An introduction to

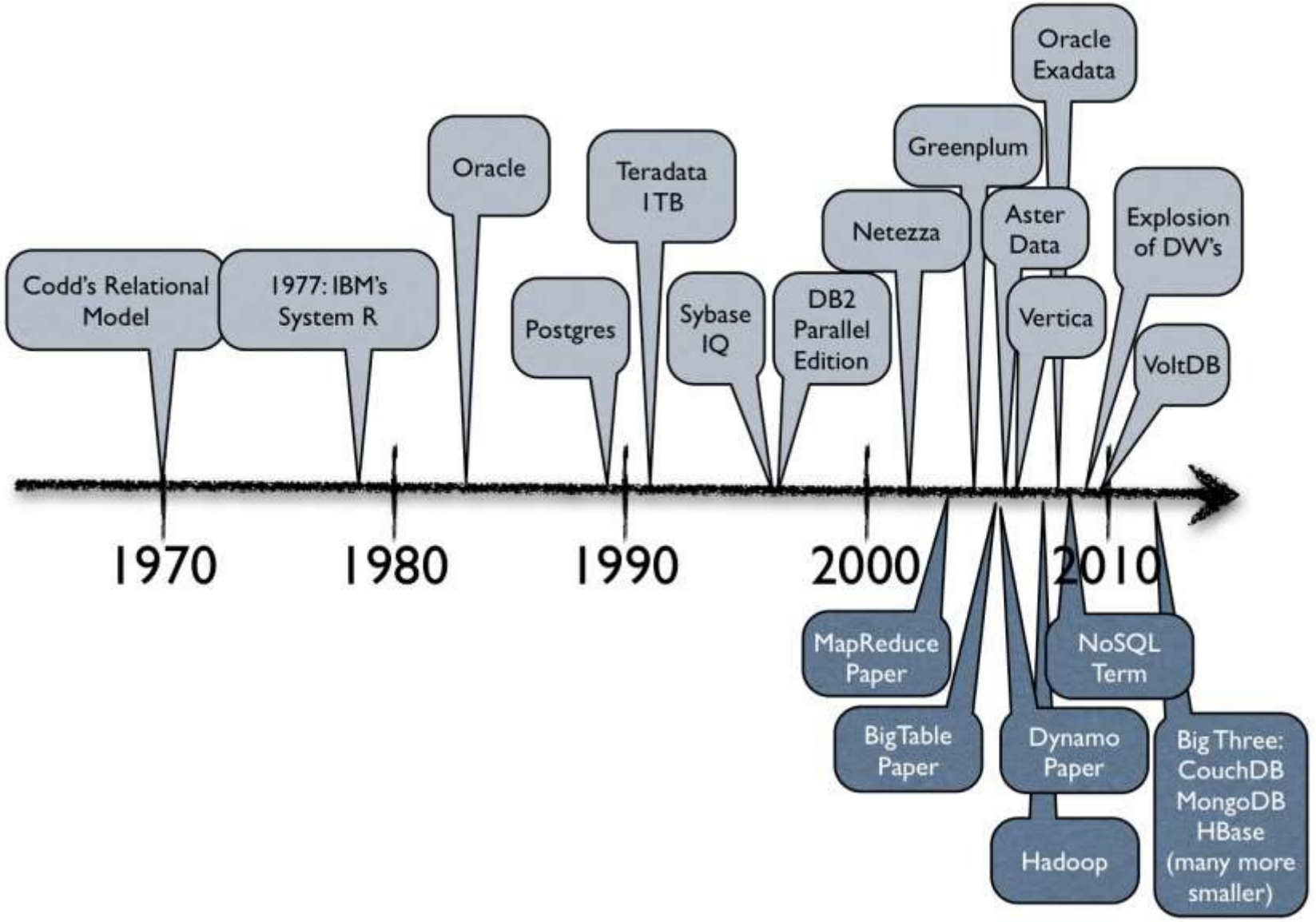
# NoSQL databases

Dr. Faisal Kamiran

# Agenda:

- A brief history of databases
- NoSQL why, what and when?
- Aggregate Data Models
- BASE vs ACID
- CAP theorem
- Denormalization

# A brief history of databases



# Relational databases

Benefits of Relational databases:

- Designed for all purposes
- ACID
- Strong consistancy, concurrency,  recovery
- Mathematical background
- Standard Query language (SQL)
- Lots of tools to use with i.e: Reporting services, entity frameworks, ...
- Vertical scaling (up scaling)

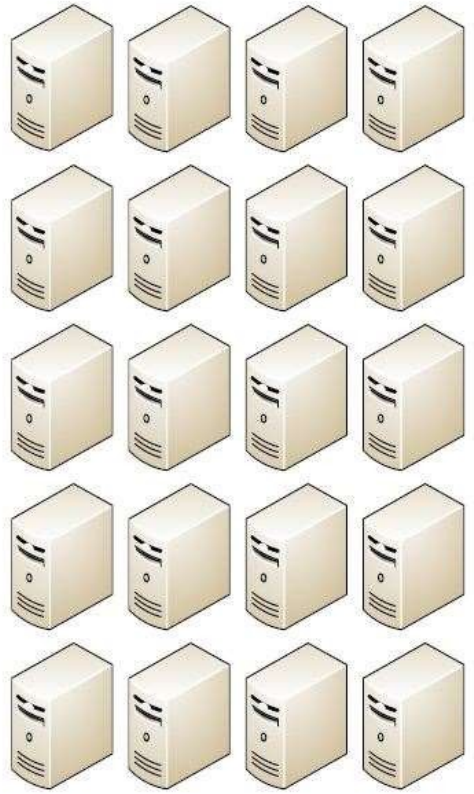
# Era of Distributed Computing

## But...

- Relational databases were not built for **distributed applications.**

## Because...

- Joins are expensive
- Hard to scale horizontally
- Expensive (product cost, hardware,  Maintenance)



# Era of Distributed Computing

## But...

- Relational databases were not built for **distributed applications.**

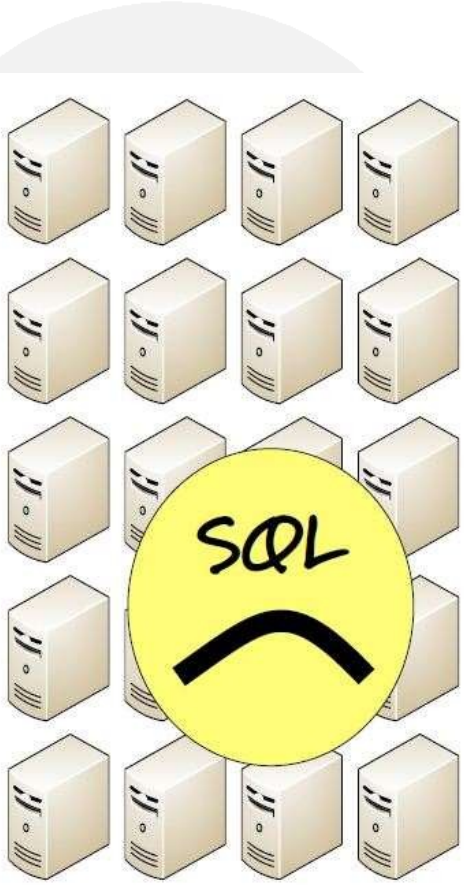
## Because...

