# Beyond Relational Databases: MongoDB, Redis & ClickHouse

Marcos Albe - Principal Support Engineer @ Percona

# Introduction

MySQL everyone?

# Introduction

Redis?

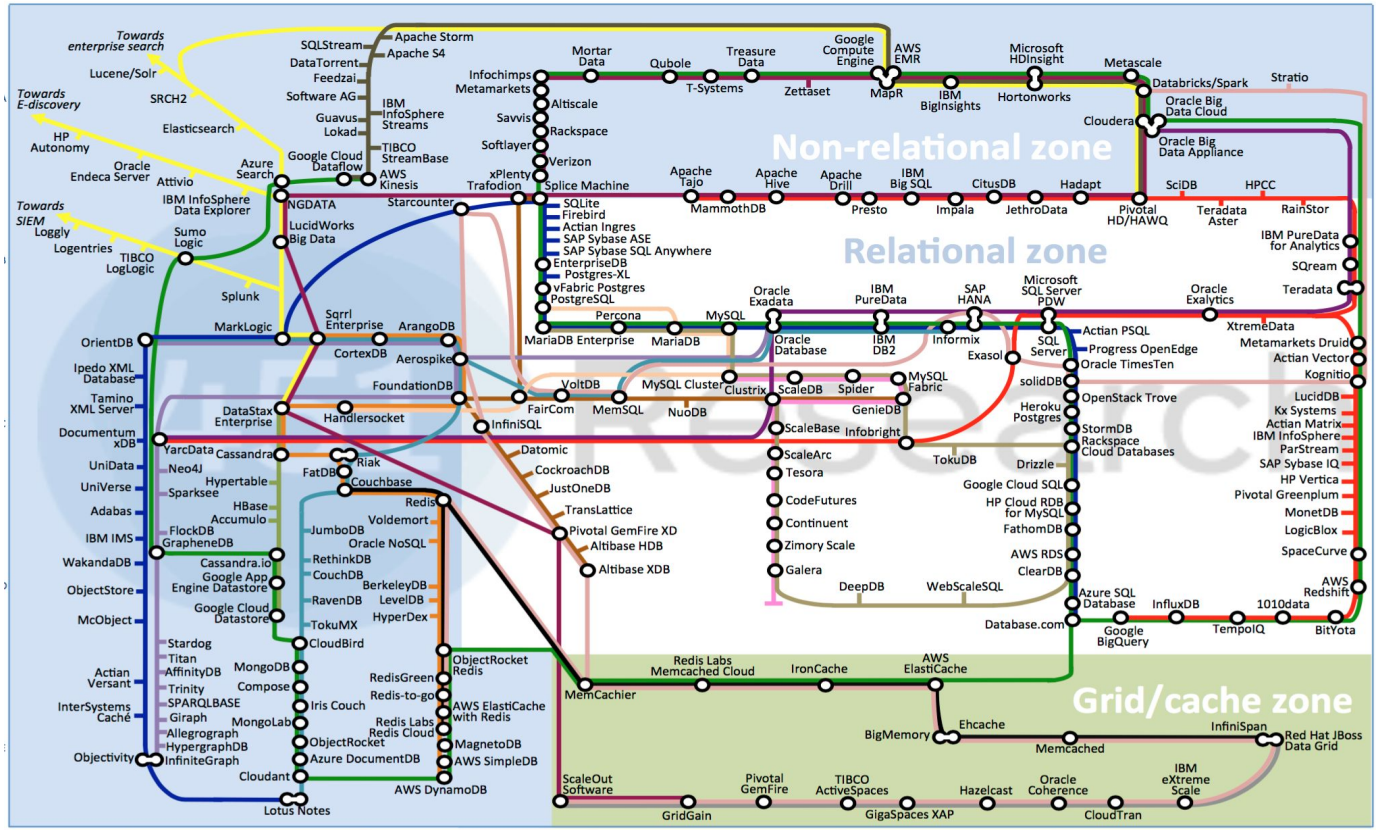



Image credits: 451 Research (https://451research.com/state-of-the-database-landscape)

# Agenda

- **Introduction**
- Why not Relational Databases?
- Redis: Key-Value
- MongoDB: Document
- ClickHouse: Columnar

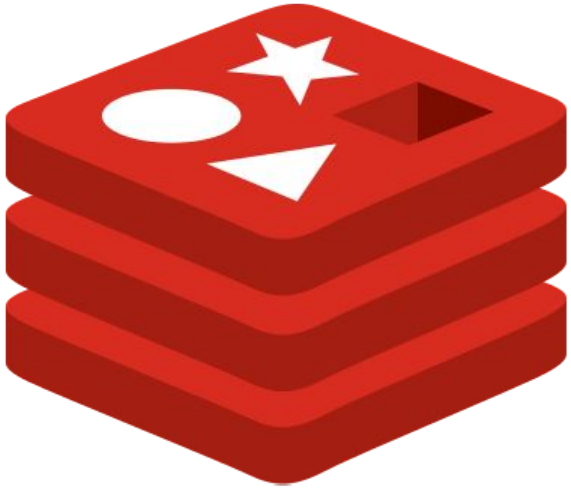
# Why **not** Relational Databases?

