally data are actually stored on disk files. Disk files remain transparent to database user.
3. Normalized data storage minimizing data redundancy that avoids various operational anomalies.
4. Facilitates data integrity by its ability to define various kind of database integrity constraints.
5. Schema is part of database and automatic enforcement of database constraints.

Relational "Reign"

Relational Systems:

1. Shared Access of Integrated data
2. Data Independence [Three Schema Architecture]
3. Programming Standardization [ANSI SQL, ODBC, JDBC, etc]
4. Concurrent Access of data and Transaction Processing support – provides ACID properties support
5. Query Optimization – DBMS generates most optimal physical query execution plan in terms of "file scan", "index scan", "hash-join", "hash-sort", etc.

Relational "Reign"

Relational Systems:

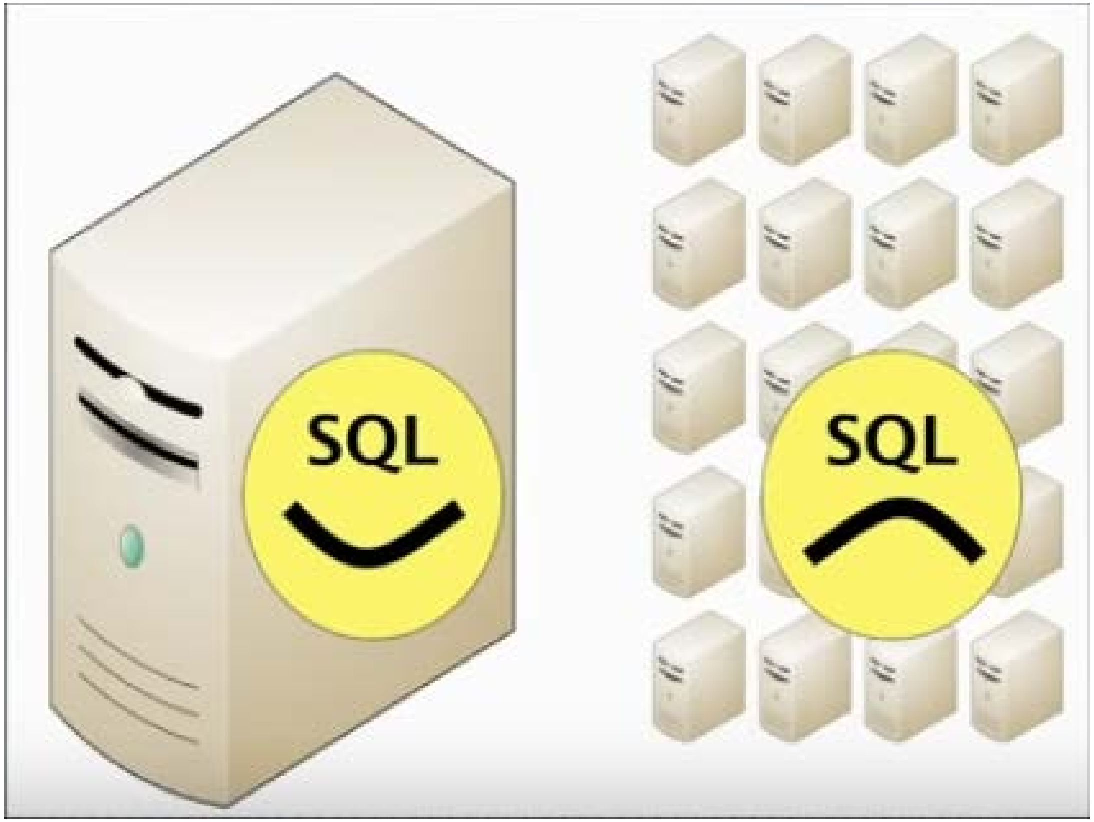
1. Shared Access of Integrated data
2. Data Independence [Three Schema Architecture]
   - Schema at one level is for users (easier to use for users), where as at conceptual level optimized for data redundancy, and at physical level optimized for efficient query execution.
   - Each schema can independently optimized for respective objectives
3. Programming Standardization [ANSI SQL, ODBC, JDBC, etc]
4. Concurrent Access of data – provides ACID properties support
5. Query Optimization – DBMS generates most optimal physical query execution plan in terms of "file scan", "index scan", "hash-join", "hash-sort", etc.

Why Relational data model is no more "universal" solution?

1. Can not scale at web scale as unable to run on "computing clusters"
2. Relational tables are not compatible with "in memory" program data structures
   - Objects and Tables do not match well
   - Martin Fowler names it as "impedance mismatch" problem

Relational can not scale at web scale

- Because relational can not run on computing clusters
- People tried and did not work!
- Alternative efforts were made, and
- Google Big Table, facebook cassandra, amazon dynanoDB, are result of that.
- No SQL take birth here!