- Joins are expensive
- **Hard to scale horizontally**
- Expensive (product cost, hardware, Maintenance)

## And....

It's weak in:

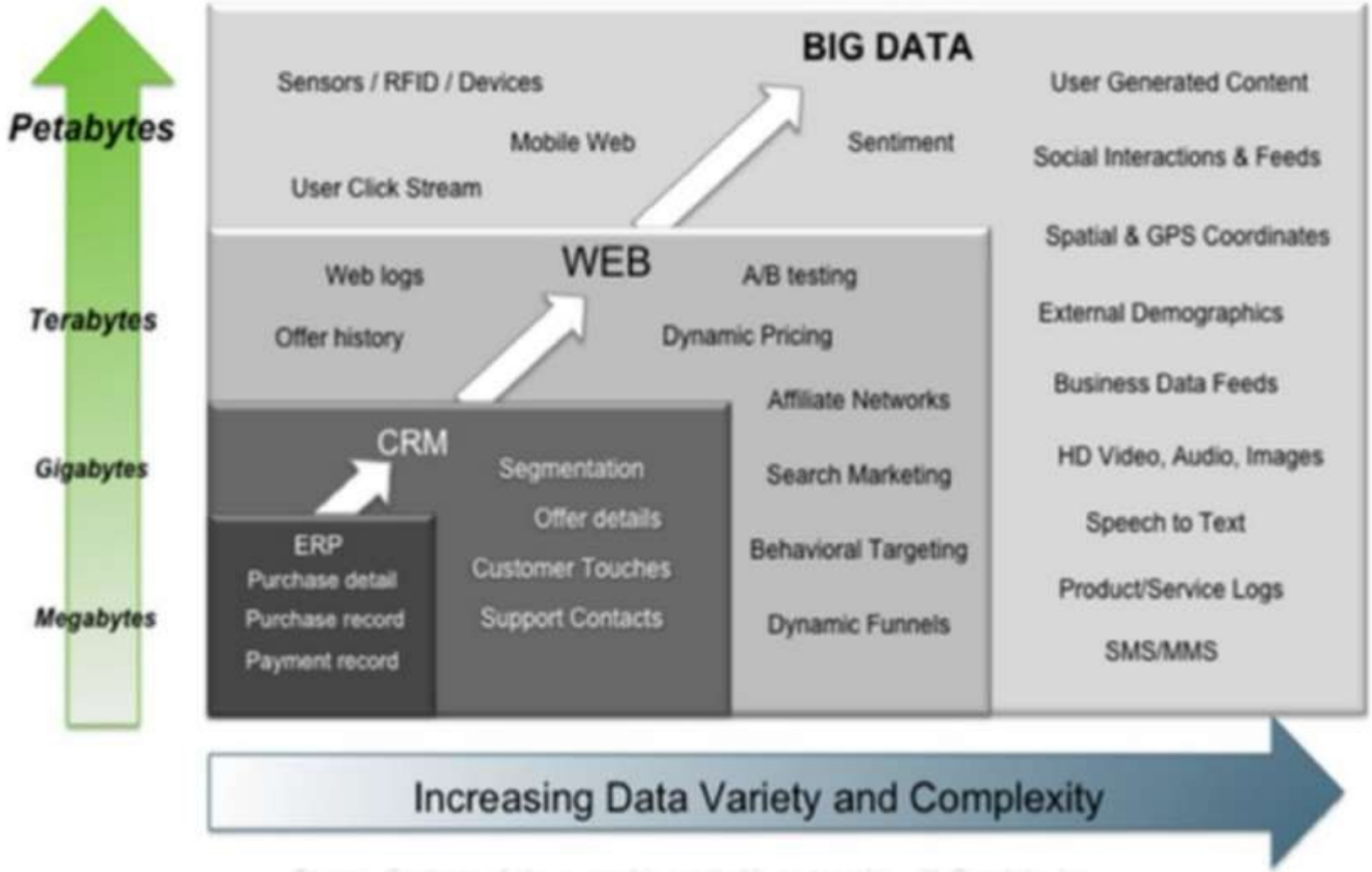
- Speed (performance)
- High availability
- Partition tolerance

# Rise of Big data

Three V(s) of Bigdata:

- Volume
- Velocity
- Variety



# NoSQL why, what and when?

- Google & Amazon bulit their own databases (Big table & Dynamo)
- Facebook invented Cassandra and is using thousands of them
- #NoSQL was a twitter hashtag for a conference in 2009
- The name doesn’t indicate its characteristics
- There is no strict defenition for NoSQL databases
- There are more than 150 NoSQL databases (nosql-database.org)

# Characteristics of NoSQL databases

- Non relational
- Cluster friendly
- Schema-less
- 21 century web
- Open-source

# Characteristics of NoSQL databases

NoSQL avoids:

- Overhead of ACID transactions
- Complexity of SQL query
- Burden of up-front schema design
- DBA presence
- Transactions (It should be handled at application layer)

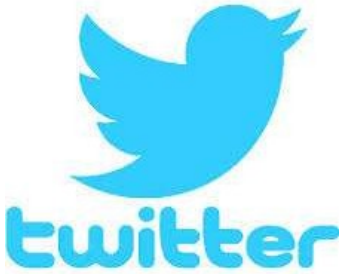
Provides:

- Easy and frequent changes to DB
- Horizontal scaling (scaling out)
- Solution to Impedance mismatch
- Fast development

# NoSQL is getting more & more popular



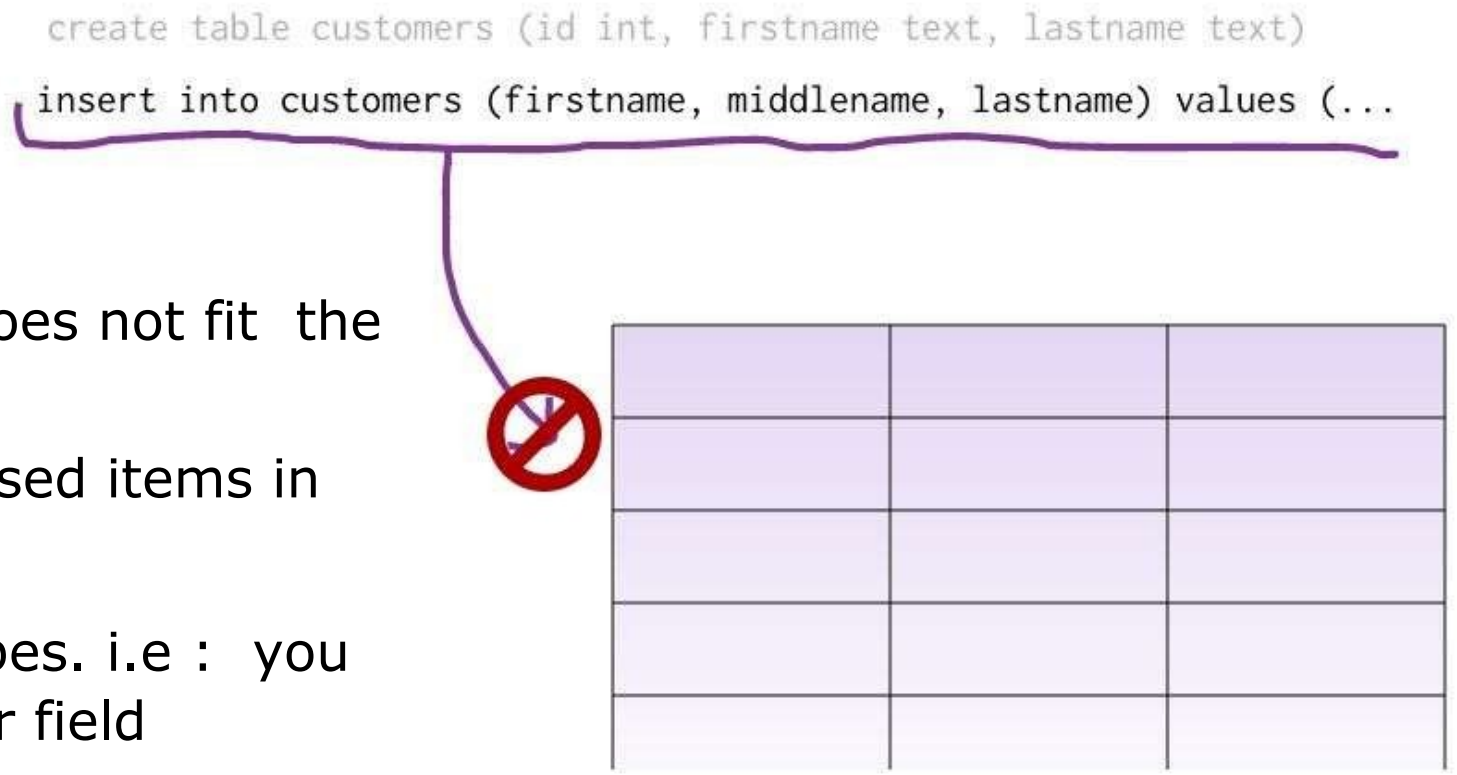





# What is a schema-less data model?

In relational Databases:

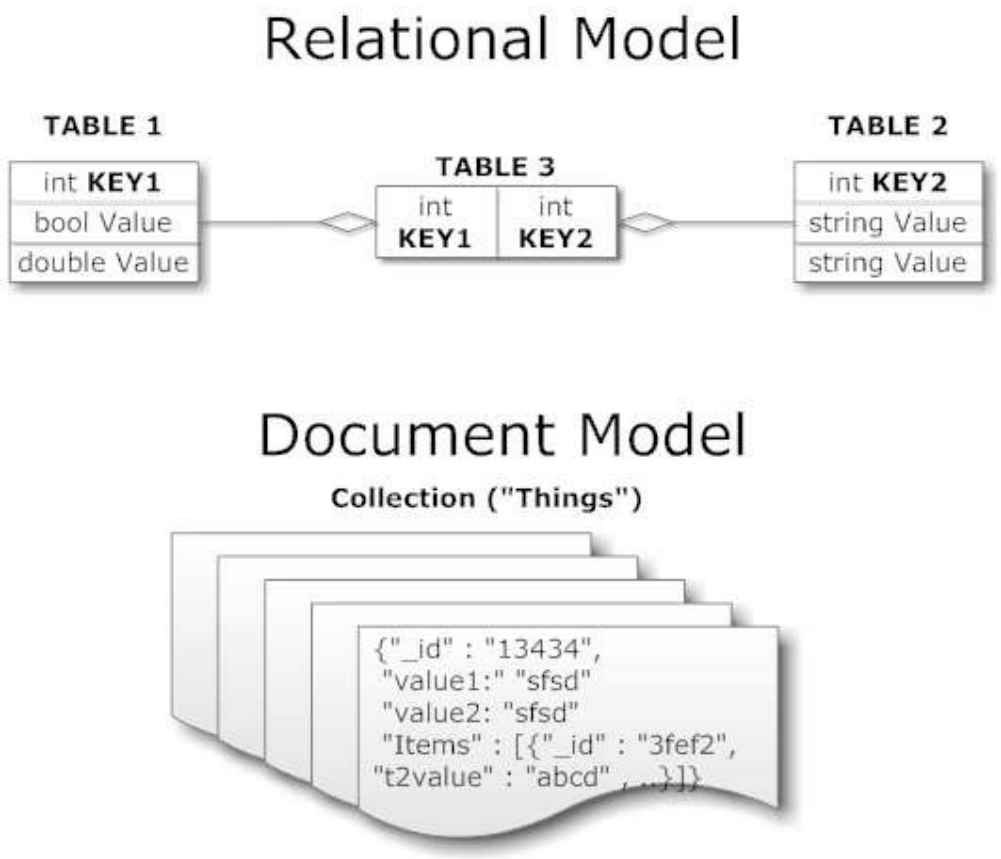



- You can’t add a record which does not fit the schema
- You need to add NULLs to unused items in a row
- We should consider the datatypes. i.e : you can’t add a stirng to an interger field
- You can’t add multiple items in a field (You should create another table: primary-key, foreign key, joins, normalization, ... !!!)

# What is a schema-less datamodel?

In NoSQL Databases:

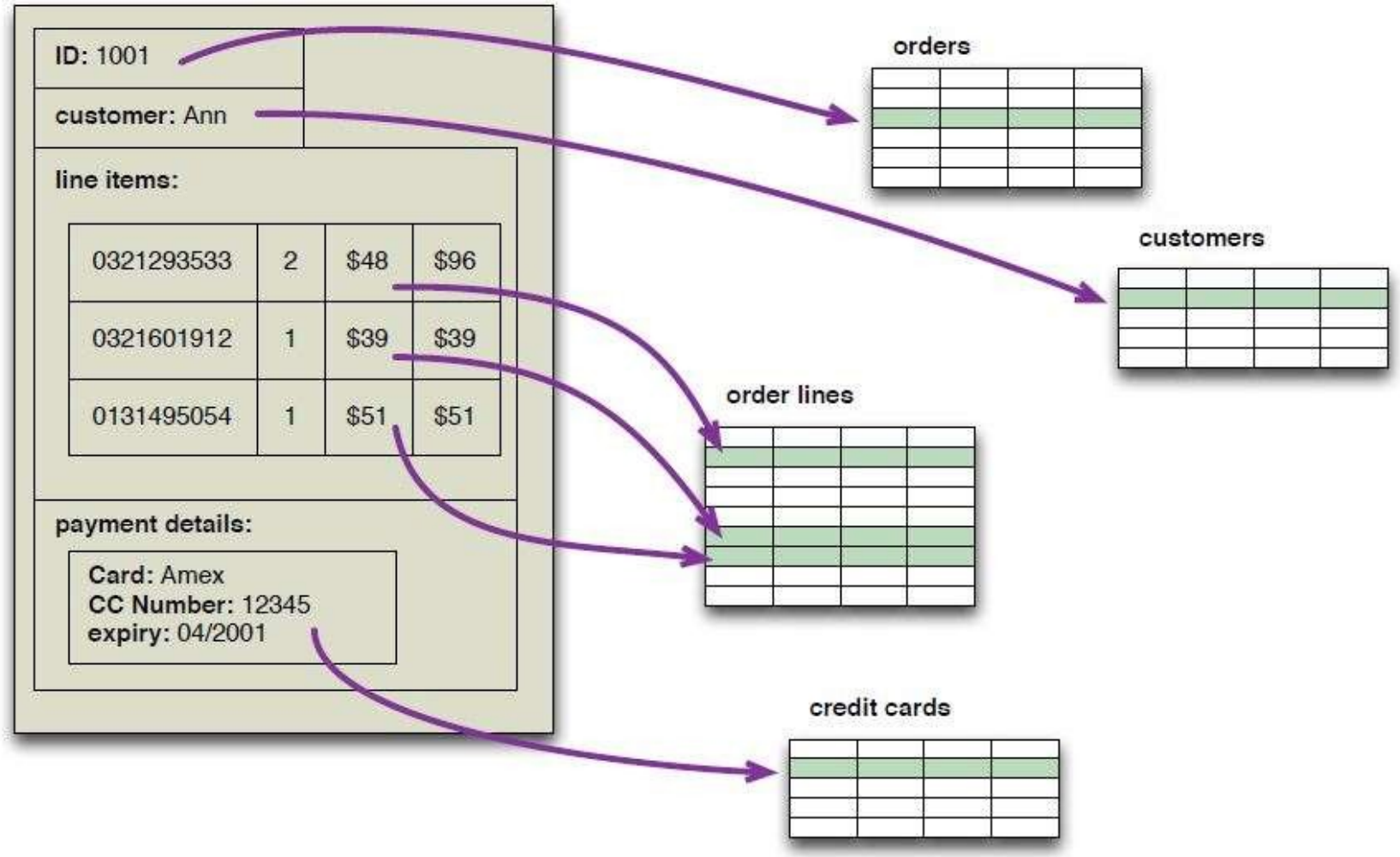
- There is no schema to consider
- There is no unused cell
- There is no datatype (implicit)
- Most of considerations are done in application layer
- We gather all items in an aggregate (document)



# What is Aggregation?

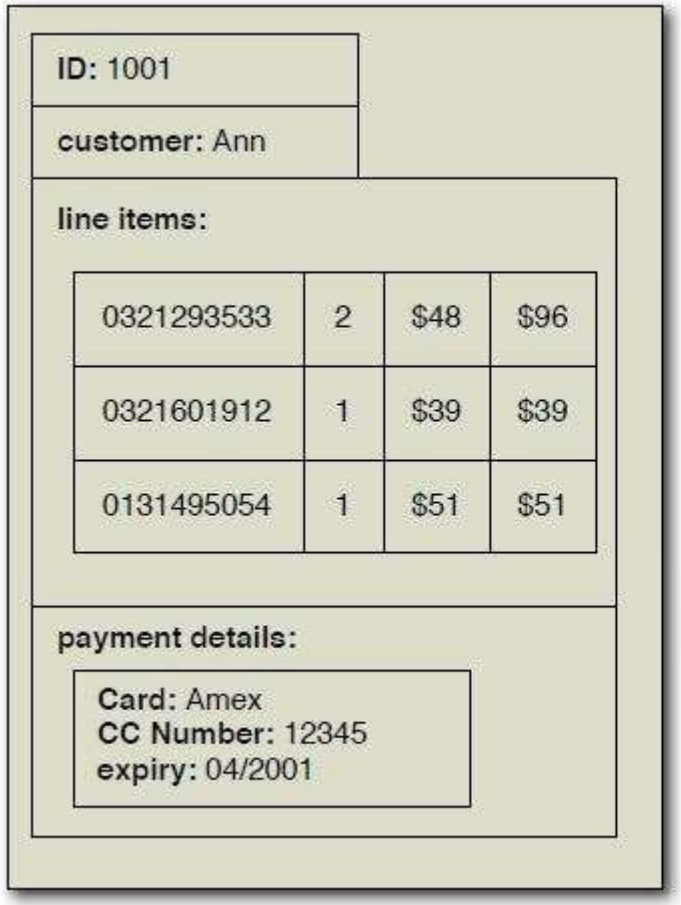
- The term comes from Domain Driven Design
- Shared nothing architecture
- An aggregate is a cluster of domain objects that can be treated as a single unit
- Aggregates are the basic element of transfer of data storage - you request to load or save whole aggregates
- Transactions should not cross aggregate boundaries
- This mechanism reduces the join operations to a minimal level