# Why not Relational Databases?

- General purpose but not optimal for all purposes
- Impedance mismatch with developers
- ACID/locking induced latency
- Storage architecture not optimal for big data

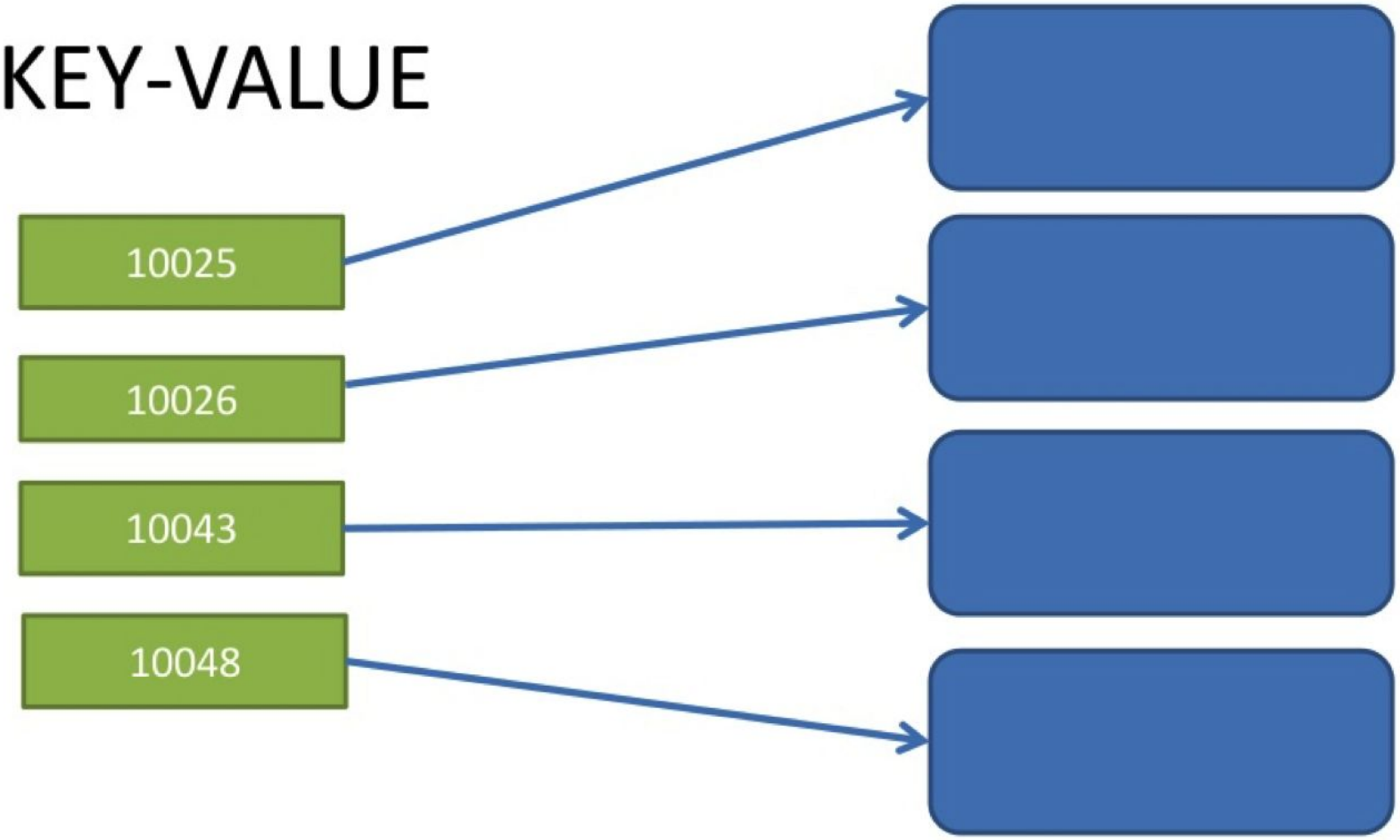
# Agenda

- Introduction
- Why not Relational Databases?
- **Redis: Key-Value**
- MongoDB: Document
- ClickHouse: Columnar



redis

# KEY-VALUE



# Redis highlights

- Fast… very fast.
- Mature / large community
- Many data structures
- Advanced features
- Horizontally scalable / built-in HA (Sentinel / Cluster)
- BSD license / Commercial licenses available
- Client libraries for about every programming language