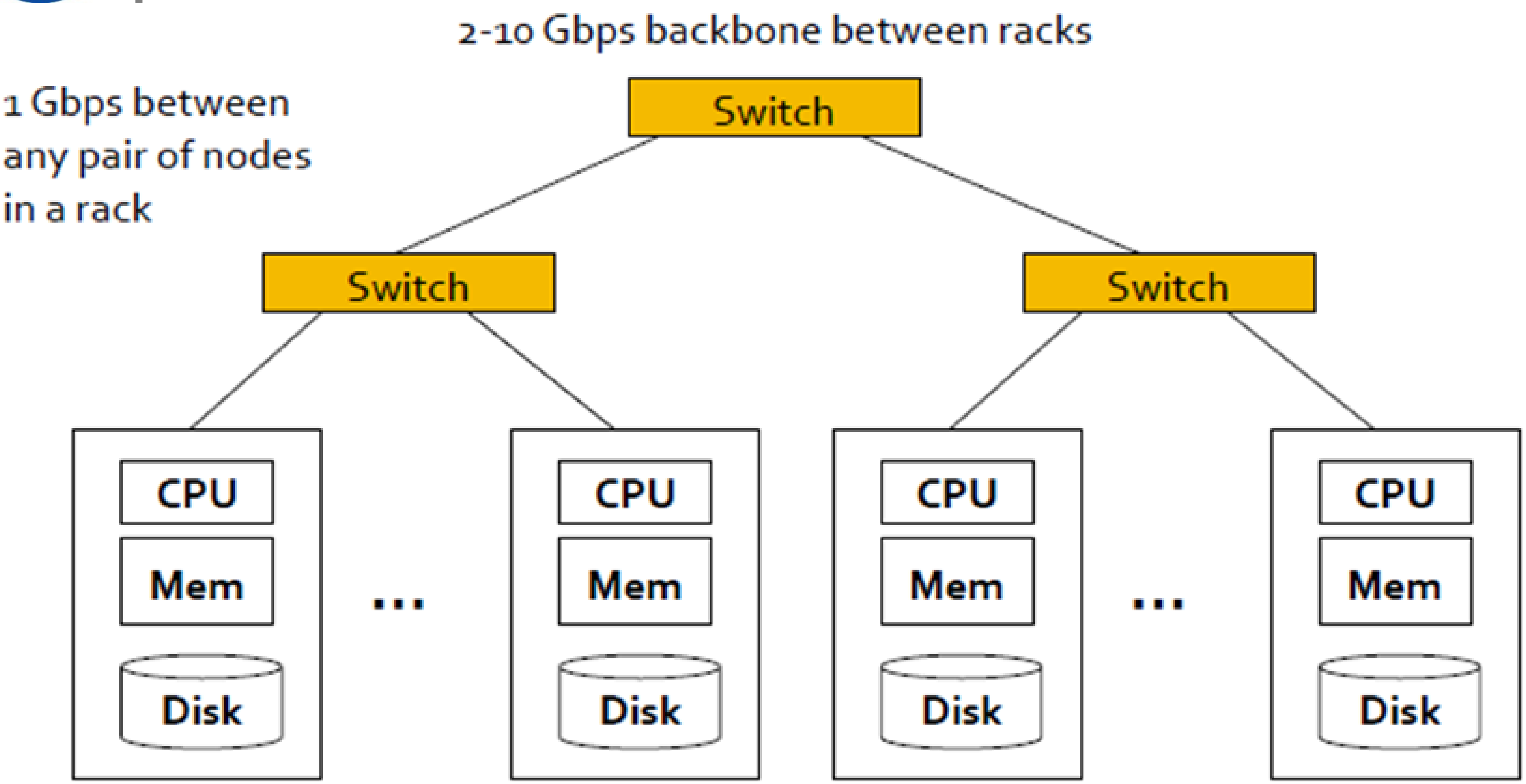


Cluster Computing

- Large number of systems work in parallel for computational tasks.
- Low cost Computers (termed as “commodity hardware” in map reduce) are connected through network and collaboratively work for a computational task in parallel.
- They work in “Nothing Shared” manner in terms of parallel computing terminology.
- Google was first to introduce a complete solution on clustered computing – **Google Distributed File System** and **Map Reduce Programming Abstraction** through a revolutionary article [5]

http://mmds.org/mmds/v2.1/ch02-mapreduce.pdf

Cluster Computing Architecture



Each rack contains 16-64 nodes

http://mmds.org/mmds/v2.1/ch02-mapreduce.pdf

In 2011 it was guestimated that Google had 1M machines, http://bit.ly/Shh0RO

Relational databases are hard to scale

- Relational databases require more powerful systems to deal with increased load
  - Upgrading hardware in terms of “power” is called “**Vertical Expansion**” where as adding “more computers” with same power is called as “**Horizontal Expansion**”.
  - Cluster Computing works on the principle of horizontal expansion.
- When relational system expand, they **require data migration**
- Vertical expansion will always have **some limit to grow**!
- Impossible to have “**elastic**” systems (seamless growing and shrinking of system requirements)

Elasticity

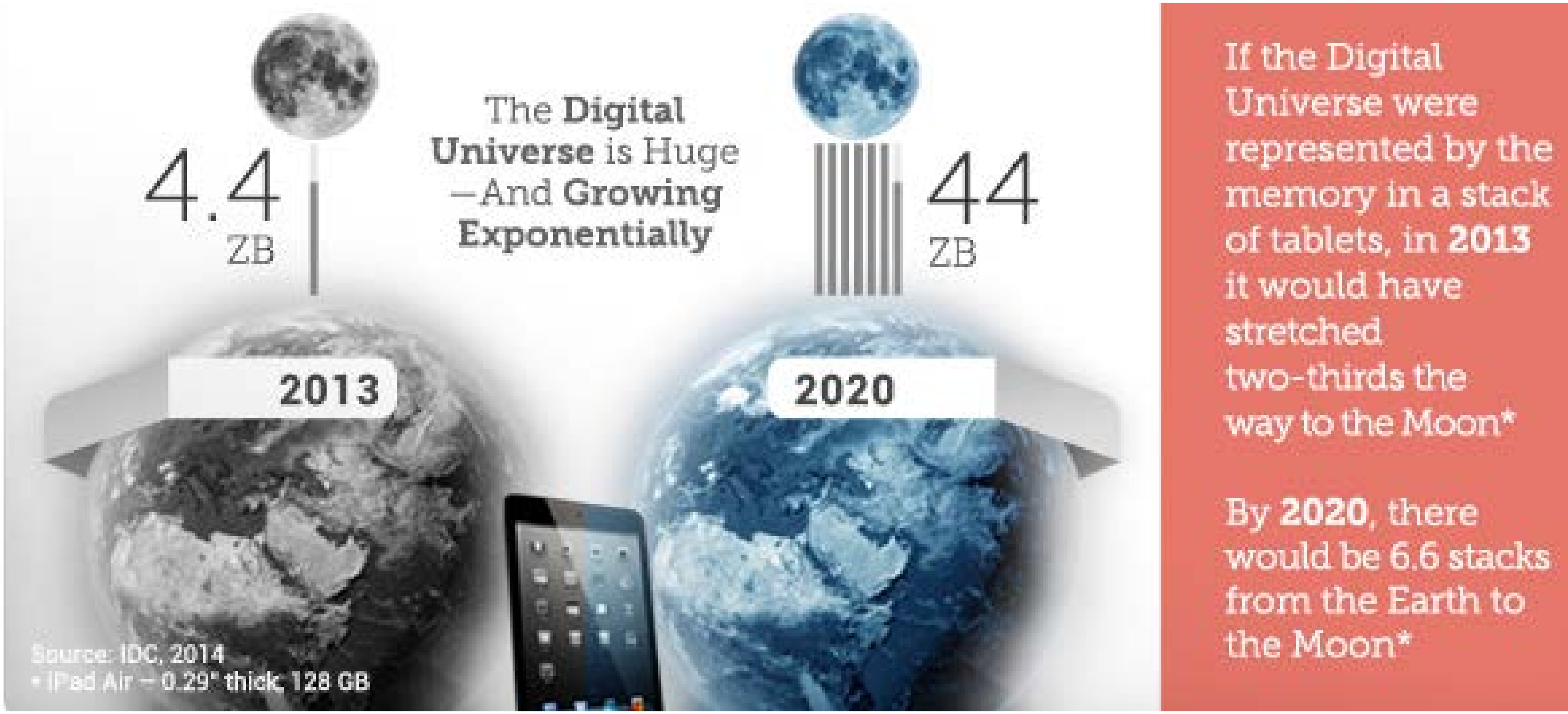
- Definition from [3]

  Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible

RDBs are not designed for Scale[1]

