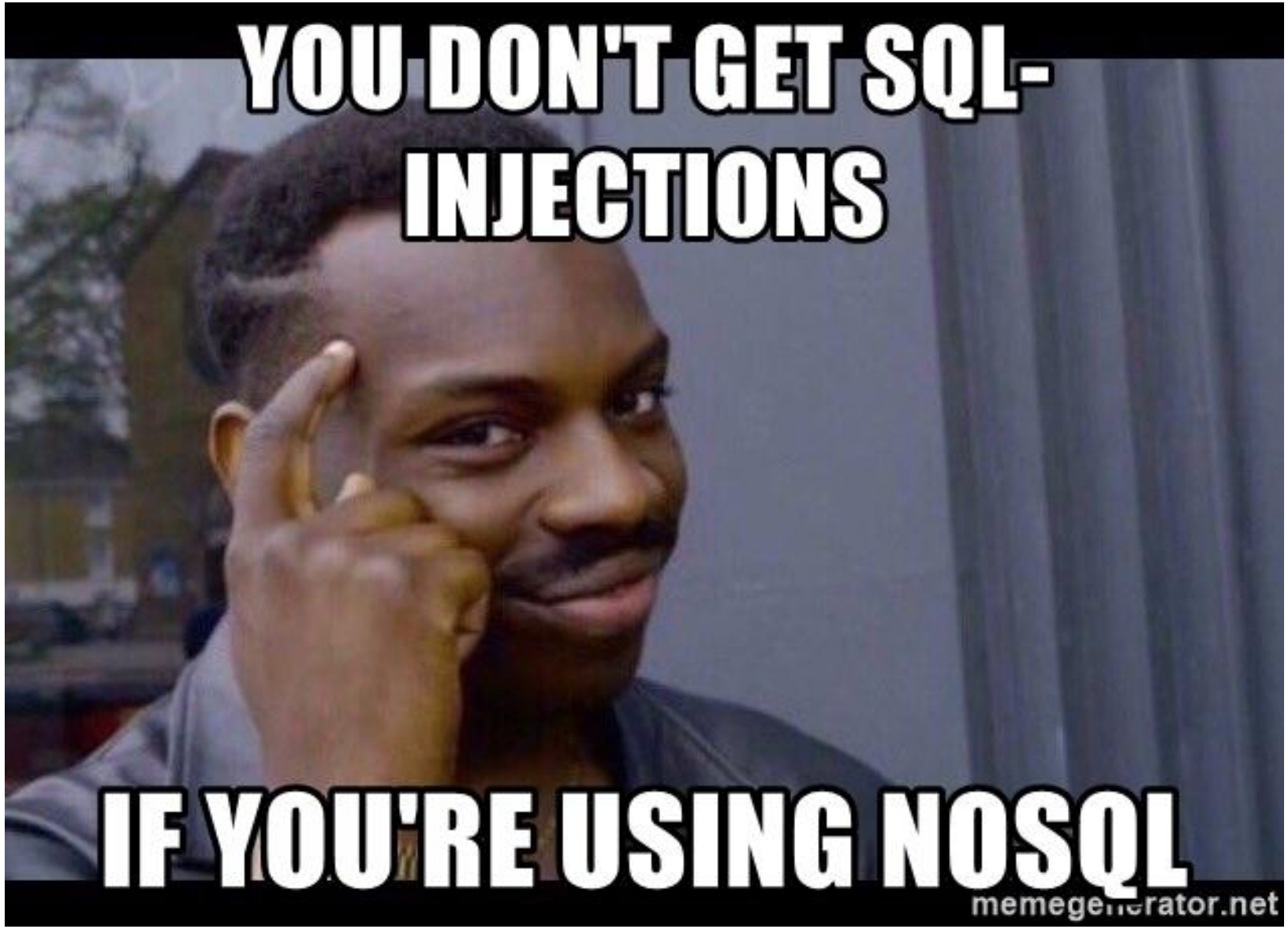
NoSql

Part 1: NOSQL overview

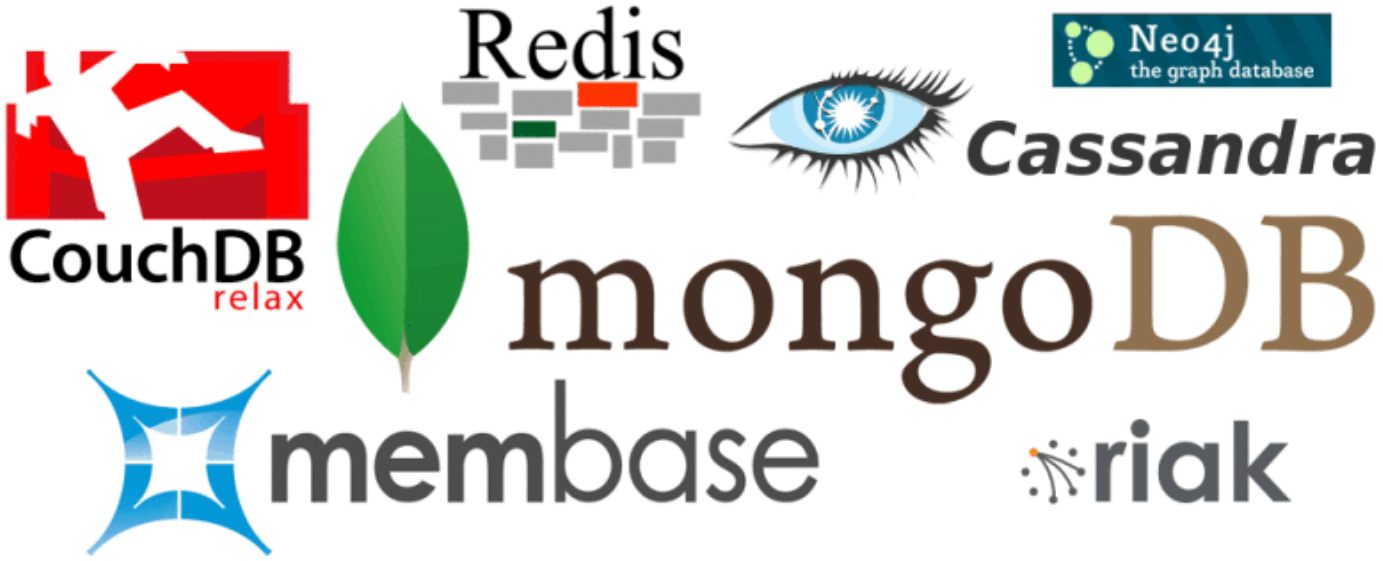
NOSQL vs SQL

	SQL Databases	NOSQL Databases
Schemas	Structure and data types are fixed in advance.	Typically dynamic, with some enforcing data validation rules. Applications can add new fields on the fly, and unlike SQL table rows, dissimilar data can be stored together as necessary.
Scaling	Mostly vertically	Horizontally, meaning that to add capacity, a database administrator can simply add more commodity servers or cloud instances. The database automatically spreads data across servers as necessary.
Supports multi-record ACID transactions	Yes	Mostly no. MongoDB 4.0, scheduled for Summer 2018*, will add multi-document transactions
Data Manipulation	Specific language using Select, Insert, and Update statements, e.g. SELECT fields FROM table WHERE…	Through object-oriented APIs