# Redis data types

- Lists
- Sets
- Sorted sets
- Hashes
- Bitmaps
- Geohash

# Redis good use cases

- Lots of data
- High concurrency
- Massive small-data intake
- Simple data access patterns
- Session Cache / Full page cache
- Counters / Leaderboards
- Queues

# Redis bad Use Cases

- Durability and consistency
- Complex data access patterns
- Non-PK access
- Security concerns
- Operations

# PROs

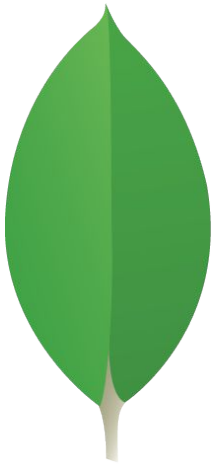
- Fast
- Highly scalable/available
- Simple access patterns
- Advanced data types
- Advanced features for KV store
    - Atomic operations
- Ubiquitous / large community

# CONs

- Lower durability
- Operational complexities
- Limited access patterns
- Lack of security
- No secondary keys

# Agenda

- Introduction
- Why not Relational Databases?
- Redis: Key-Value
- **MongoDB: Document**
- ClickHouse: Columnar



mongoDB®

# MongoDB Flexible Schema

```
mysql> select * from user \G
*************************** 1. row ****
       id: 1
    email: me@percona.com
birthdate: 2006-11-01 00:00:00
1 row in set (0.00 sec)
```

```
replset:PRIMARY> db.user.find().pretty()
{
        "_id" : ObjectId("57130f3f3a25efe746fda544"),
        "email" : "me@percona.com",
        "birthdate" : ISODate("2006-11-01T00:00:00Z")
}
replset:PRIMARY>
```

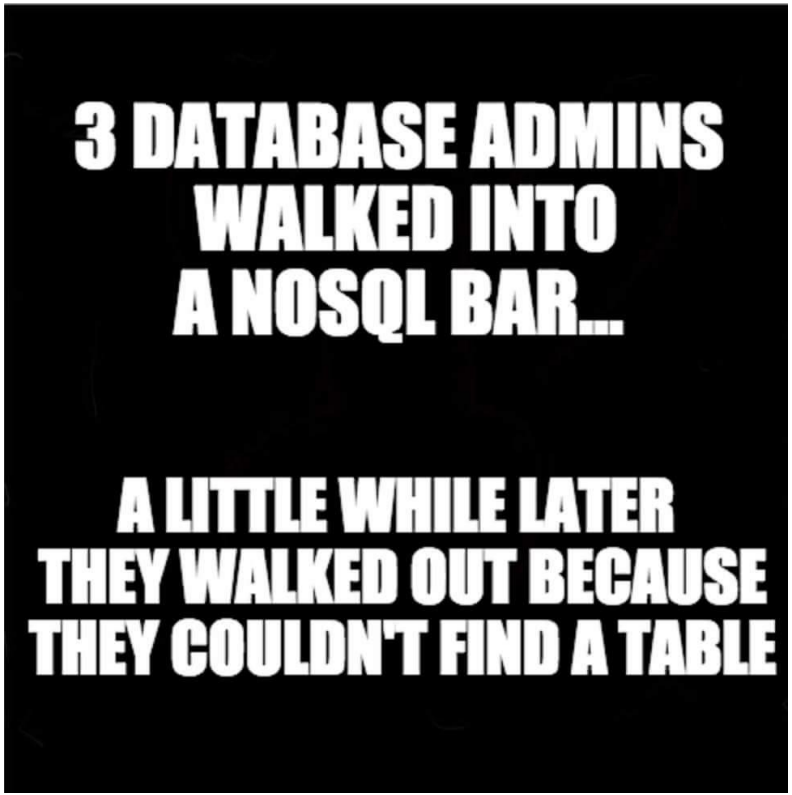
# MongoDB Flexible Schema

```
mysql> select user_id, email, birthdate, site
    -> from user
    -> inner join sites on (user.id = sites.user_id) \G
*************************** 1. row ***************************
  user_id: 1
    email: me@percona.com
birthdate: 2006-11-01 00:00:00
     site: www.google.com
*************************** 2. row ***************************
  user_id: 1
    email: me@percona.com
birthdate: 2006-11-01 00:00:00
     site: www.percona.com
2 rows in set (0.01 sec)
```

```
replset:PRIMARY> db.user.find().pretty()
{
        "_id" : ObjectId("57130f3f3a25efe746fda544"),
        "email" : "me@percona.com",
        "birthdate" : ISODate("2006-11-01T00:00:00Z"),
        "sites" : [
                "www.google.com",
                "www.percona.com"
        ]
}
replset:PRIMARY>
```