Some snapshots from an related article, should augment the argument

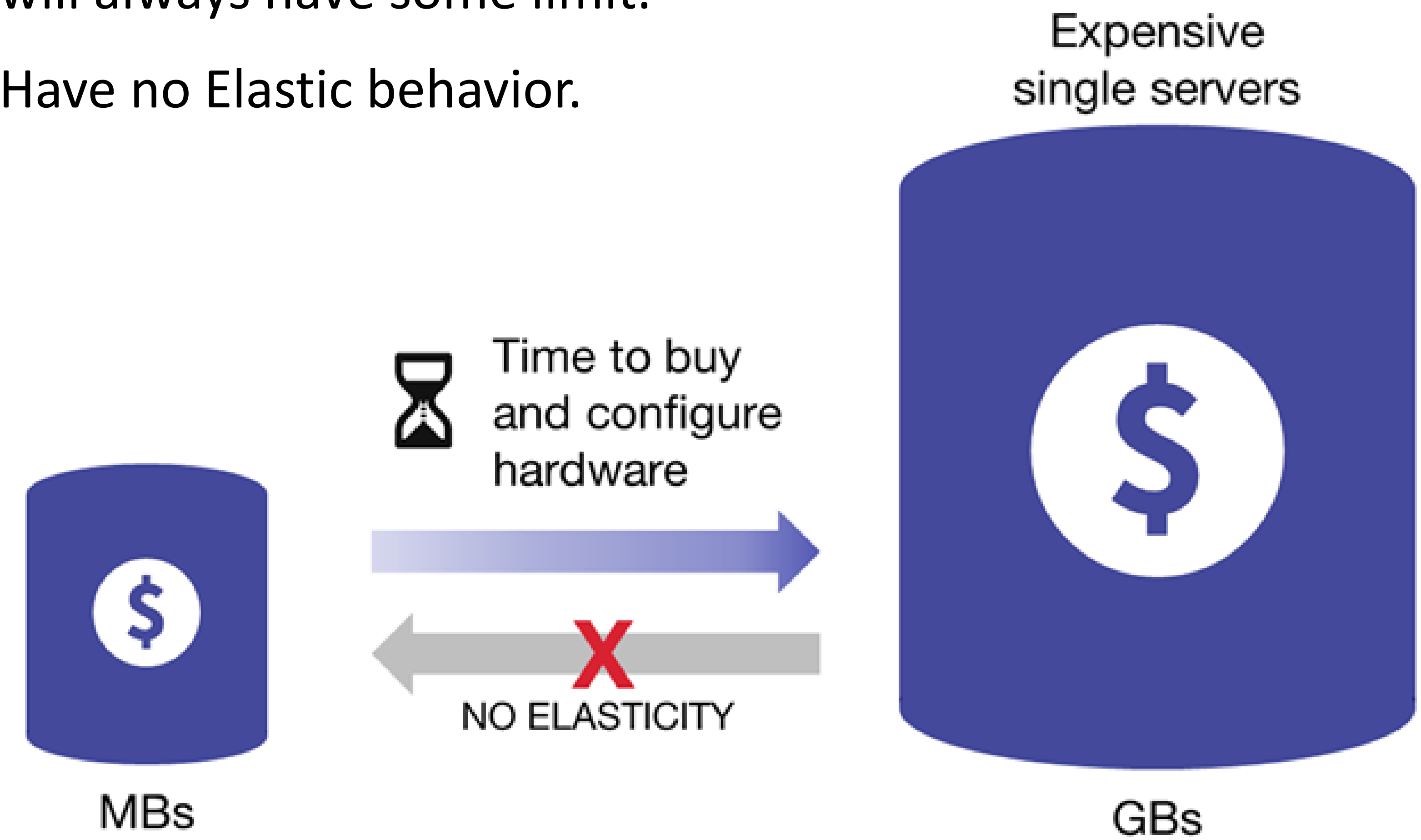
- Data to be stored and processed are seeing exponential growth[2] from 4.4ZB in 2013 to 44ZB in 2020



[1] https://www.marklogic.com/blog/relational-databases-scale/
[2] https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm

RDBs are not designed for Scale[1]

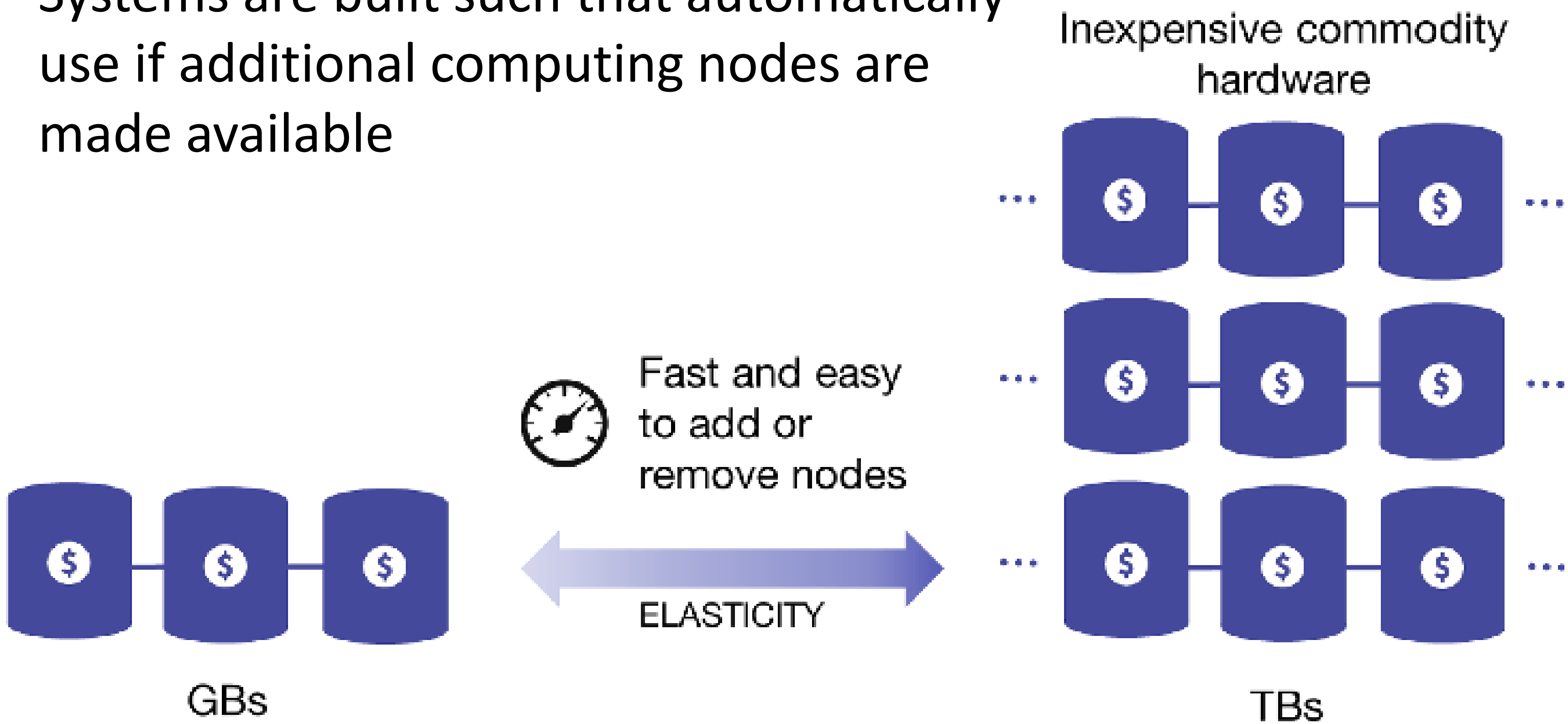
- Horizontal Scaling is expensive, require data migration, and will always have some limit.
- Have no Elastic behavior.