YOU DON'T GET SQL-INJECTIONS
IF YOU'RE USING NOSQL
memegenerator.net

NoSql Features

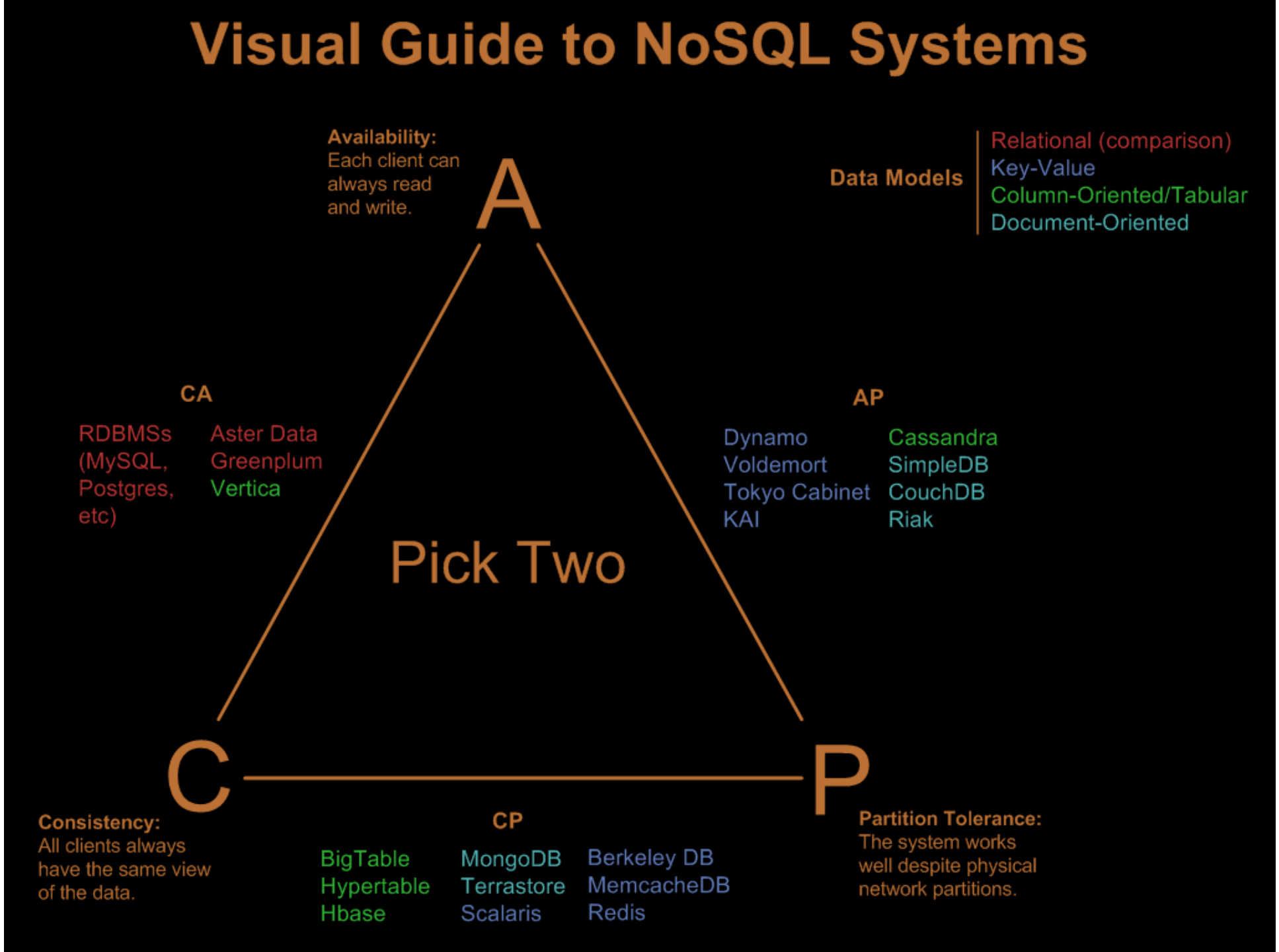
- **Multi-Model**
- **Easily Scalable**
- Flexible
- Distributed
- Zero downtime



NoSql database types

- Key-Value Store
- Document-based Store
- Column-based Store
- Graph-based
- *Multi-model

Visual Guide to NoSQL Systems



Questions

1. What are the main differences between NOSQL & RDBMS Databases?
2. List all NOSQL database types.

Part 2: Key-Value Database

Overview

- Dictionary structure
- Key is string
- Value can be any kind, stored in blob*
- different types: RAM, persistent, eventually consistent

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

* Blob - A Binary Large Object is a collection of binary data stored as a single entity in a database management system.


redis

Data Types

The following types can be assigned to key:

- Binary-safe strings
- Lists
- Sets
- Sorted sets
- Hashes

Redis naming

Name	Description
RedisKey	The Redis key
Value	The string value associated with the RedisKey and ValueIndex
ValueIndex	Varies by type; 1 for strings; the one-based index for sets, lists, and sorted sets; or the associated field name for hashes
RedisType	The Redis data type
ValueScore	Varies by type; *NULL* for strings, lists, sets, and hashes; or the associated score for sorted sets

Creating string

Query:

```
> set mykey somevalue
OK
```

Result:

RedisKey	ValueIndex	Value	RedisType	ValueScore
mykey	1	somevalue	String	NULL

Creating set

Query:

```
> sadd myset 1 2 3
(integer) 3
```

Result:

RedisKey	ValueIndex	Value	RedisType	ValueScore
myset	1	2	Set	NULL
myset	2	1	Set	NULL
myset	3	3	Set	NULL

Creating Zset (sorted set)

Query:

```
> zadd hackers 1940 "Alan Kay" 1957 "Sophie Wilson" 1953 "Richard Stallman" 1949 "Anita Borg"
(integer) 9
```

Result:

RedisKey	ValueIndex	Value	RedisType	ValueScore
hackers	1	Alan Kay	ZSet	1940
hackers	2	Anita Borg	ZSet	1949
hackers	3	Richard Stallman	ZSet	1953
hackers	4	Sophie Wilson	ZSet	1957

Select

```
keys pattern              # Find key matching exactly
keys pattern*             # Find keys matching in back
keys *pattern*            # Find keys matching somewhere
keys pattern*             # Find keys matching in front
```

```
get <key>
set <key> <value>
setnx <key> <value>       # Set key value only if key does not exist
```