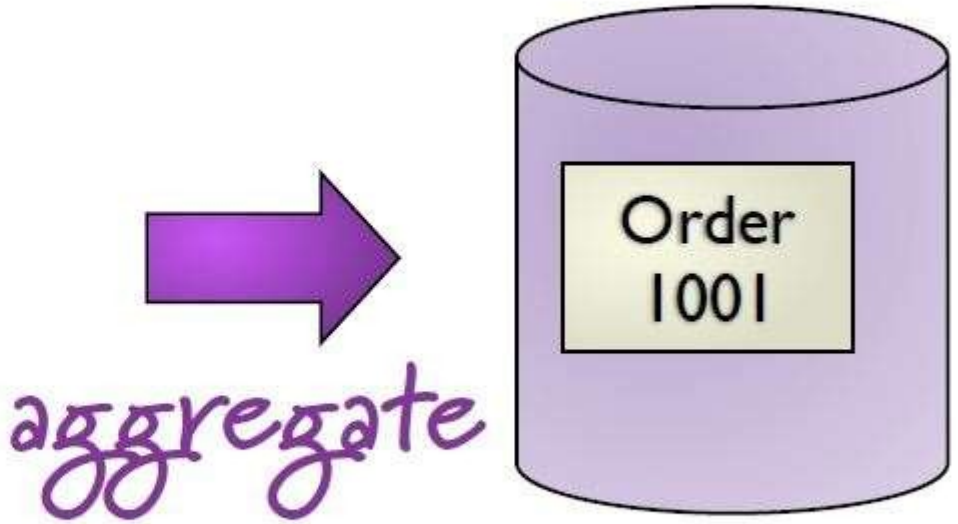
# What is Aggregation?



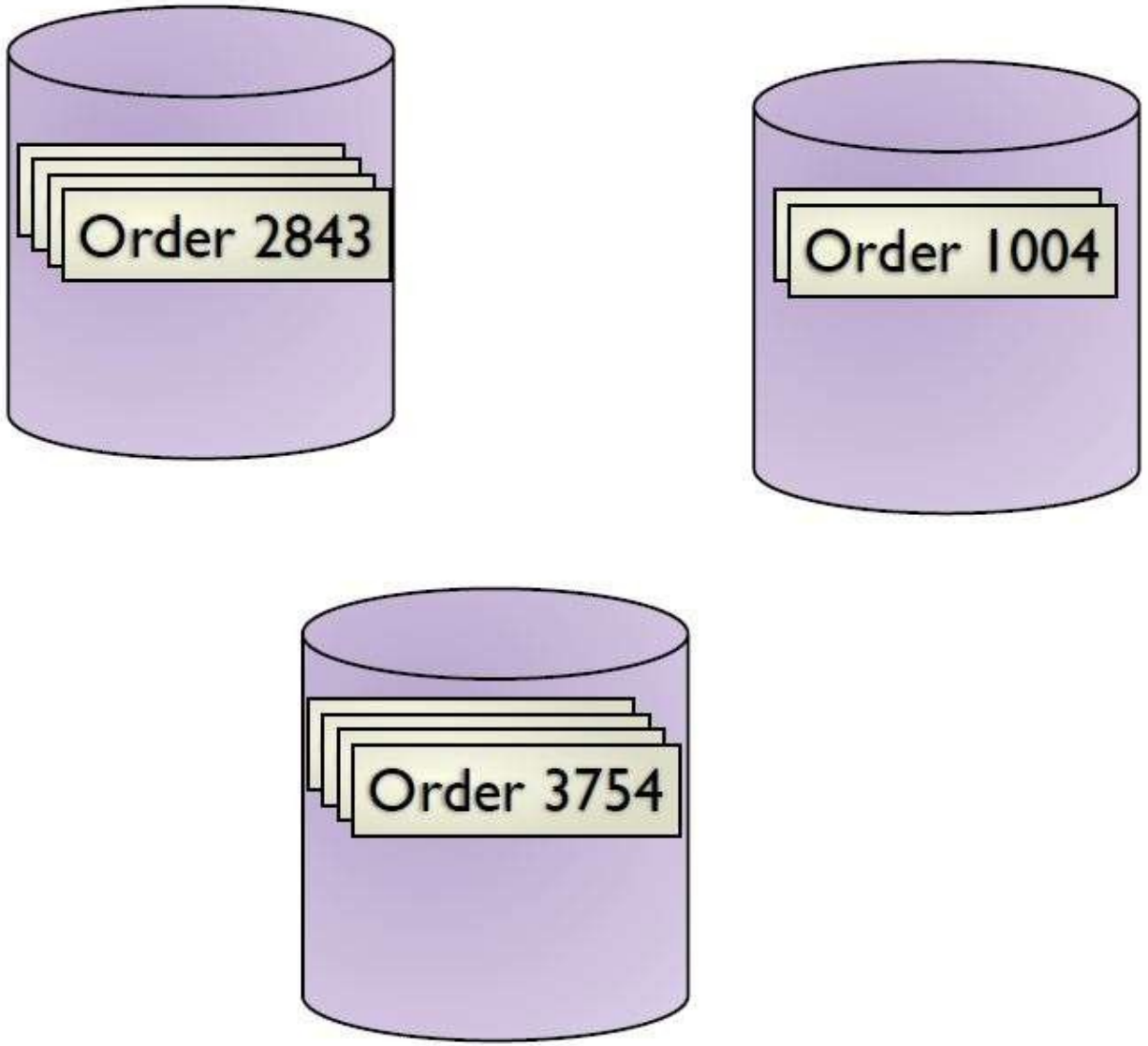
# What is Aggregation?

```
{
 "id": "1001",
 "firstName": "Ann",
 "lastName": "Williams",
 "age": 55,
"purchasedItems":
   {
    0321290533 {qty, price… }
    0321601912 {qty, price… }
    0131495054 {qty, price… }
   }
"paymentDetails":
   { cc info… }
"address":
   {
    "street": "1234 Park",
    "city": "San Francisco",
    "state": "CA",
    "zip": "94102"
   }
}
```



# What is Aggregation?



# Aggregate Data Models

NoSQL databases are classified in four major datamodels:

- Key-value
- Document
- Column family
- Graph


<u>Each DB has its own query language</u>

# Key-value data model

- The main idea is the use of a hash table
- Access data (values) by strings called keys
- Data has no required format – data may have any format
- Data model: (key, value) pairs
- Basic Operations:

  Insert(key,value), Fetch(key),Update(key), Delete(key)

| Car | |
|---|---|
| Key | Attributes |
| 1 | Make: Nissan<br>Model: Pathfinder<br>Color: Green<br>Year: 2003 |
| 2 | Make: Nissan<br>Model: Pathfinder<br>Color: Blue<br>Color: Green<br>Year:2005<br>Transmission: Auto |

# Key-value data model

- "Value" is stored as a "blob"
  - Without caring or knowing what is inside
  - Application is responsible for understanding the data

- Main observation from Amazon (using **Dynamo**)

  – "There are many services on Amazon's platform that only need primary-key access to a data store."