[1] https://www.marklogic.com/blog/relational-databases-scale/

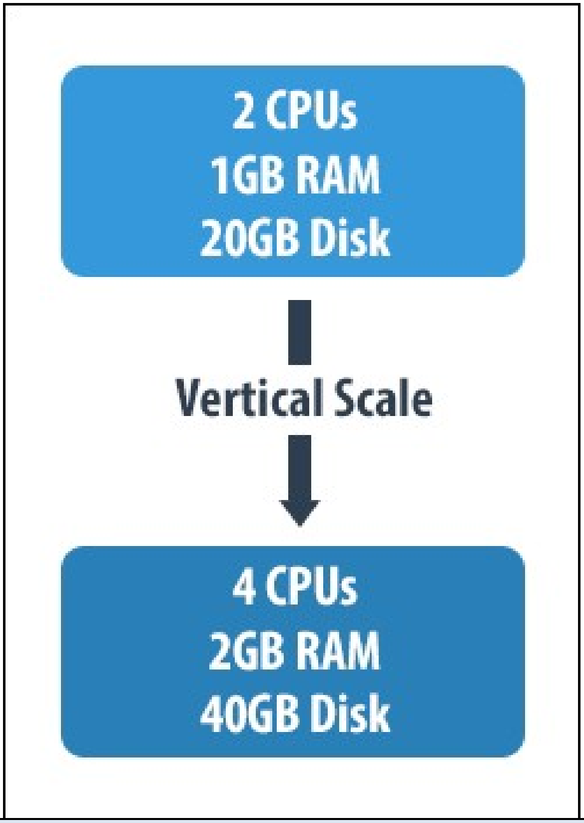
No SQL databases are designed for Scale[1]

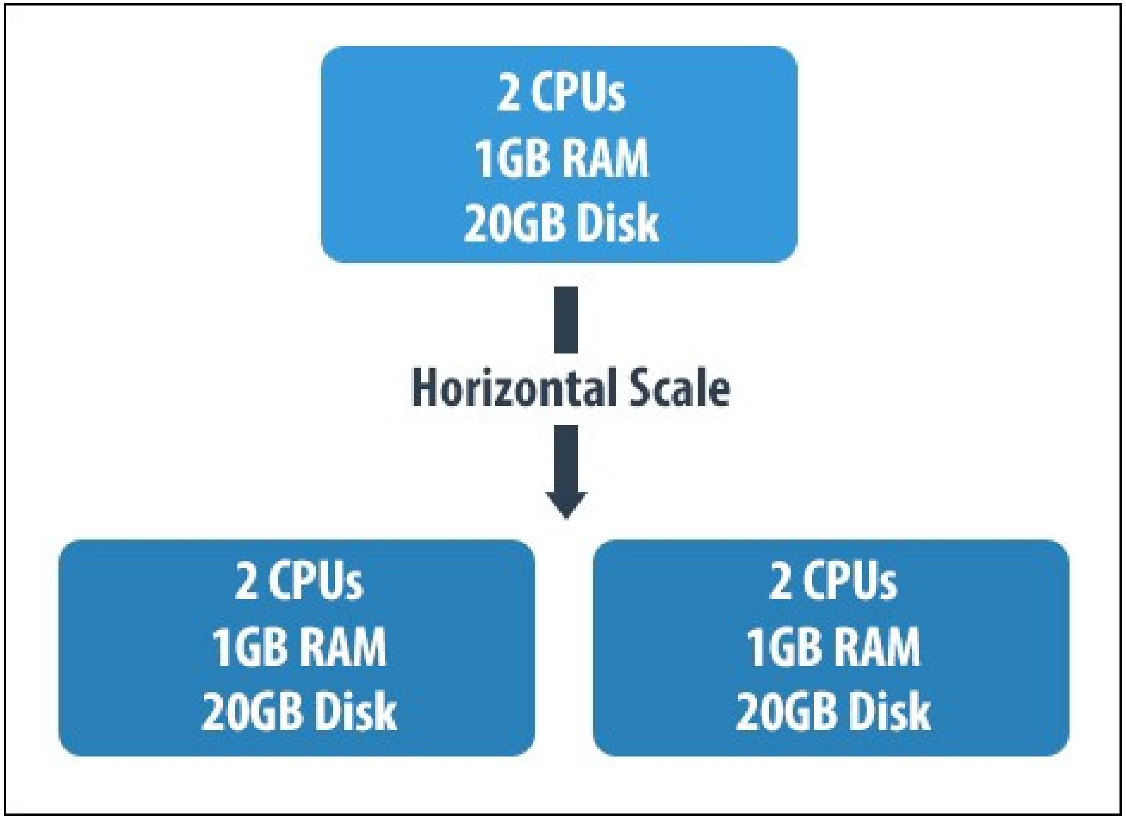
- Diagram below, taken from MarkLogic Server, captures the concept
- Systems are built such that automatically use if additional computing nodes are made available



[1] https://www.marklogic.com/blog/relational-databases-scale/

Vertical vs Horizontal Scale





https://hackernoon.com/database-scaling-horizontal-and-vertical-scaling-85edd2fd9944

Why Relational is hard on clusters!

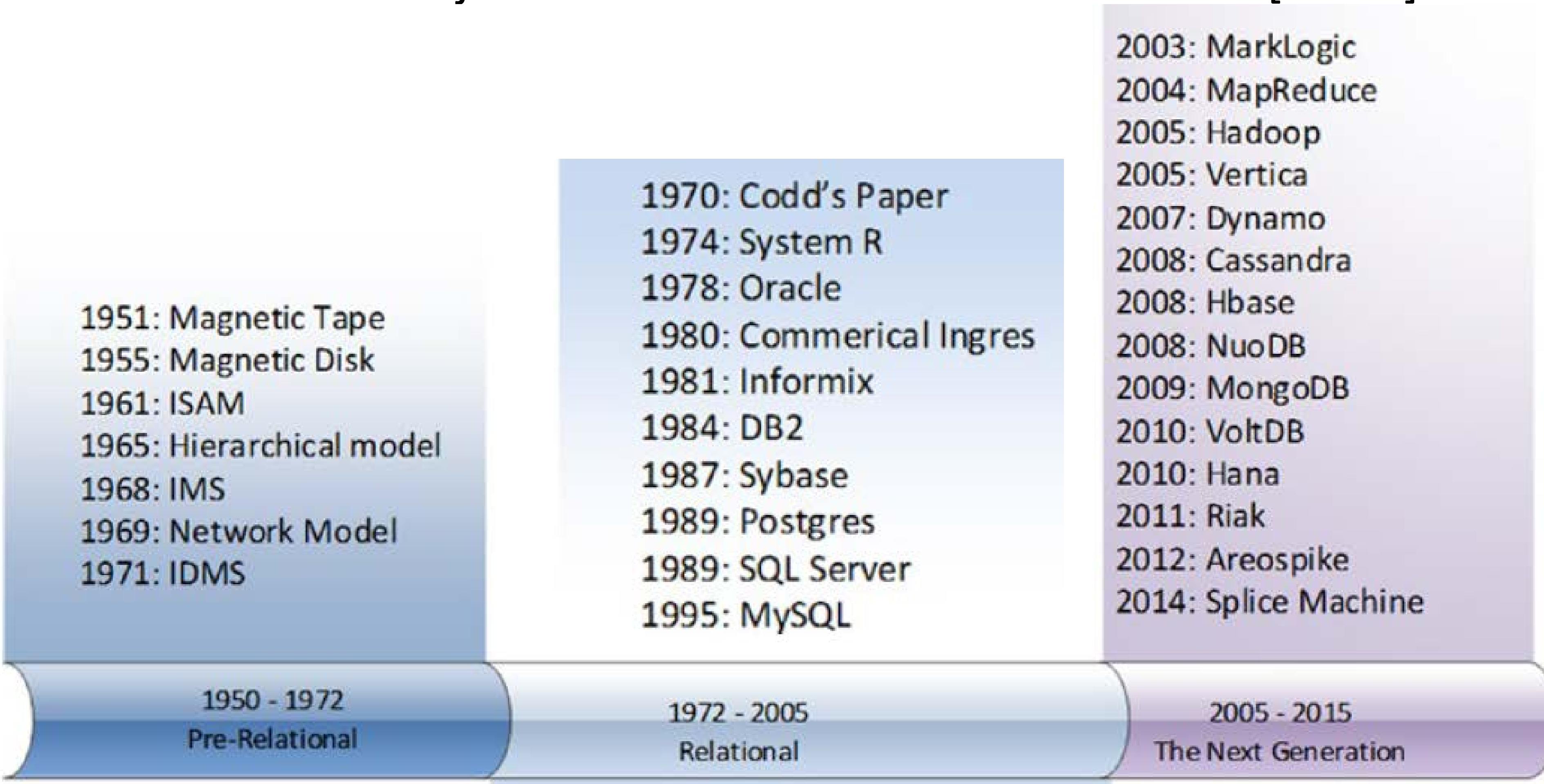
- May not really hard, if we do not need
  - Referential Integrity
  - ACID properties
- Moreover “row based sharding” turn out to be inefficient for query execution or so