Redis persistence

- RDB Mechanism
- AOF
- SAVE Command

Snapshots

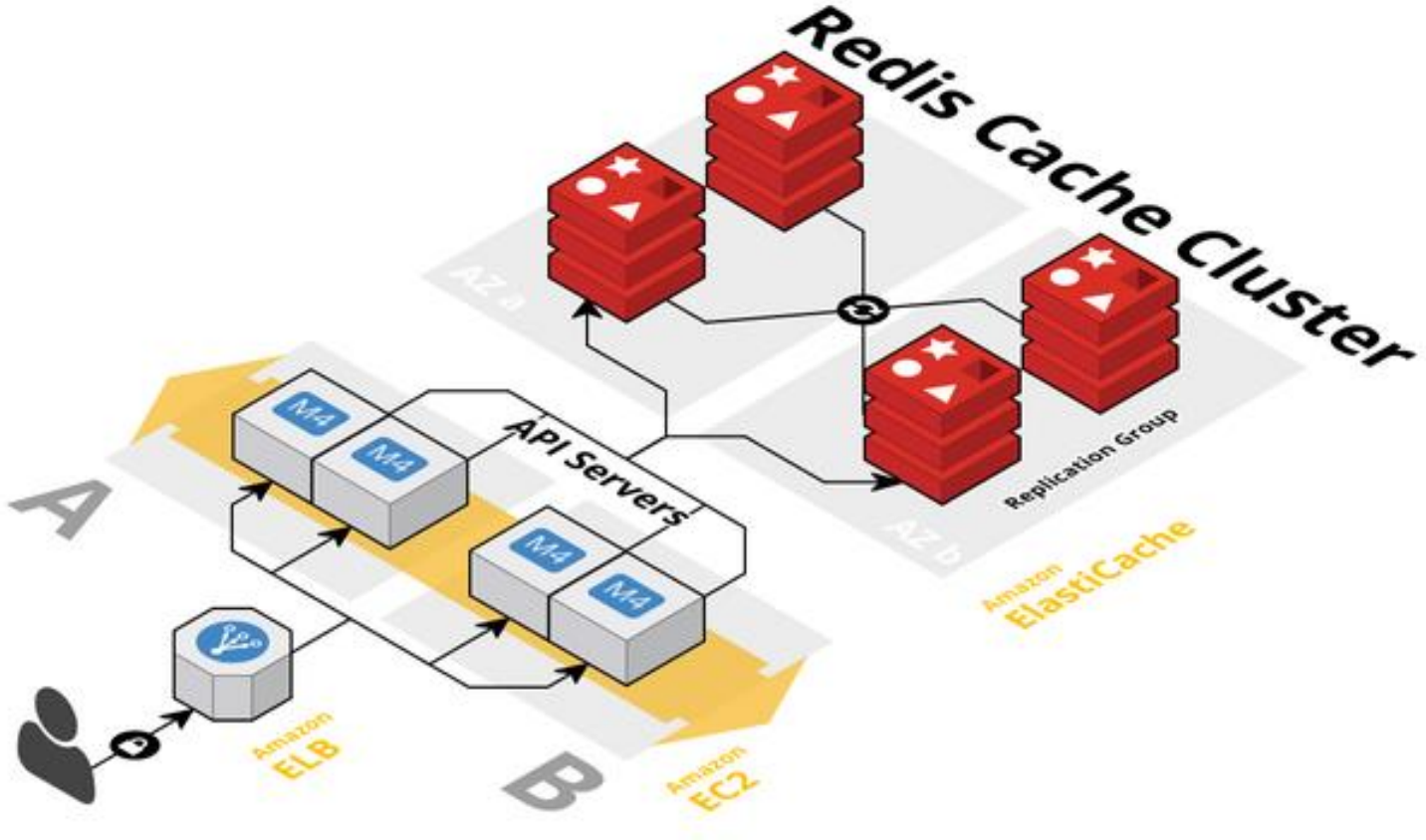
Special interval

AOF

Each time new data arrived

Other features

- Everything stored in RAM
- Replication and clustering out of box



Use cases

- Session management at high scale
- User preference and profile stores
- Product recommendations; latest items viewed on a retailer website drive future customer product recommendations
- Ad servicing; customer shopping habits result in customized ads, coupons, etc. for each customer in real-time
- Can effectively work as a cache for heavily accessed but rarely updated data

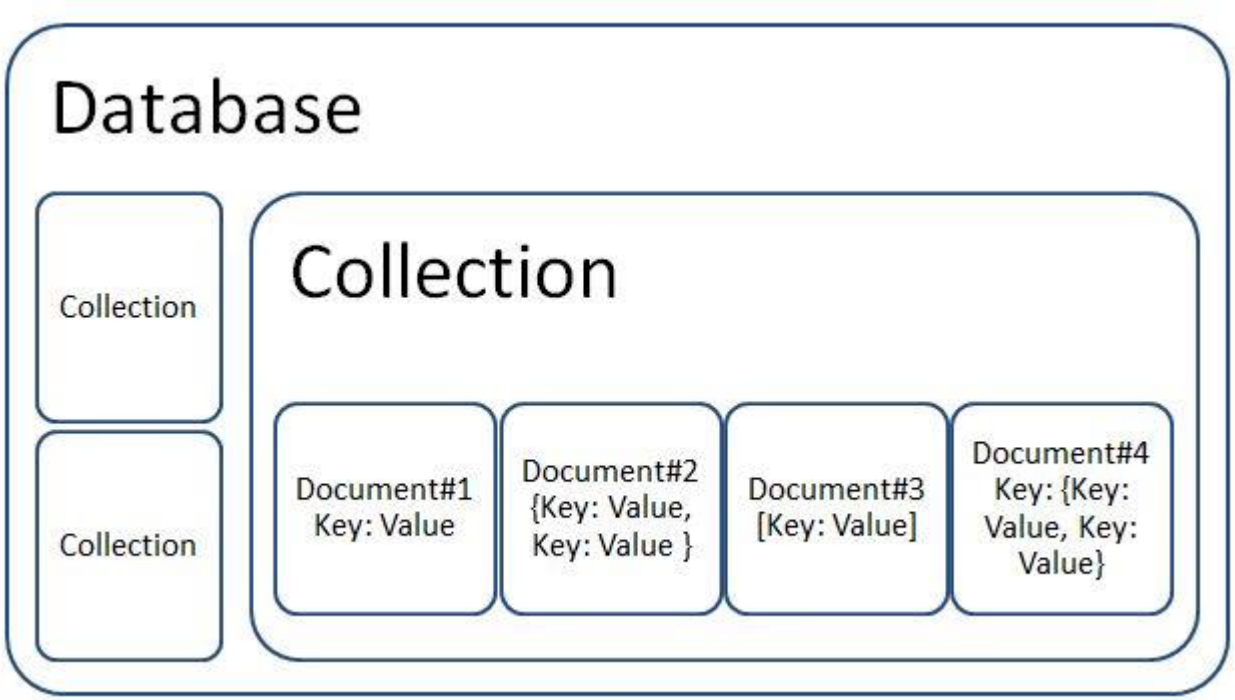
Questions

1. How to persist Redis data?
2. What are the use cases for key-value databases?

Part 3: Document Databases

Overview

- Document may refer to a Microsoft Word or PDF document but is commonly a block of XML or JSON.
- Document contains a description of the data type and the value for that description.
- Each document can have the same or different structure.