  E.g. Best seller lists, shopping carts, customer preferences, session management, sales rank, product catalog

# Column family data model

- The column is lowest/smallest instance of data.
- It is a tuple that contains a name, a value and a timestamp

ColumnFamily: Authors

| Key | Value |
|---|---|
| "Eric Long" | Columns<br>Name: Value<br>"email": "eric (at) long.com"<br>"country": "United Kingdom"<br>"registeredSince": "01/01/2002" |
| "John Steward" | Columns<br>Name: Value<br>"email": "john.steward (at) somedomain.com"<br>"country": "Australia"<br>"registeredSince": "01/01/2009" |
| "Ronald Mathies" | Columns<br>Name: Value<br>"email": "ronald (at) sodeso.nl"<br>"country": "Netherlands, The"<br>"registeredSince": "01/01/2010" |

# Column family data model

**Some statistics about Facebook Search** (using **Cassandra**)

- MySQL > 50 GB Data
  - Writes Average : ~300 ms
  - Reads Average : ~350 ms
- Rewritten with Cassandra > 50 GB Data
  - Writes Average : 0.12 ms
  - Reads Average : 15 ms

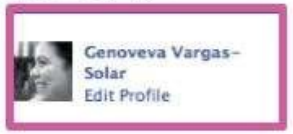
# Document-based datamodel

- Usually JSON like interchange model.
- Query Model: JavaScript-like or custom.
- Aggregations: **Map/Reduce**
- Indexes are done via B-Trees.
- unlike simple key-value stores, both keys and values are fully searchable in document databases.

```
{
  person: {
    first_name: "Peter",
    last_name: "Peterson",
    addresses: [
      {street: "123 Peter St"},
      {street: "504 Not Peter St"}
    ],
  }
}
```

# Overview of a Document-based datamodel

```
SELECT  name, pic, profile_url
FROM    user
WHERE   uid = me()
```




```
SELECT  name
FROM    friendlist
WHERE   owner = me()
```

```
SELECT  name
FROM    group
WHERE   gid IN ( SELECT  gid
                 FROM    group_member
                 WHERE   uid = me() )
```

https://developers.facebook.com/docs/reference/fql/

```
SELECT  message, attachment
FROM    stream
WHERE   source_id = me() AND type = 80
```

```
SELECT  name, pic
FROM    user
WHERE   online_presence = "active"
        AND
        uid IN ( SELECT  uid2
                 FROM    friend
                 WHERE   uid1 = me() )
```

# A sample MongoDB query

**MySQL**

```
SELECT * FROM customerOrder, orderItem, product
WHERE
customerOrder.orderId = orderItem.customerOrderId
AND orderItem.productId = product.productId
AND product.name LIKE '%Refactoring%'
```

**MongoDB**

```
db.orders.find( {"items.product.name":/Refactoring/} )
```

**There is no join in MongoDB query**
**Because we are using an aggregate data model**

# BASE

Almost the opposite of ACID.

- Basically available: Nodes in the a distributed environment can go down, but the whole system shouldn’t be affected.
- Soft State (scalable): The state of the system and data changes over time.
- Eventual Consistency: Given enough time, data will be consistent across the distributed system.

# BASE vs ACID

ACID:

- Strong consistency.
- Less availability.
- Pessimistic concurrency.
- Complex.

BASE:

- Availability is the most important thing. Willing to sacrifice for this (CAP).
- Weaker consistency (Eventual).
- Best effort.
- Simple and fast.
- Optimistic.

# What we need?

- We need a distributed database system having such features:
  - **Fault tolerance**
  - **High availability**
  - **Consistency**
  - **Scalability**

# What we need?

- We need a distributed database system having such features:
  - **Fault tolerance**
  - **High availability**
  - **Consistency**
  - **Scalability**

**Which is impossible!!!**

**According to CAP theorem**

# Should we...?

- In some cases getting an answer quickly is more important than getting a correct answer
- By giving up ACID properties, one can achieve higher performance and scalability.
- Any data store can achieve Atomicity, Isolation and Durability but do you always need **consistency**?
- Maybe we should implement Asynchronous Inserts and updates and should not wait for confirmation?

# CAP theorem

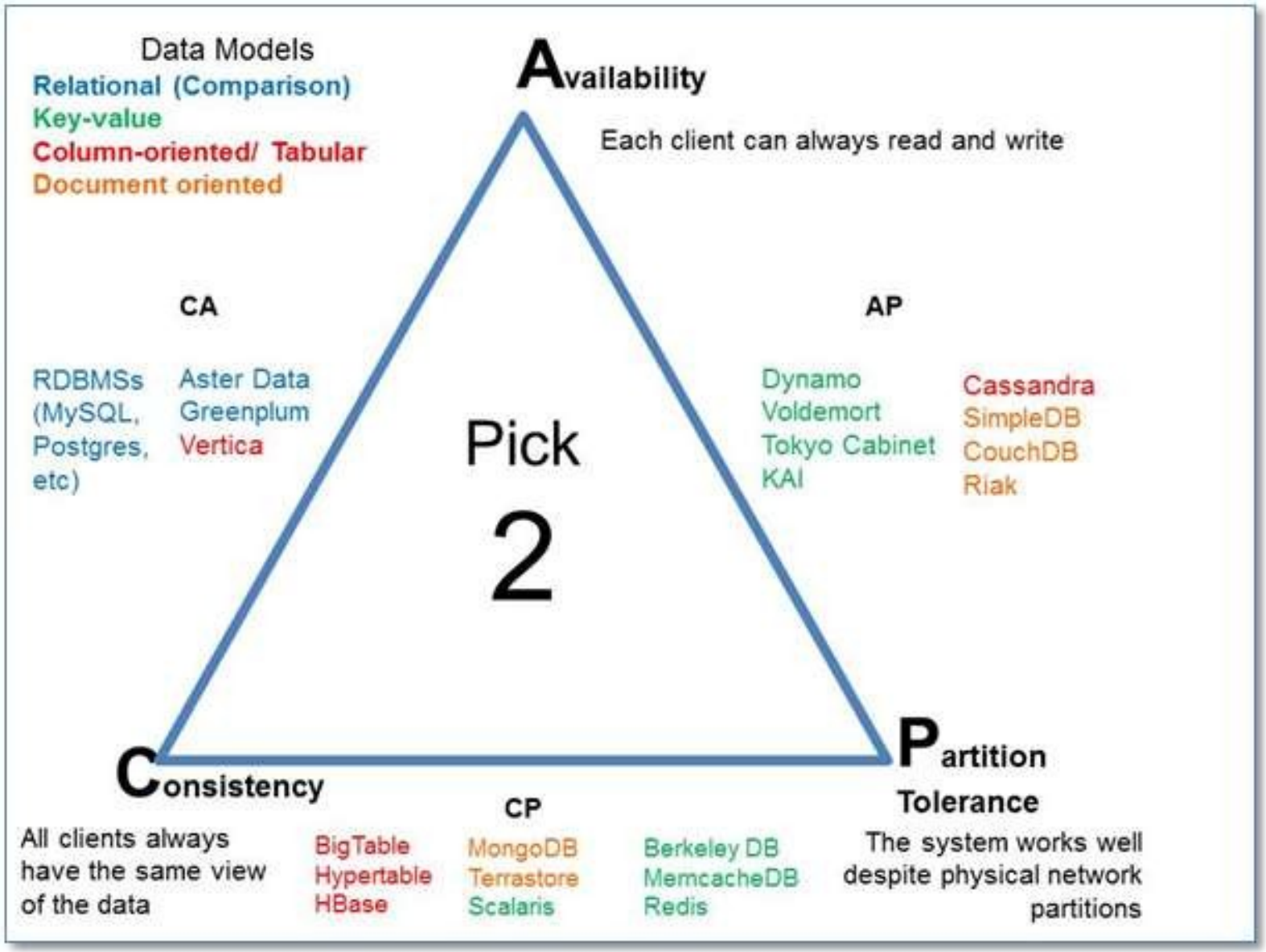
- **C**onsistency: **Clients should read the same data**. There are many levels of consistency.
- **Availability**: Data to be available.
- **P**artial Tolerance: Data to be partitioned across network segments due to network failures.