Recap of Last Lecture

- Relational databases has its advantages in certain terms and dominated the for decades till the time web 2.0 came post 2000 (a historical note in figure next slide)
- However Relational DBMS fail in following terms
  - Do not fit onto "cluster computing" and hence **do not scale well**
  - Relational Tables and "in memory data objects" are **structurally different** and require lots of reorganization while writing into databases and reading from.
  - "**Strict schema**" comes as a problem in many modern applications

Database Technological Evolutions

- Timeline of major database releases and innovations [text 4]



Is RDB getting phased out?

- Answer is definitely NO!
- RDBs are still going to be as the most common form of database in use [text 1]
- RDB still stands out as better option where
  - **Transaction databases** where ACID compliance is required and cluster distribution is not required
- Or we say other way round; actually there are only two loudly known reasons of choosing NoSQL databases are-

  (1) Huge Data Size

  (2) Ease of development (bypassing ORM and so; however there is a cost associated with, when we do this)

No SQL - Motivations

- Primarily addresses problems of “Relational databases” at web scale and others.
- Relational databases are hard to scale and will always have some limit to grow.
- Relational Databases are not designed to run on “computing clusters”
- Structural mismatch between data in memory (objects) and data in database tables – impedance mismatch.
- “Fixed Schema” comes as a problem sometimes
- Database Integration

No SQL - Motivations

- Fixed Schema
  - modern applications have high variability in terms of data values different objects of a type store.
  - Application may see “unseen” fields as application evolves with time, etc.
  - Putting them in fixed schema makes them inflexible, complex, and wastes storage space, etc.
- Database Integration
  - Integrating databases based on relational have strong dependency on schema
  - If one application changes the schema, other application using the same schema will crash.

“No SQL” History [text 1]

- Player like Google, Amazon were already having their own solutions for challenges faced in their applications by 2005 or so.
- Such solutions created an excitement in database community and there was call for a meet in 2009.
- The term “NoSQL” was a hurray name for the meet.
- The call was for “**open-source, Distributed, non-relational databases**.”
- There were presentation in Meet from Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB, and MongoDB.

The Term “No SQL” [text 1]

- How do we interpret the term “No SQL” and define No SQL database?
- The term can not be interpreted as database “~~systems not having SQL~~”(many no sql systems do provide some variation of SQL, and some, like Cassandra provide almost SQL interface!)
- One most commonly interpretation of No SQL is “Not Only SQL”?
- However this also has a problem?
- This means “Not Only SQL” - does it mean No SQL include SQL (relational) systems too?

  Not correct (relational are not included in No SQL)
- As such, No-SQL does not seem to have a common definition, *Martin Fowler* suggests that it can be described by certain characteristics!

Primary characteristics of No SQL

- Not Relational: NoSQL systems do not include database systems that are Relational, and before!
- Runs on “cluster of computers” – can scale seamlessly well
- Flexible Schema – can be even schema less
  - NoSQL databases operate on either No Schema or has notion of Flexible Schema (Note: relation databases work on strict schema)!
- Easier to program – program objects can be saved as such!
- Do not implement Referential Integrity
- Do not implement ACID but have different notion of consistency [**ensuring ACID is hard on clusters**]

Some related Concepts

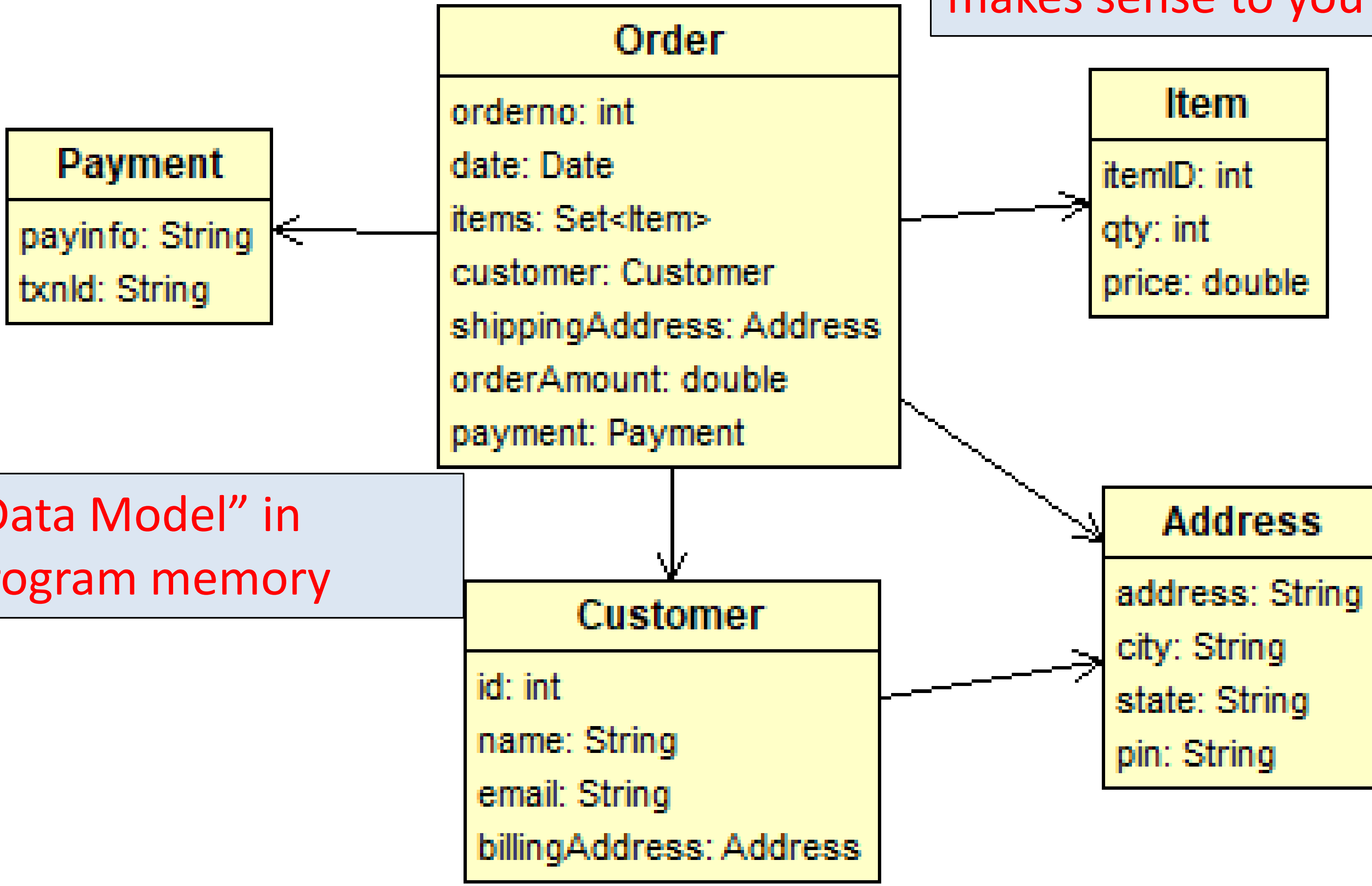
- "**Impedance Mismatch**" as database programming problem
- "Aggregated Oriented Databases"
- Concept of "Key Value"
- Concept of data distribution on "computing nodes" - sharding