Document Databases: Mongo DB

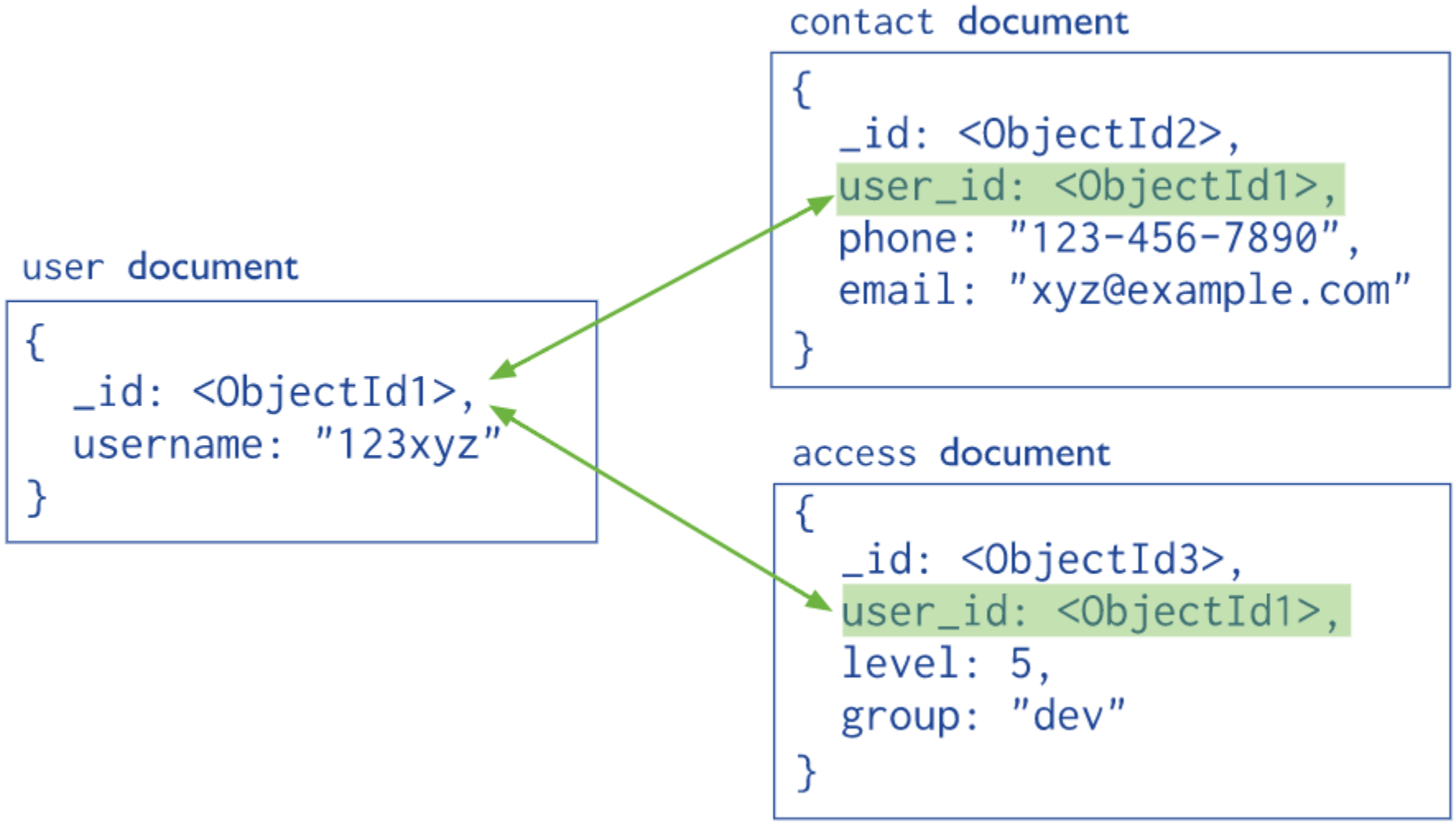
```
{
   _id: ObjectId(7df78ad8902c)
   title: 'MongoDB Overview',
   description: 'MongoDB is no sql database',
   by: 'tutorials point',
   url: 'http://www.tutorialspoint.com',
   tags: ['mongodb', 'database', 'NoSQL'],
   likes: 100,
   comments: [
      {
         user:'user1',
         message: 'My first comment',
         dateCreated: new Date(2011,1,20,2,15),
         like: 0
      },
      {
         user:'user2',
         message: 'My second comments',
         dateCreated: new Date(2011,1,25,7,45),
         like: 5
      }
   ]
}
```

Document

MongoDB Naming

RDBMS	MongoDB
Database	Database
Table	Collection
Tuple/Row	Document
column	Field
Table Join	Embedded Documents
Primary Key	Primary Key (Default key _id provided by mongodb itself)

MongoDB References



MongoDB Embedded Data

```
{
  _id: <ObjectId1>,
  username: "123xyz",
  contact: {
              phone: "123-456-7890",
              email: "xyz@example.com"
            },
  access: {
              level: 5,
              group: "dev"
            }
}
```

Embedded sub-document

Embedded sub-document

Queries: create db & insert data

```
> use crashcoursedb
switched to db crashcoursedb
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
> db.movie.insert({"name": "crash course"})
WriteResult({ "nInserted" : 1 })
> show dbs
admin          0.000GB
config         0.000GB
crashcoursedb  0.000GB
local          0.000GB
>
```

Select

MongoDB	RDBMS
db.movie.find({})	SELECT * FROM movie
db.movie.find({name : “…”})	SELECT * FROM movie WHERE name =“…”
db.movie.find({name: $in :[“…”, “…”, “…”] } })	SELECT * FROM moview WHERE name in (“…”, “…”, “…”)
db.movie.find({ $or: [{ name: “…”}, {time : {$lt : 160 } }] })	SELECT * FROM movie WHERE name=“…” OR time < 160
db.movie.find({filmed: “2018”, $or: [{ name: “…”}, {time : {$lt : 160 } }] })	SELECT * FROM movie WHERE filmed=“2018” AND (name=“…” OR time < 160)

Select on embedded documents [1]

```
db.inventory.find( { size: { h: 14, w: 21, uom: "cm" } } )
```

```
db.inventory.find(  { size: { w: 21, h: 14, uom: "cm" } }  )
```

What’s the difference between these two queries?

Select on embedded documents [2]

```
db.inventory.find( { "size.uom": "in" } )
```

```
db.inventory.find( { "size.h": { $lt: 15 }, "size.uom": "in", status: "D" } )
```

Update

```
db.restaurant.updateOne(
    { "name" : "Central Perk Cafe" },
    { $set: { "violations" : 3 } }
);
```