**2000** Prof. Eric Brewer, PoDC Conference Keynote

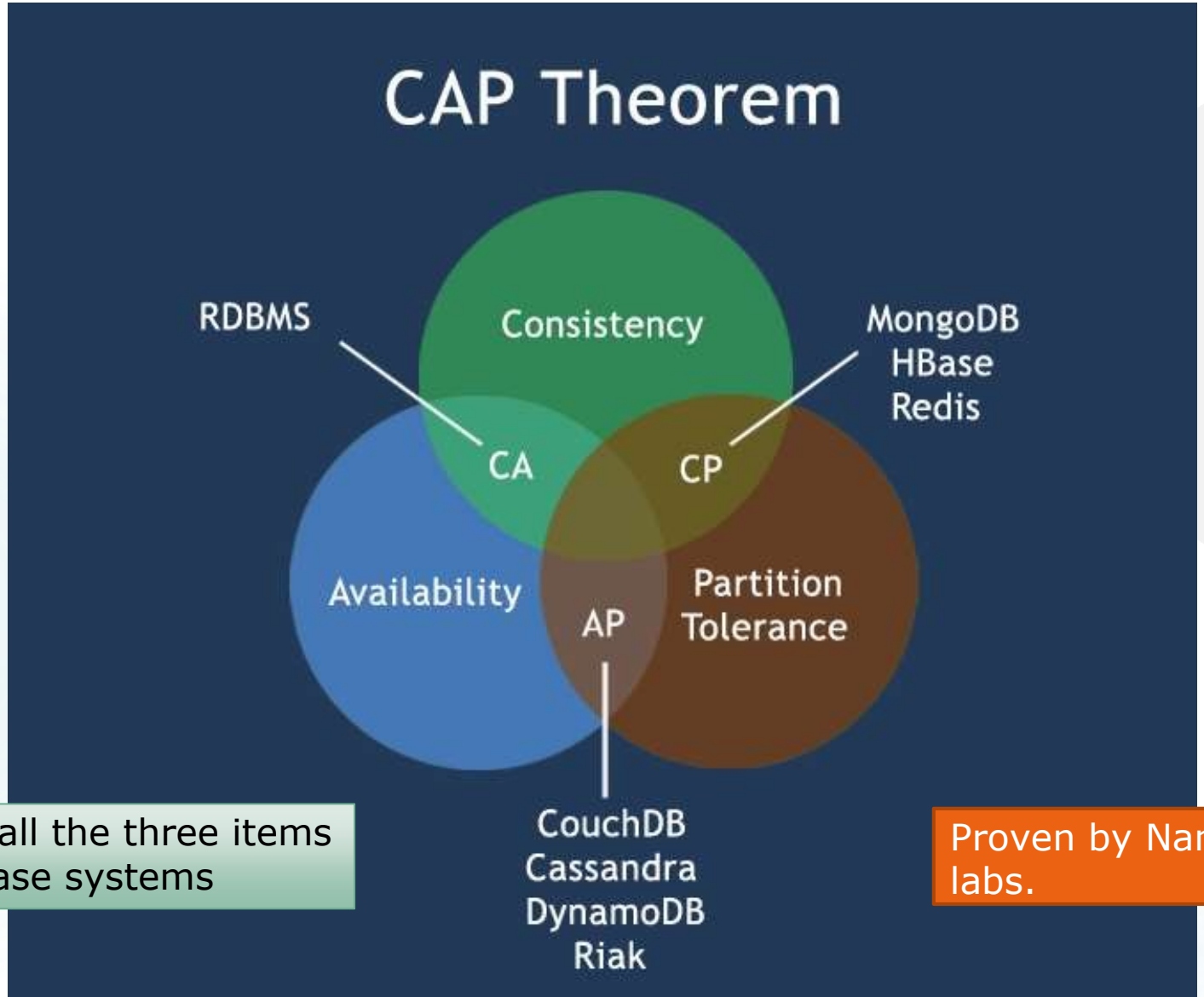
**2002** Seth Gilbert and Nancy Lynch, ACM SIGACT News 33(2)

> *Of three properties of shared-data systems - data **C**onsistency, system **A**vailability and tolerance to network **P**artitions - only two can be achieved at any given moment in time.*

# Which data model to choose



# CAP theorem in different SQL/NoSQL databases

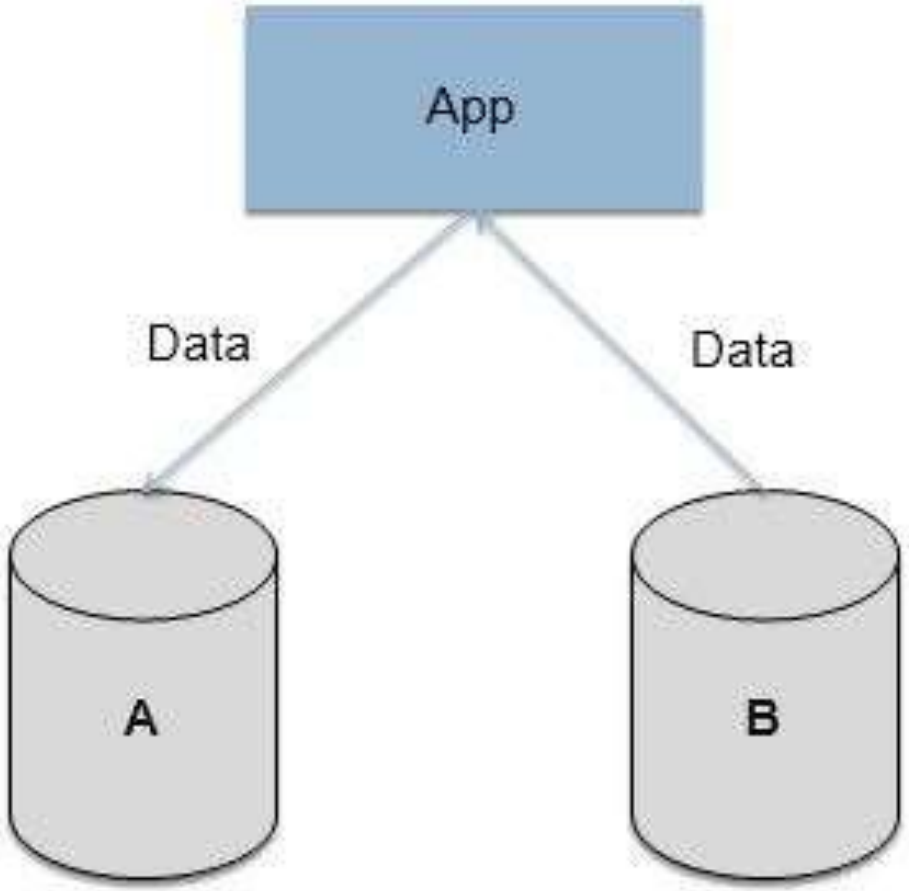



We can not achieve all the three items In distributed database systems (center)

Proven by Nancy Lynch et al. MIT labs.

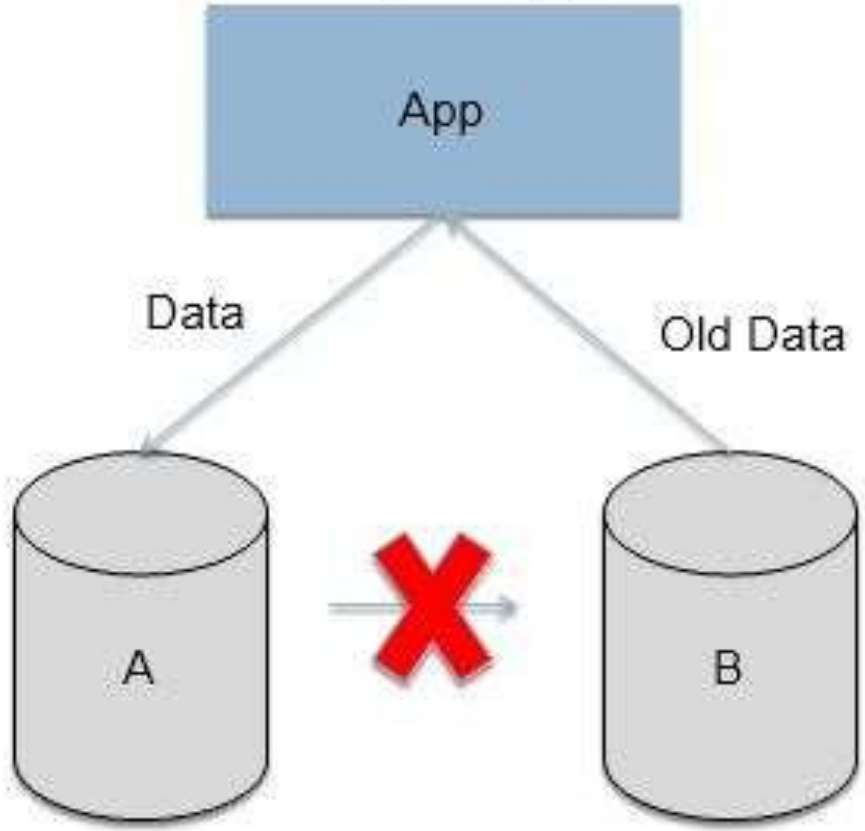
# CAP theorem : A simple proof

Consistent and available
No partition.