Impedance Mismatch

- Consider a scenario of "Order Management of MyFlipCart"
- Let us say we have following data objects with specified data fields that are to be saved in database.
  - Customer: ID, Name, email, billingAddress; ID is unique
    - Customer places orders
    - billingAddress is object with fields of Adress type
  - Order: OrderNo, Date, OrderAmount, customer (customer object – can be reference); OrderNo is unique
  - Order has a list or Item objects, and each Order Item holds data: ItemID, ItemName, ItemCat, Qty, Price

An Ordering System

Hope this diagram makes sense to you?



“Data Model” in program memory

```java
public class Payment {

    private String payinfo;
    private String txnId;
```

```java
public class Customer {

    //business operations to be added

    private int id;
    private String name;
    private String email;
    private Address billingAddress;
```

```java
public class Address {

    private String id;
    private String address;
    private String city;
    private String state;
    private String pin;
}
```

```java
public class Order {

    private int orderno;
    private Date date;
    private Set<Item> items;
    private Customer customer;
    private Address shippingAddress;
    private double orderAmount;
    private Payment payment;
```

```java
public class Item {

    private int itemID;
    private int item_name;
    private int category;
    private int qty;
    private double price;

}
```

- Hop you can visualize content of an order object?
- Here is order object in JSON representation

Problem Context:
How do we store this in database?

A sample Order object

```
{
  "order": {
    "orderNo":123,
    "customer": {
      "id": 1,
      "name": "Mayank",
      "email": "mayank@gmail.com",
      "billingAddress": {"city": "Bangalore"}
    },
    "orderItems":[
      {
        "productId":27,
        "price": 325,
        "cat":"book",
        "productName": "NoSQL Distilled"
      },
      {
        "productId":19,
        "price": 550,
        "cat":"computer accessories",
        "productName": "Pen Drive : 128 GB"
      }
    ],
    "shippingAddress":[{"city":"Ahmedabad"}],
    "payment": {
      "ccinfo":"1000-1000-1000-1000",
      "txnId":"abelif879rft"
```

Relational Tables

Relational Schema to store order data:

Address(id, address, city, state, pin)

Customer(id, name, email, billingAddress_id)

FK: billingAddress_id refers into address

Order(OrderNo, date, cust_id, shipingAddress_id orderAmount,   payinfo, txnId)

FK: cust_id refers into customer, shippingAddress_id refers into address

Items(itemID, ItemName, category)

OrderItem(OrderNo, itemID, qty, price)

FK: OrderNo refers into Order, ItemID refers into Item

Saving Object in relational Table is real pain !!!

```
function save_order(Order order) {

    if !customer_exists() {
        //INSERT INTO Customer Table by collecting data from order object
        cust_id = order.getCustomer().getId()
        cust_name = order.getCustomer().getName
        //so on

        //build a dynamic "INSERT INTO CUSTOMER ..." SQL statement and execute
    }

    //Save Order Row
    //build a dynamic "INSERT INTO ORDER ..." SQL statement and execute
    //so forth
    //should throw exception if already there!

    //Add rows to Order Items
    //build a dynamic "INSERT INTO ORDER ..." SQL statement and execute
    //so forth

}
```

Though JPA tools like hibernate makes life much easy, still problem remains!

Reading Object from Tables is hard !!!

```
Order read_order(int order_no) {

    Order order = new Order();

    //Read row from Order table and populate order object
    sql = "select * from order where order_no=" + order_no
    result = execute(sql);
    order.setId( order_no );
    order.setDate( result.getColumn("date") );
    order.setAmount( result.getColumn("order_amount") );
    customer_id = result.getColumn("customer_id")

    //Read related cutomer row, construct and populate a cust
    Customer customer = new Customer()
    customer.setID( customer_id )
    customer.setName( queryresult.getColumn("name") )
    customer.setEmail( queryresult.getColumn("email") )
    //
    order.setCustomer( customer );
    //so on

    return order;
```

Impedance Mismatch!

- **Problems are mainly due to structure mismatch between data object in memory and its storage in tables.**
  - **“Aggregated Objects” and Normalized Tables**

- It is referred as “impedance mismatch”

- Does it make it clear that “what is impedance mismatch”?

Impedance Mismatch

- People hoped that some thing like Object Oriented Databases would be replacing relational databases but it did not happen
  - Probability OODB could not become success due to its own complexities, and could not provide many of RDB features
- Though most translations (Object to Relations and Relations to Objects) are automated by Object Relational Mapping (ORM) tools like Hibernate or so; they bring in their own complexity, maintenance and computational cost.
- A nice critic of ORM is available from Martin Flower at https://martinfowler.com/bliki/OrmHate.html
- No SQL Databases is helps here!

“Aggregation Oriented Databases”

- Often objects aggregate other objects and that too at multiple level
  - Customer has multiple Orders, and then each order holds multiple items, so forth!
- Therefore let us call objects as aggregated data in this context!
- Normalized tables in Relational Database do not allow storing aggregated data objects. They require to be split in terms of table rows.
  - recall atomic values in relations – definition of 1NF. And hence relational is not aggregate database.
- Database that allow storing “aggregated objects” are called aggregation oriented objects.
- In aggregated databases, our database is in this case is **“collection of Order objects**” or **“collection of Order aggregates”**!

Aggregation Oriented Databases

- Here is an Aggregated Object “Order”
- Order object embeds all object data including customer, items, payments.
- Aggregation Oriented Databases allow us storing such an object directly in the database without breaking in pieces!
- Relational do not; therefore not aggregation oriented.
- All No SQL systems (except graph databases) are aggregation oriented

```
"order": {
  "orderNo":123,
  "customer": {
    "id": 1,
    "name": "Mayank",
    "email": "mayank@gmail.com",
    "billingAddress": {"city": "Bangalore"}
  },
  "orderItems":[
    {
      "productId":27,
      "price": 325,
      "cat":"book",
      "productName": "NoSQL Distilled"
    },
    {
      "productId":19,
      "price": 550,
      "cat":"computer accessories",
      "productName": "Pen Drive : 128 GB"
    }
  ],
  "shippingAddress":[{"city":"Ahmedabad"}],
  "payment": {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft"
```

“Aggregated Oriented Databases”

- Aggregation oriented databases are in advantage because programmers do not have to all object splitting jobs for saving object in databases. Save in terms of computation required.
- Now what do we loose in such a database?
  - “Not Normalized” – will have data redundancy and hence anomalies.
    - Note: customer data will be repeated with all orders of a customer.
  - Aggregations are often good only for certain set of queries where bad for other kind of queries?