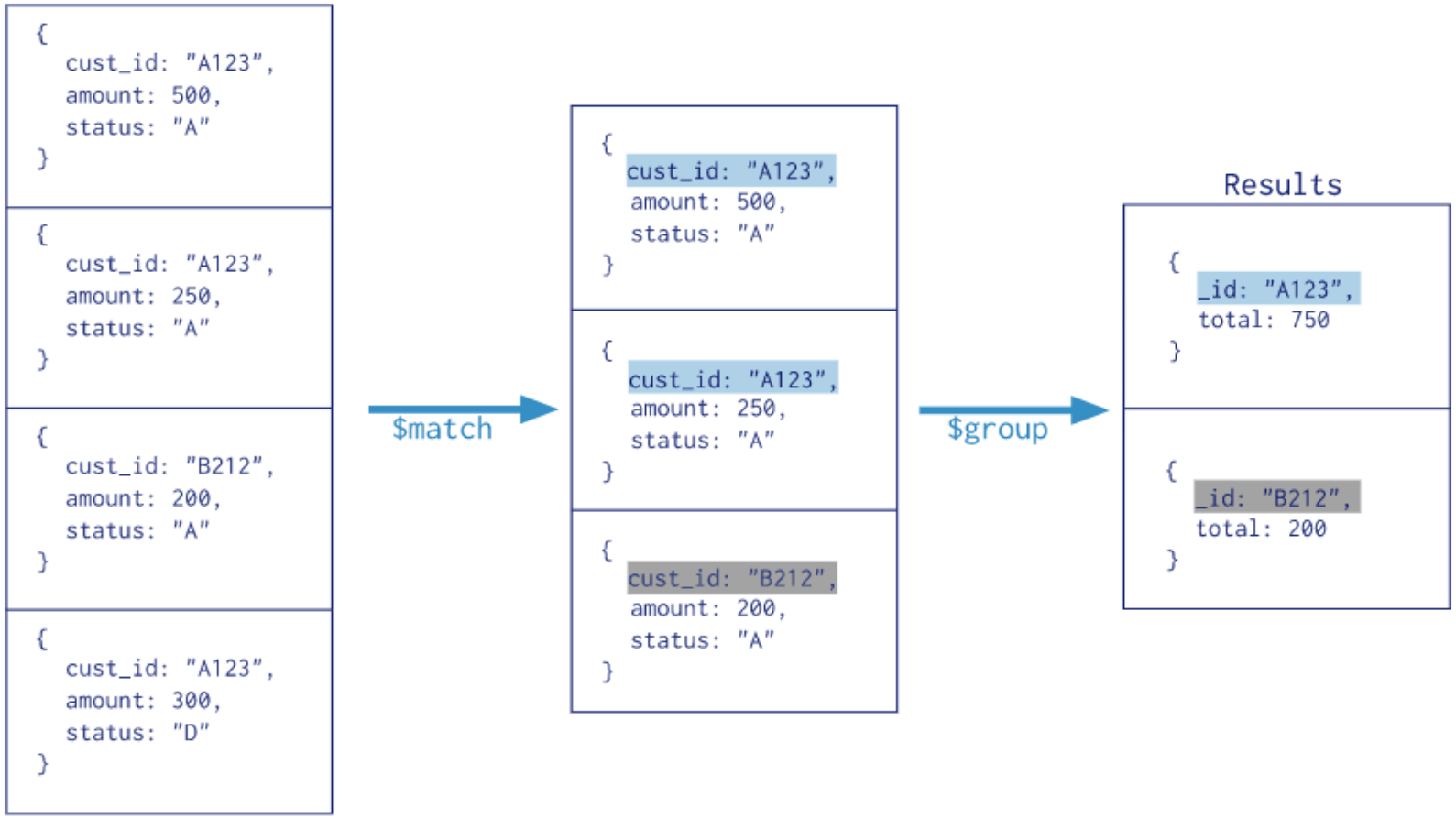
Delete

- db.inventory.deleteMany({ status : "A" })

- db.inventory.deleteOne({ status: "D" })

Aggregation

```
db.orders.aggregate( [
     $match stage ──►   { $match: { status: "A" } },
     $group stage ──►   { $group: { _id: "$cust_id",total: { $sum: "$amount" } } }
                    ] )
```



Questions

- Explain how the data is stored in document databases. You many take MongoDB as example.
- How to select embedded documents and what is important to know?

Part 4: Column-based Databases

Overview

- All data for specific ‘table’ is stored in columns
- Fast data selection among bit dataset
- rowId is used to select further data from other columns
- Data type in column is not constrained, null data does not take place

Sales

Product	Customer	Date	Sale
Beer	Thomas	2011-11-25	2 GBP
Beer	Thomas	2011-11-25	2 GBP
Vodka	Thomas	2011-11-25	10 GBP
Whiskey	Christian	2011-11-25	5 GBP
Whiskey	Christian	2011-11-25	5 GBP
Vodka	Alexei	2011-11-25	10 GBP
Vodka	Alexei	2011-11-25	10 GBP

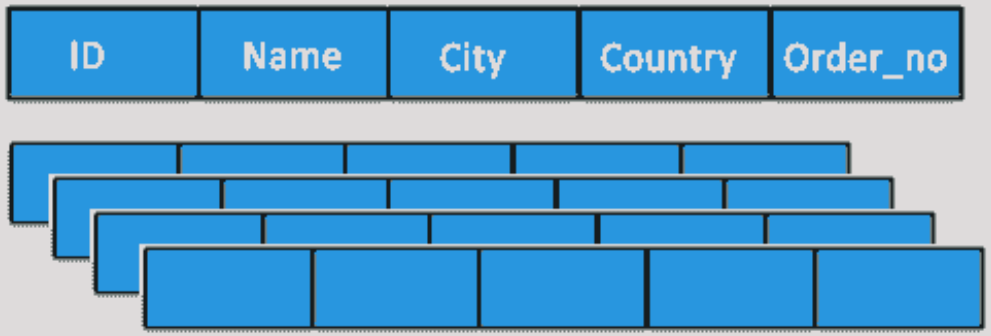
Product

ID	Value
1	Beer
2	Beer
3	Vodka
4	Whiskey
5	Whiskey
6	Vodka
7	Vodka

Customer

ID	Customer
1	Thomas
2	Thomas
3	Thomas
4	Christian
5	Christian
6	Alexei
7	Alexei

row-store



+ easy to add/modify a record

- might read in unnecessary data

column-store

+ only need to read in relevant data

- tuple writes require multiple accesses

=>suitable for read-mostly, read-intensive, large data repositories


Cassandra

Cassandra vs RDBMS

Relational Database	**Cassandra**
Manages primarily structured data	Manages all types of data
Supports complex/nested transactions	Supports simple transactions
Single points of failure with failover	No single points of failure; constant uptime
Centralized deployments	Decentralized deployments
Data written in mostly one location	Data written in many locations
Supports read scalability (with consistency sacrifices)	Supports read and write scalability