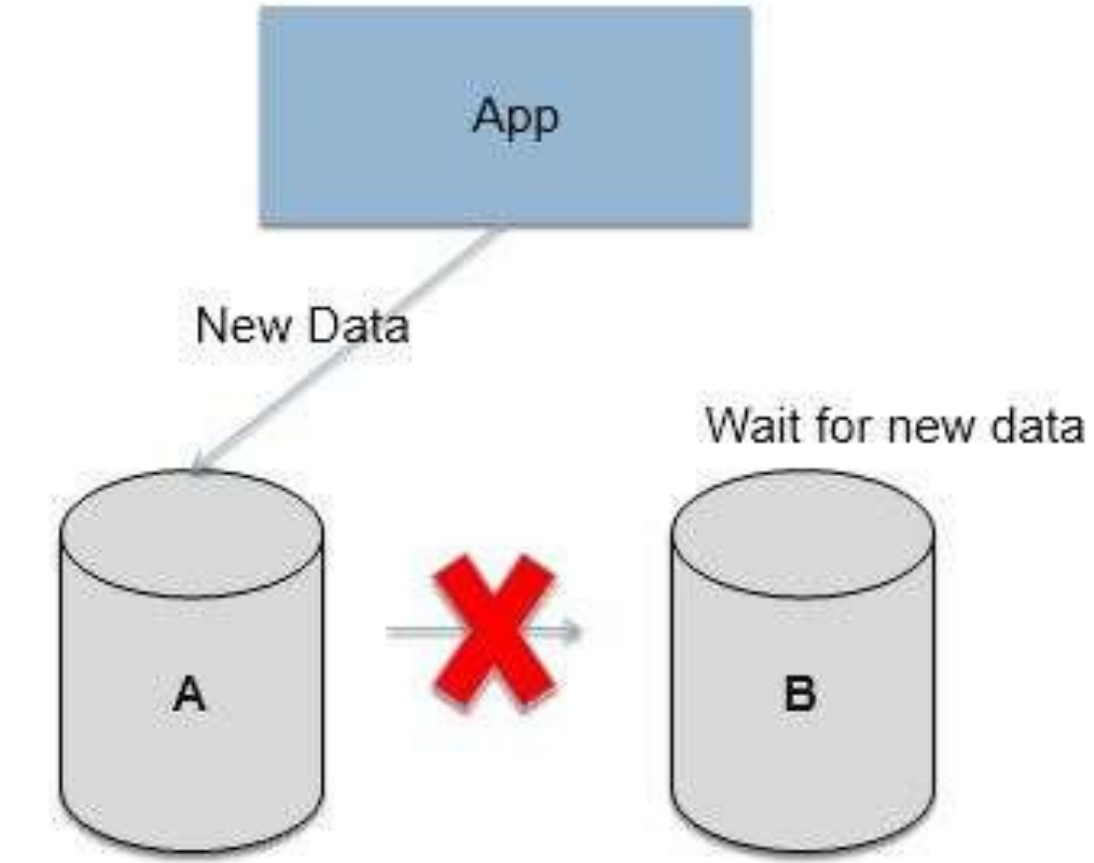
# CAP theorem : A simple proof

Available and partitioned
Not consistent, we get back old data.



# CAP theorem : A simple proof

Consistent and partitioned
Not available, waiting…



# WHEN TO USE A NOSQL DATABASE?

- Large amounts of data
  - Terabytes and Petabytes of data
- Need horizontal scalability
- Need high throughput – fast reads
- Need a flexible schema
  - No fixed number of columns
- Need high availability
- Need to be able to store different data type formats
- Users are distributed – low latency
- Need redundancy in case of failures

# WHEN NOT TO USE A NOSQL DATABASE?

- Need ACID Transactions
- Need ability to do JOINS
- Ability to do aggregations and analytics
- Have changing business requirements
- Queries are not available and need to have flexibility
- Have a small dataset

# Denormalization

# Definition :-

- Denormalization is the process of taking a normalized database and modifying table structures to allow controlled redundancy for increased database performance.

# Method of denormalization:-

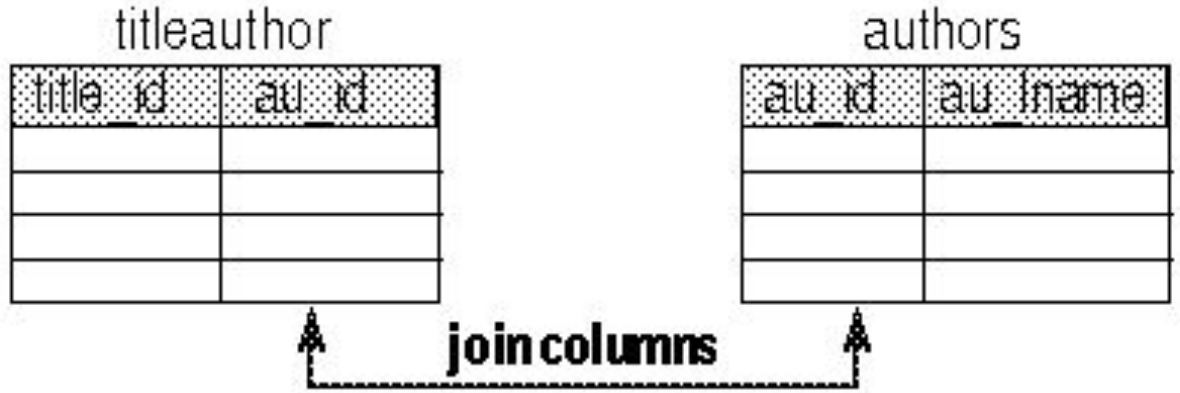
1) Adding Redundant Columns.
2) Adding Derived Columns
3) Combining Tables
4) Repeating Groups
5) Partitioning Relations

# Adding Redundant Columns

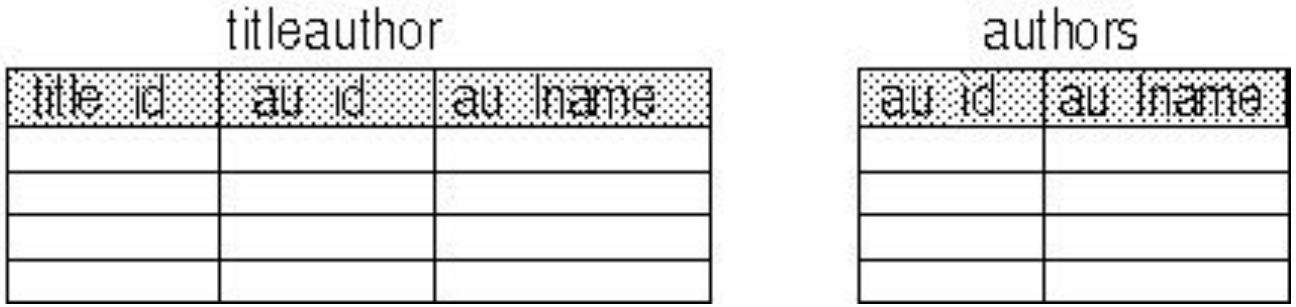
You can add redundant columns to eliminate frequent joins.

**For example**, if frequent joins are performed on the *titleauthor* and *authors* tables in order to retrieve the author's last name, you can add the *au_lname* column to *titleauthor*.

```
select ta.title_id, a.au_id, a.au_lname
from titleauthor ta, authors a
where ta.au_id = a.au_id
```




```
select title_id, au_id, au_lname
from titleauthor
```



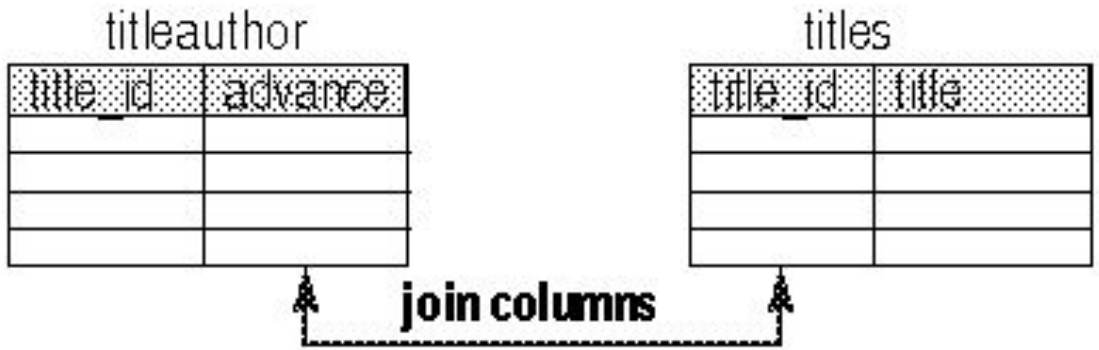
- Adding redundant columns eliminates joins for many queries. The problems with this solution are that it:
- Requires maintenance of new column. All changes must be made to two tables, and possibly to many rows in one of the tables.
- Requires more disk space, since *au_lname* is duplicated.