Downside of "Aggregation Oriented Databases"

- First one is not normalized and hence redundancy and anomalies.
- An "aggregations" is always good for certain set of queries only while bad for other kind of queries.
- Consider our Order aggregation here. (assuming that orders are sorted in order of order no and hence in the order of order date). Then it is good for queries like
  - Get an Order for a given order-no
  - All order for a given date, or month
  - Compute monthly sales or so.

Downside of "Aggregation Oriented Databases"

- But not good for following kind of queries
  - All orders of a customer?
  - State-wise sales of an item
  - Monthly sales of an Item?
- Since data are ordered in terms of order number and hence date.
- Normally No SQL databases distribute their data on multiple computing nodes based on "Key" and here typically Key Order No and date can be part of key.
- Now for specific customer (and item for that matter) data are spread across all nodes.
- For a customer or item based queries, we are require processing objects on all computing nodes. <u>A very undesirable situation</u>.

Downside of “Aggregation Oriented Databases”

- A serious drawback of No SQL databases, therefore is there aggregation (which is basically database design) is dictated by kind of queries it is aimed to answer.
- Which is not the case with relational model, database design does not consider the “kind of queries”.
- So we may different ways of aggregations!

Downside of "Aggregation Oriented Databases"

- So we may different ways of aggregations:
  - Customer embedded in Order, or
  - Order embedded in Customer
  - Customer and Order two different aggregations and having references into other one.
    - However this may lead to split the program objects into multiple pieces – something that bothers us in relational at extreme
    - Higher aggregation granularity require less splitting of object at save time, but then it becomes more specific to a set of queries!

Different Aggregations

- **Databases is collection of Customer objects**
- All orders of a customer are embedded in customer objects
- Not good strategy when we need to process and query orders in time dimension.

Customer Aggregate


```
{
  "customer": {
    "id": 1,
    "name": "Mayank",
    "email": "mayank@gmail.com",
    "billingAddress": [{"city": "Bangalore"}],
    "orders": [
      {
        "orderNo":123,
        "customerId":1,
        "orderItems":[
          {
            "productId":27,
            "price": 325,
            "cat":"book",
            "productName": "NoSQL Distilled"
          },
          {
            "productId":19,
            "price": 550,
            "cat":"computer accessories",
            "productName": "Pen Drive : 128 GB"
          }
        ],
        "shippingAddress":[{"city":"Ahmedabad"}],
        "payment": {
          "ccinfo":"1000-1000-1000-1000",
```

- **Databases contains two collections: Customers and Orders**
- Orders are taken out of customer
- Order object has reference to customer objects (like the foreign key in relational, but referential integrity check is not done.
- Now allows processing orders in time space also

Order Aggregate

```
{
'order": {
  "orderNo":123,
  "customerId":1,
  "orderItems":[
    {
      "productId":27,
      "price": 325,
      "cat":"book",
      "productName": "NoSQL Distilled"
    },
    {
      "productId":19,
      "price": 550,
      "cat":"computer accessories",
      "productName": "Pen Drive : 128 GB"
":[{"city":"Ahmedabad"}],
-1000-1000-1000",
f879rft"
}
```

Customer aggregate

```
{
  "customer": {
    "id": 1,
    "name": "Mayank",
    "email": "mayank@gmail.com",
    "billingAddress": {"city": "Bangalore"}
  }
}
```

- **Databases contains two collections**
- Both collections have reference to each other. Called bidirectional references.
- Eases out and querying becomes efficient!

Order Aggregate

```
rder": {
"orderNo":123,
"customerId":1,
"orderItems":[
  {
    "productId":27,
    "price": 325,
    "cat":"book",
    "productName": "NoSQL Distilled"
  },
  {
    "productId":19,
```

```
ies",
e : 128 GB"
Ahmedabad"}],
000",
     }
   }
}
```

Customer aggregate

```
{
  "customer": {
    "id": 1,
    "name": "Mayank",
    "email": "mayank@gmail.com",
    "billingAddress": {"city": "Bangalore"},
      "orders": [{ "orderno" : 123}, { "orderno" : 154} ]
  }
}
```

“Aggregation Oriented” Databases

Wrap Up

- If a database system allows us saving “aggregated” objects as such, the we call them as “Aggregation Oriented” Database system.
- Most No SQL database systems are aggregation oriented except graph databases.
- Aggregate oriented databases are biased towards a “query load” (querying use cases)
- Having appropriate aggregate is key in No SQL database designs!

Notion of “Key Value”

- Concept of Key value is not new in Programming world.
- Hopefully you have used one following in your programming
  - HashMap in Java
  - map in C++
  - maps in python
- In some languages have concepts of associated arrays which is basically a simplification of map only.
- Idea of Map is basically to perform a <u>“key based lookup in a collection of objects”</u>
- We can have collection of items, orders, customer as map.

Notion of “Key Value”

- Idea of Map is basically to perform a “key based lookup in a collection of objects”
- In a sense Maps can also be called and used as “In Memory Key Value Database”
- Benefit of maps is “efficient access” of an object in a “collection” of objects
  - Note the word collection here!
- Operational examples on Map of Item objects

```
Item a = items.get( 123 );
```

//gets the item from collection. Parameter is Item No

```
items.put( 313, item_x );
```

//puts an item into items collection. Parameters are item no and item object to be put. If item with the given item number already exists, the operation replaces existing object with passed one.

Last Lecture - Recap

- How No SQL systems fits into problem space of Relational Systems
- No SQL Database characteristics
  - Runs on cluster
  - Flexible Schema
  - Addresses the impedance mismatch problem
  - Aggregation Orientation
- Impedance Mismatch – structural mismatch between program data objects

Last Lecture - Recap

- Aggregation Oriented Databases
  - Databases that allow saving aggregates as such.
  - It makes No SQL systems program friendly, and
  - Cluster friendly – distributed over computers based keys of on aggregates
- Concept of Key-Value
  - Idea of key-value is not new to programming community – fundamentally has been used in binary search trees, or so.
  - It is central to No SQL systems
  - Right from basic systems like map-reduce to sophisticated no-sql systems like Cassandra

Notion of "Key Value"

- So, what is "Key" and "Value"?
  - Key is search or Lookup Key for data object in its collection
  - Value is "data object" that is put in the collection
  - In our example items; item_no is key and item object is value.
- Primary operations on maps are "**Put**" and "**Get**".
- <u>Again note that search is possible only on Key</u>
- No SQL database explores the idea of key value in databases.