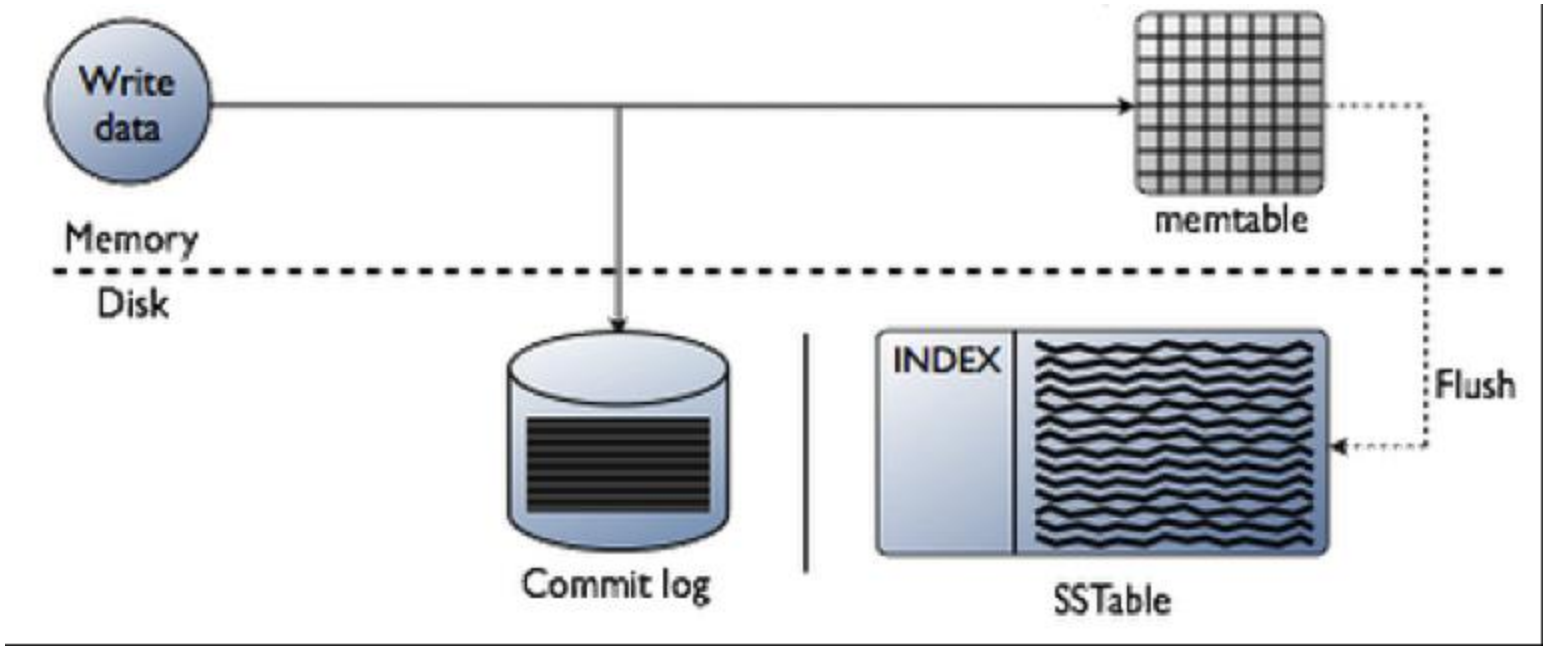
Cassandra key features

- Masterless design
- Active everywhere design
- Transparent fault detection and recovery
- Strong data protection

Cassandra architecture with no master node

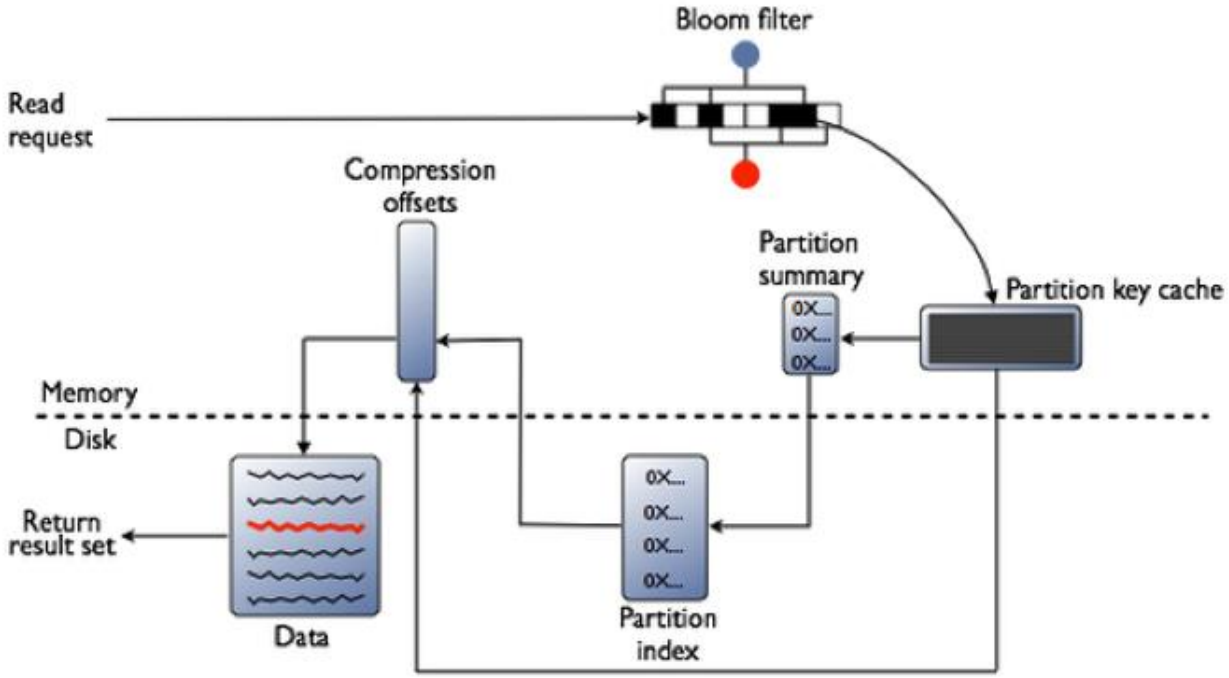
How Cassandra writes data

1. Data came to node. The data first is written to log file and then written to memory-based structure ‘memTable’.
2. Memtable’s size is exceeded. Data is written to an immutable file on disk (SSTable)



How Cassandra reads data

1. Read request goes to Bloom filter. Bloom filter checks whether data exists on SSTable very quickly.
2. If no, the request goes back. Otherwise Bloom filter asks other caches and fetches the data on disk.



Cassandra Data Objects

Object	Description
Keyspace	Like database in RDBMS. Container of all other objects.
Table	Works the same as table in RDBMS. Can handle **large volumes of data**. Fast row inserts and column level reads.
Primary Key	used to identity a row uniquely in a table and also distribute a table's rows across multiple nodes in a cluster.
Index	similar to a relational index in that it speeds some read operations

Transactions

- AID transactions
- No ‘C’-consistency (since there is no foreign keys)
- Tunable consistency for single transaction

Use Cases

- Internet of things
- Product catalogs and retail apps
- User activity tracking and monitoring
- Messaging
- Social media analytics and recommendation engines

Create key space & table

- CREATE KEYSPACE Excelsior WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor' : 3 };

```
CREATE TABLE excelsior.users (
  userid text PRIMARY KEY,
  first_name text,
  last_name text,
  emails set<text>,
  top_scores list<int>,
  todo map<timestamp, text>
);
```

INSERT & SELECT

- INSERT INTO excelsior.users (userid, first_name, last_name) values (‘1’, ‘Vlad’, ‘Matyunin’);

SELECT * FROM excelsior.users [WHERE …];

UPDATE & DELETE

- DELETE FROM excelsior.users WHERE userid='1';

UPDATE excelsior.users SET first_name='Vladislav' WHERE userid='1';