# MongoDB Flexible Schema

- Embedded in blogpost document (natural case)
- Embedded in user document (bad idea)
- Denormalize and keep duplicate data (VERY bad idea)
- Apply normalization and user lookup() (JOIN equivalent)


3 DATABASE ADMINS
WALKED INTO
A NOSQL BAR...
A LITTLE WHILE LATER
THEY WALKED OUT BECAUSE
THEY COULDN'T FIND A TABLE

# Document Stores: Flexible Schema

```
mysql> select * from user \G
*************************** 1. row ***************************
                    id: 1
               user_id: 0
                  site: 2006-11-01 00:00:00
             is_active: 1
     accepts_newletters: 1
          email_format: 1
preferred_payment_method: 1
               is_more: 1
*************************** 2. row ***************************
                    id: 2
               user_id: 0
                  site: 2006-11-01 00:00:00
             is_active: 1
     accepts_newletters: NULL
          email_format: NULL
preferred_payment_method: NULL
               is_more: NULL
2 rows in set (0.00 sec)
```

# Document Stores: Flexible Schema

```
replset:PRIMARY> db.user.find().pretty()
{
        "_id" : ObjectId("5713159e3a25efe746fda546"),
        "email" : "me@percona.com",
        "birthdate" : ISODate("2006-11-01T00:00:00Z"),
        "is_active" : 1,
        "accepts_newletters" : 1,
        "email_format" : 1,
        "preferred_payment_method" : 1,
        "is_more" : 1
}
{
        "_id" : ObjectId("571315b53a25efe746fda547"),
        "email" : "you@percona.com",
        "birthdate" : ISODate("2006-11-01T00:00:00Z"),
        "is_active" : 1
}
replset:PRIMARY>
```

# MongoDB highlights

- Sharding and replication for dummies!
- Flexible schema
- Multi-document transactions (new in 4.0)
- Pluggable storage engines for distinct workloads.
- Excellent compression options with PerconaFT, RocksDB, WiredTiger
- On disk encryption (Enterprise Advanced)
- Connectors for all major programming languages
- Sharding and replica aware connectors
- Geospatial functions
- Aggregation framework

# MongoDB good use cases

- Catalogs
- Analytics/BI (BI Connector on 3.2)
- Time series
- Metadata repositories
- Prototype Development

# MongoDB bad use cases

- Recursiveness
- Multiple views of the data
- Developer comfort

# PROs

- Fast
- Easy sharding
- Simple access patterns
- Rich feature set
- Async connectors
- Flexible schema

# CONs

- Inefficiency for JOINs
- Still immature internally
- Attribute names bloat space
- MMAP db-level locking

# Agenda

- Introduction
- Why not Relational Databases?
- Redis: Key-Value
- MongoDB: Document
- **ClickHouse: Columnar**