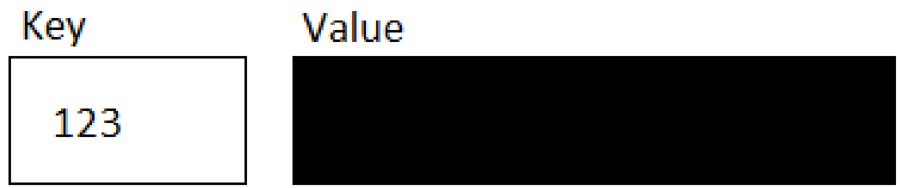
Types of “No SQL database models”

- No-SQL databases (except graph) explore the idea of key value in databases.
- All of them are said to be “Key-Value” databases; but have significant variability in terms of “**representation**” and “**processibility**” of value part!
- Here are databases that are based on Key-Value strategy
  - Key-Value databases
  - Document Oriented databases
  - Column Family databases

Key Value databases

- Make sure that you have understood the meaning by “Key” and “Value”
- Key value database are aggregation oriented databases, and we have
  - “Aggregation” as value, and
  - ID of aggregation as Key (every aggregation is supposed to have a key before it can be saved in the database)
- If “Map” is called as “In Memory Database” Key value database is basically a persistent version of Maps.

Key Value databases



Data Model

- We have value as stream of bytes.
- For a database system "Value" remains as chunk, and it does not do any processing on it.
- Content of "value" or "aggregate" is created and processed by applications.
- Typical content of Value is object in JSON form because this format is very much programming friendly. Easy to serialize an program object to JSON and vice versa.
- Have No Schema. You can put anything in value part.

Key Value databases

- Basic Operations are same as of "Map":
  - **Get**, and **Put**
- Key Value database system allows saving or reading data as "value" chunk as it is. Does not know what is there in it.
- <u>We can only perform search operations based on Key.</u>
- While this is basic definition of Key Value databases, real key value implementations may do little extra!
- For example Amazon Dynamo DB allows creating indexes on some data attributes (in value), and hence allow searching based on those data from value part.
- Data are distributed on computing nodes based on Key.
- DynamoDB, Redis, Riak are popular key-value databases.

Document Oriented Databases

Key	Value
234	"id": 234, "name": "Mayank", "email": "mayank@gmail.com", "billingAddress": {"city": "Bangalore"}

- Document Oriented Databases are key value database that make value part partially visible.
- Value may have partial schema specified.
  - May have a required set of attributes and some constraints on them.
  - Schema remain flexible - any extra set of attribute-values.

Document Oriented Databases

- Allow searching on
  - Key (data distribution again happens based on key)
  - Provides set of operations that allow facetted search, i.e. based on some attributes from value
- MongoDB is popular document database system

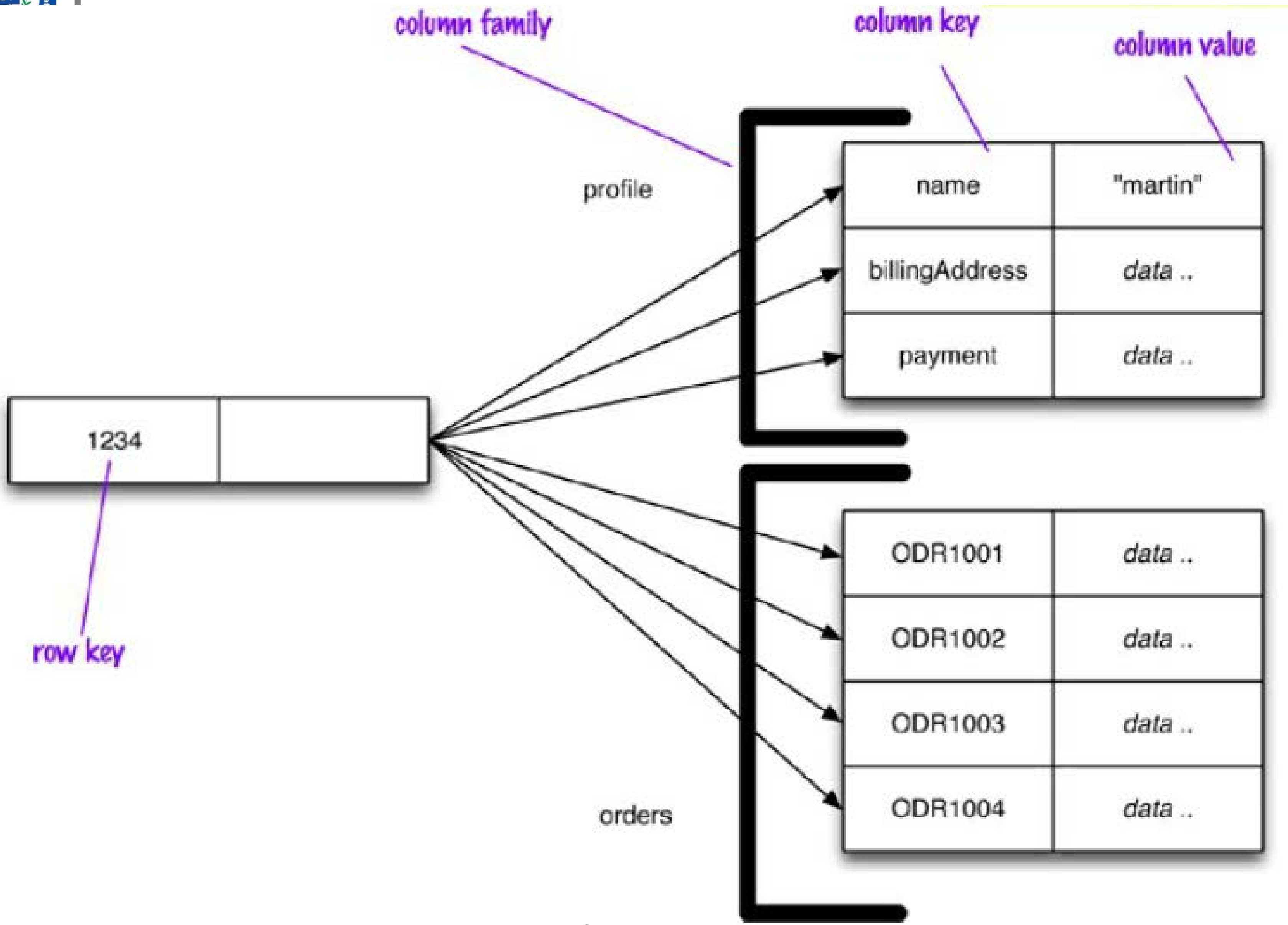
Column Family Databases

- Brings two concepts
  - Row
  - Column Family
- Still use notion of Key. Rows are identified by key.
- Row is split between “multiple aggregates”. Each aggregate is called “column family”.
- Content of each column family is schema less
- All aggregates (column families) for a given key can be joined if required and produce complete row.

Column Family Databases

- Column family databases are most structured (schema terms) among No SQL systems.
- BigTable, HBase, Cassandra, CouchBase are popular column family database.

Column Family Databases



References/Further Readings

[1] Chapter 1 of book *NoSQL distilled*.

[2] Relational Databases Are Not Designed For Scale
https://www.marklogic.com/blog/relational-databases-scale/

[3] Herbst, Nikolas Roman, Samuel Kounev, and Ralf Reussner. "Elasticity in cloud computing: What it is, and what it is not." *Proceedings of the 10th International Conference on Autonomic Computing ({ICAC} 13)*. 2013.

[4] History of Data Modeling:
http://graphdatamodeling.com/GraphDataModeling/History.html

[5] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: Simplified data processing on large clusters." (2004)