Questions

1. How to configure master node on Cassandra?
2. How does column-based database store data (compared with row based)?

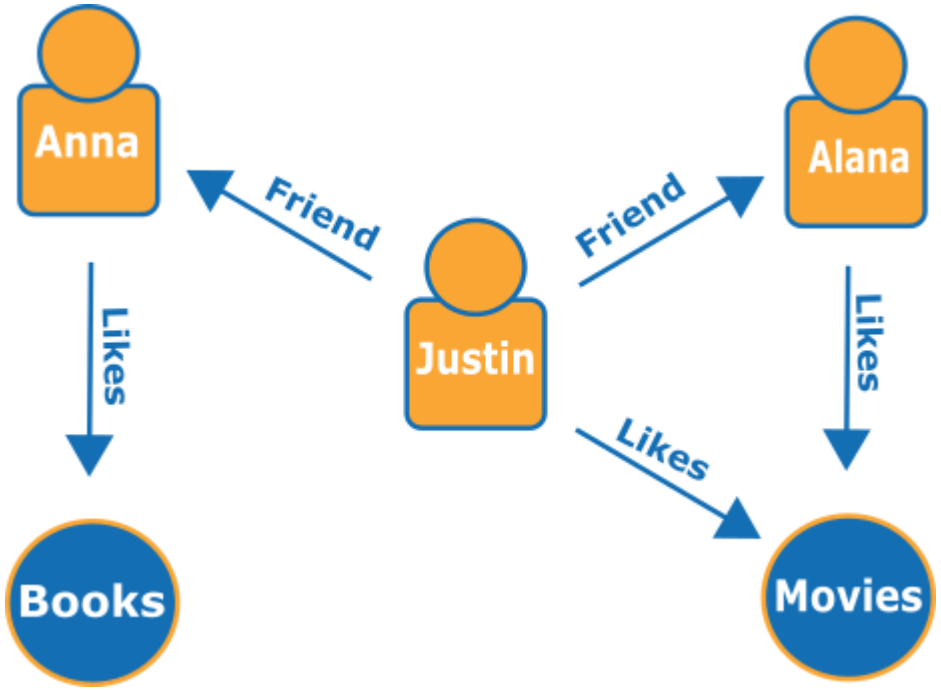
Part 5: Graph Databases

Overview

1. Structure:
- Nodes – entities storing any kind of data (the same as row in RDBMS and document in document databases)
- Edges – relationship between nodes
- Properies

2. No fixed relationships

3. Works good with multilevel data (entity has a reference to the same type)



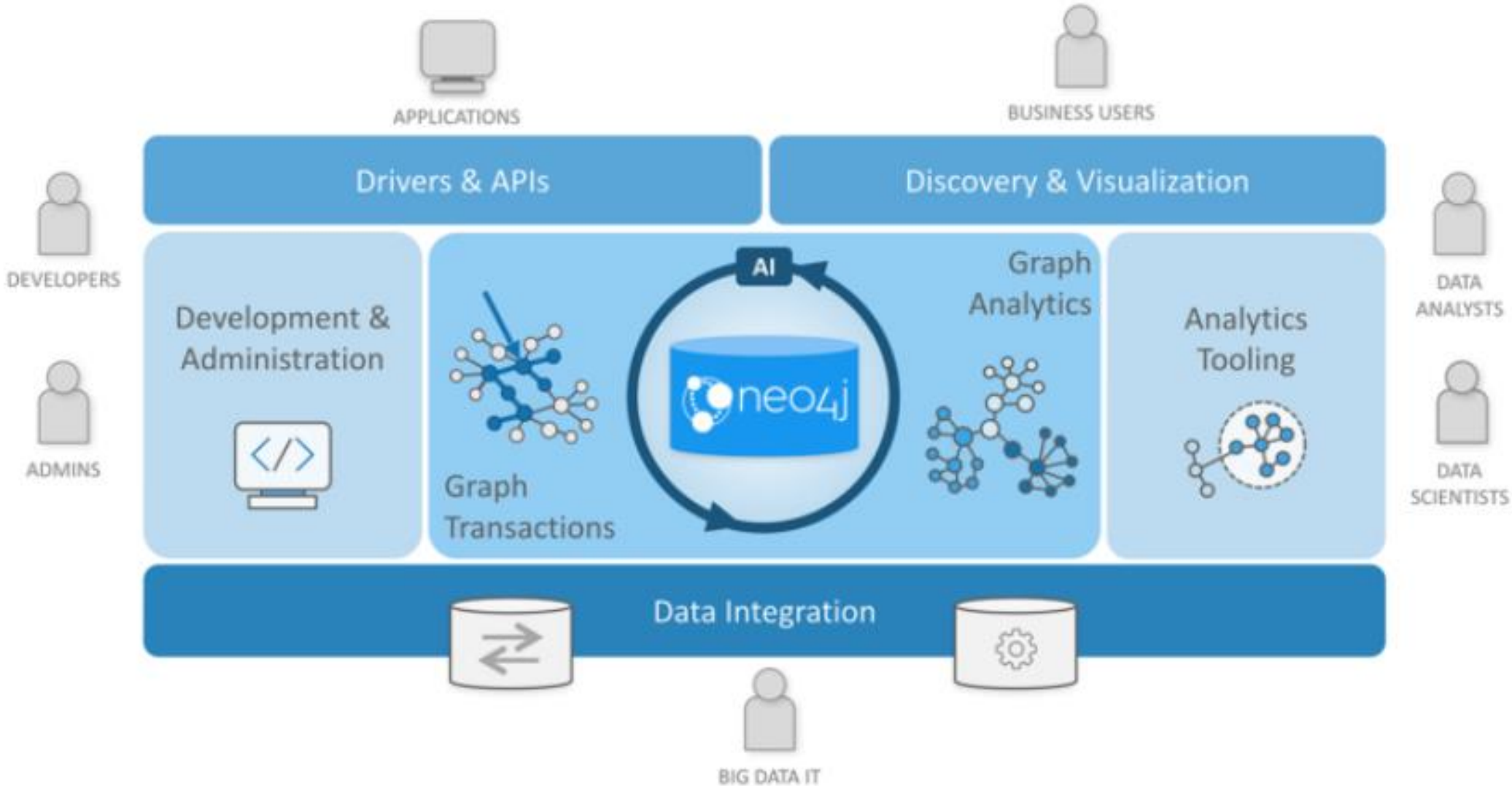
Advantages

- Object-Oriented Thinking
- Performance
- Update Data in Real-Time and Support Queries Simultaneously
- Group by Aggregate Queries
- Flexible Online Schema Environment

Disadvantages

- Bad for queries through entire database
- Bad for storing full data of entities => have to combine with another nosql or sql databases
- Almost not scalable
- Not optimized for large-volume analytics queries ("Who were all the customers with income over $100K between the ages of 35 and 50?“)


neo4j


APPLICATIONS
BUSINESS USERS
Drivers & APIs
Discovery & Visualization
DEVELOPERS
DATA ANALYSTS
Development & Administration
AI
Graph Analytics
Analytics Tooling
ADMINS
DATA SCIENTISTS
neo4j
Graph Transactions
Data Integration
BIG DATA IT

Use cases

- Fraud detection
- Manage and monitor big IT-networks(from troubleshooting to analysis)
- Real-time recommendation engines
- Social networks
- Identity & Access Management