# Adding Derived Columns

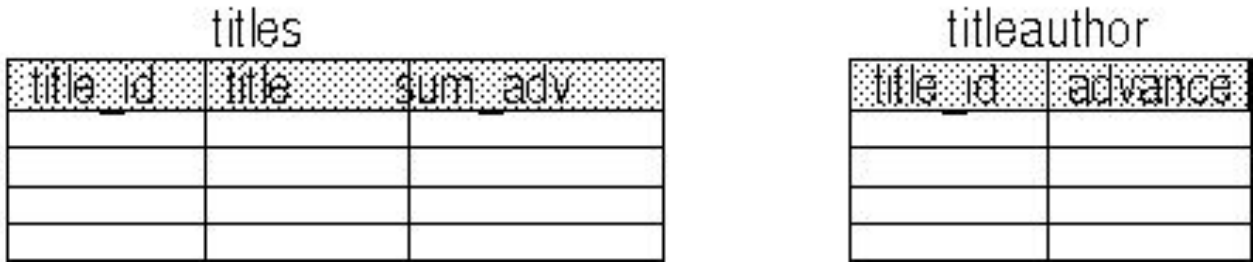
Adding derived columns can help eliminate joins and reduce the time needed to produce aggregate values.

**For example**. Frequent joins are needed between the *titleauthor* and *titles* tables to provide the total advance for a particular book title.

```
select title, sum(advance)
from titleauthor ta, titles t
where ta.title_id = t.title_id
group by title_id
```




```
select title, sum_adv
from titles
```



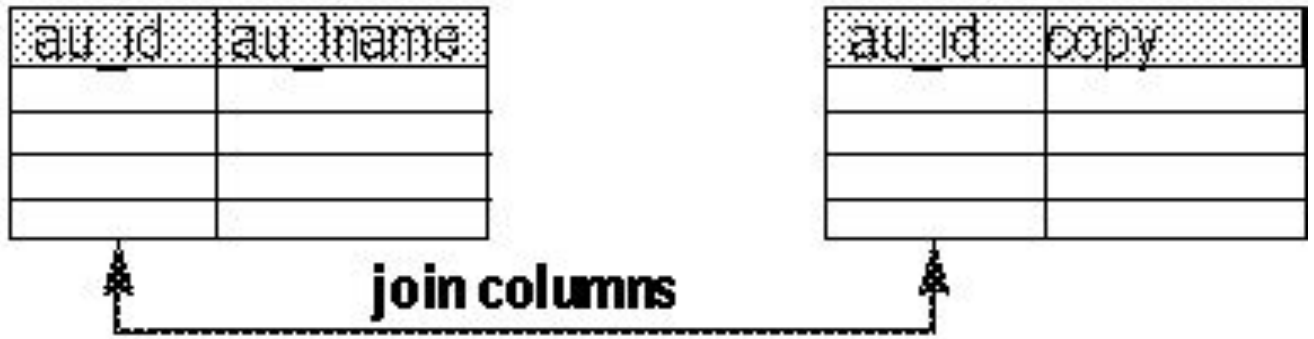
- You can create and maintain a derived data column in the *titles* table, eliminating both the join and the aggregate at run time. This increases storage needs, and requires maintenance of the derived column whenever changes are made to the *titles* table.

# Combining Tables

If most users need to see the full set of joined data from two tables, collapsing the two tables into one can improve performance by eliminating the join.

**For example**, users frequently need to see the author name, author ID, and the *blurbs* copy data at the same time. The solution is to collapse the two tables into one. The data from the two tables must be in a one-to-one relationship to collapse tables.

```
select a.au_id, a.au_lname,
b.copy
from authors a, blurbs b
where a.au_id = b.au_id
```

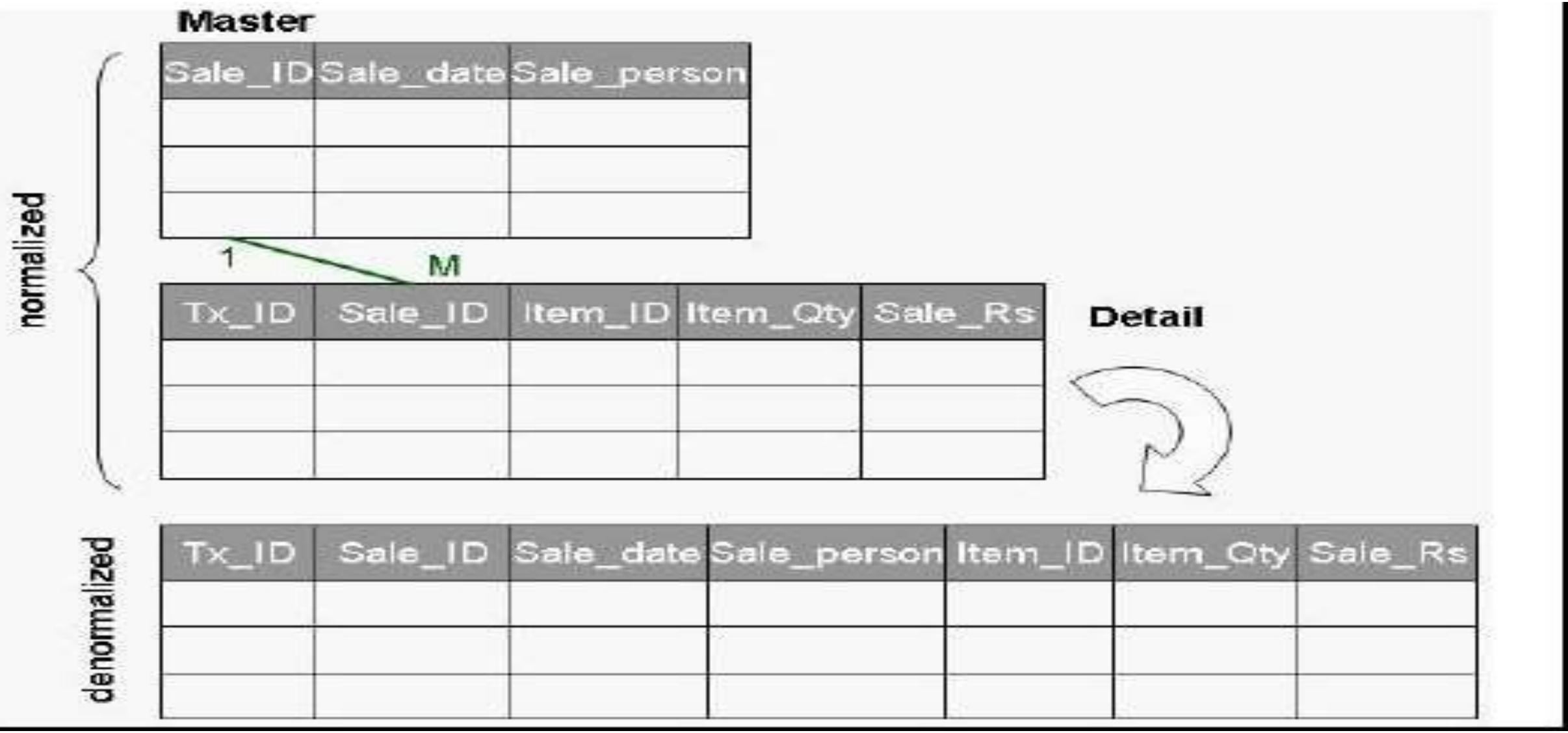



```
select * from newauthors
```

newauthors

| au_id | au_lname | copy |
|---|---|---|
| | | |
| | | |
| | | |
| | | |

# More examples:-



# Q & A

# References


- Database design & Relational theory, C.J.Date
- Database system .8th edition.
- Data Normalization, Denormalization,and the Forces of Darkness  a white paper by Melissa Hollingsworth.
- NoSQL distilled, Martin Fowler
- Martin Fowler’s presentation at Goto conference
- www.mongodb.org
- Pooyan Mehrparvar