ClickHouse

# Columnar Data Layout

Row-oriented

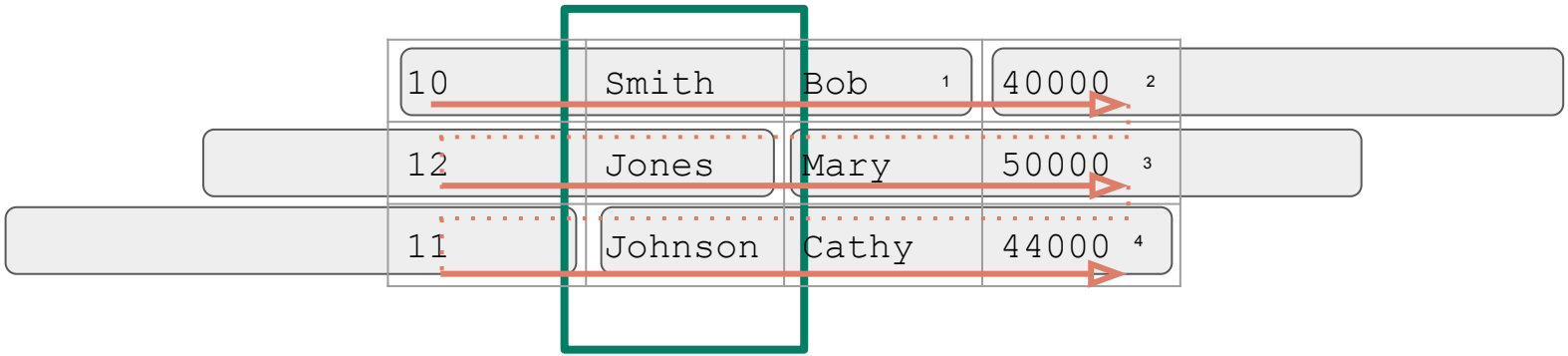
```
001:10,Smith,Joe,40000;
002:12,Jones,Mary,50000;
003:11,Johnson,Cathy,44000;
004:22,Jones,Bob,55000;
...
```

Column-oriented

```
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;
...
```

# Columnar Data Layout

**Row-oriented** Read Approach



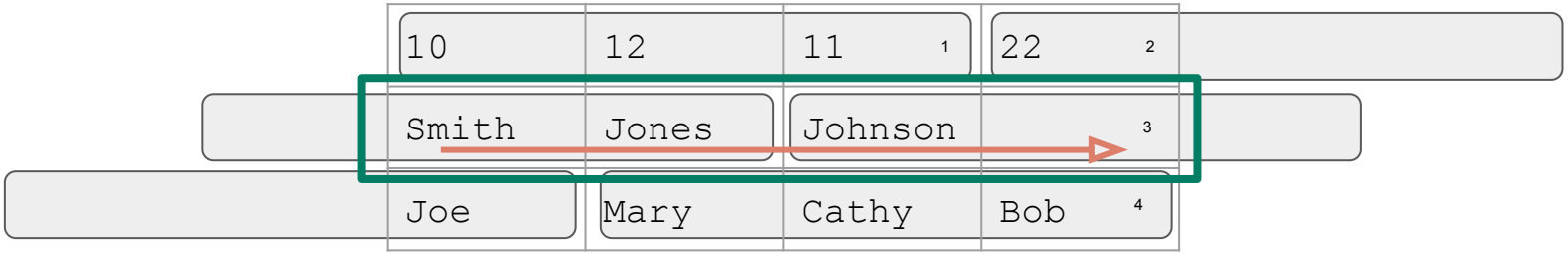

Memory Page

What we want to read

Read Operation

# Columnar Data Layout

**Column-oriented** Read Approach




Memory Page

What we want to read

Read Operation

# ClickHouse: columnar store

```
:) select count(*) from dw.ad8_fact_event;

SELECT count(*)
FROM dw.ad8_fact_event

┌─────count()─┐
│ 900627883648 │
└──────────────┘

1 rows in set. Elapsed: 3.967 sec. Processed 900.65 billion rows,
900.65 GB (227.03 billion rows/s., 227.03 GB/s.)
```

# ClickHouse highlights

- True column oriented
- Compression
- Disk based (not in-memory)
- Parallelized
- Distributed
- SQL querying;
  - Improved JOINs in latter versions
- Vector engine
- Real-time updates

# ClickHouse highglights

- Indexes
- Dictionaries
- Online queries
- On-disk GROUP BYs
- Approximated calculations
- Replication
- Few, but popular connectors
- Support for Tableau

# ClickHouse limitations

- Non-standard SQL
- UPDATE/DELETE is a bulk operation (ALTER … UPDATE)
- No windowing functions
- No transactions or constraints
- Eventual consistency
- Time data types have no milliseconds
- No implicit type conversions
- Impossible to do table partitioning (except by month)
- Lack of enterprise operation tools

# ClickHouse good use cases

- Suitable for read-mostly or read-intensive, large data repositories
- Full table / large range reads.
- Time series data
- Unstructured problems where “good” indexes are hard to forecast
- Log analysis
- Re-creatable datasets

# ClickHouse bad use cases

- Not good for “SELECT *” queries or queries fetching most of the columns
- Not good for "small"  writes; Good for bulk writes
- Not good for mixed read/write; Focus on large reads and large inserts
- Bad for unstructured data

# PROs

- VERY fast
- Distributed
- Compression
- Primary-keys
- On-disk GROUP BY

# CONs

- Non-standard SQL
- Few connectors
- Driven mostly by Yandex needs
- No partitioning