Create a node

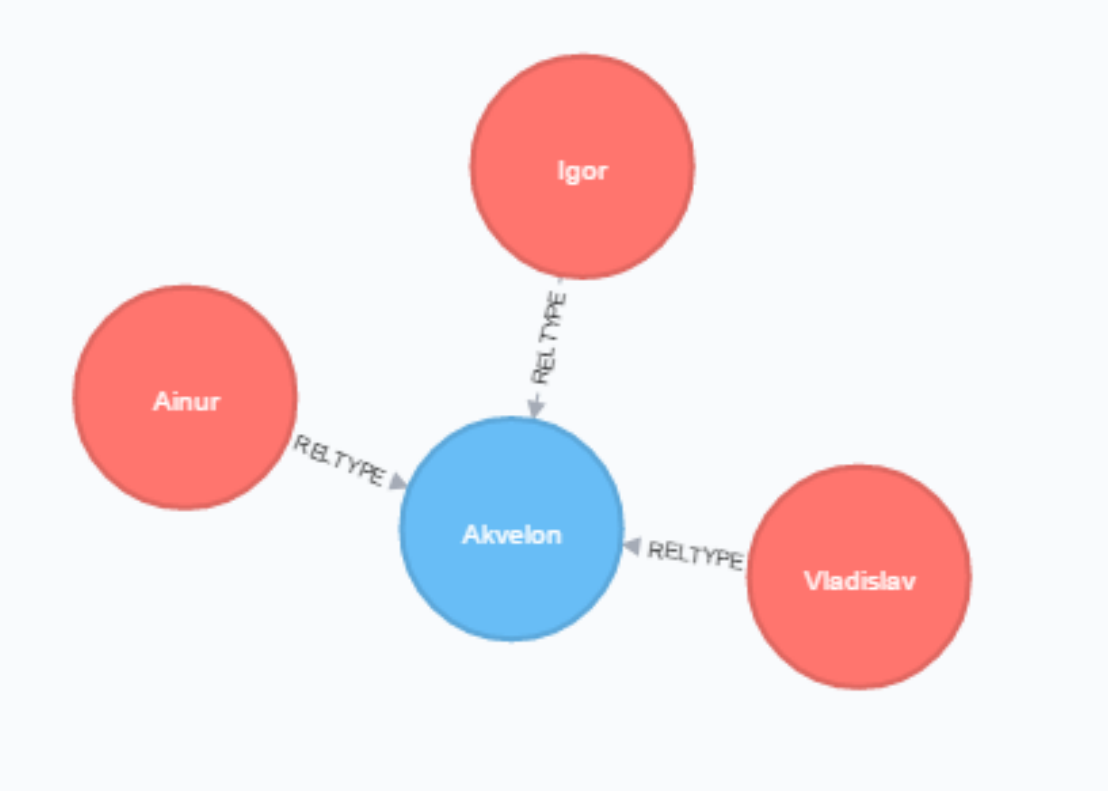
Create (n: company {name: "Akvelon"})

Create (Worker{name: "Vladislav", surname: "Matyunin"})

Relationships

MATCH (w:Worker),(c:company) WHERE c.name = 'Akvelon' CREATE (w)-[r:RELTYPE { name: w.name +' '+w.surname+ 'works in ' + c.name }]

->(c) RETURN type(r), r.name



Deleting node

```
Match(Worker {name: "Vladislav"})
DETACH DELETE Worker
```

(Detach deletes all relationships)

Select

Match (n) return n – return all nodes

START n=node(*) MATCH (n)-[r]->(m) RETURN n,r,m; - all nodes with relationships

Questions

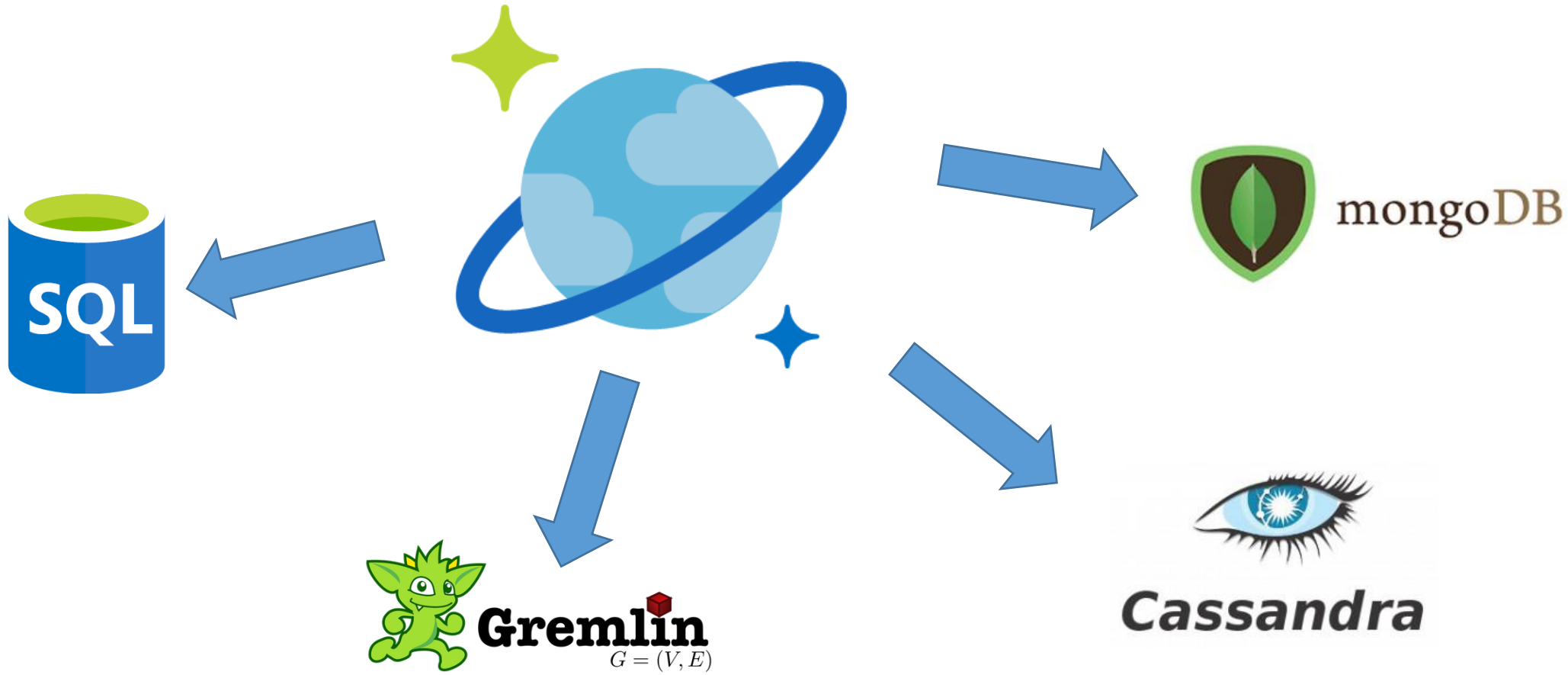
1. What is the structure of Graph databases? (Name 3 main concepts)
2. How to delete nodes with relationships in neo4j?
3. What are the disadvantages of Graph databases?

Part 6: Multi-Model databases

Global distribution Elas cy models Comprehensive SLAs

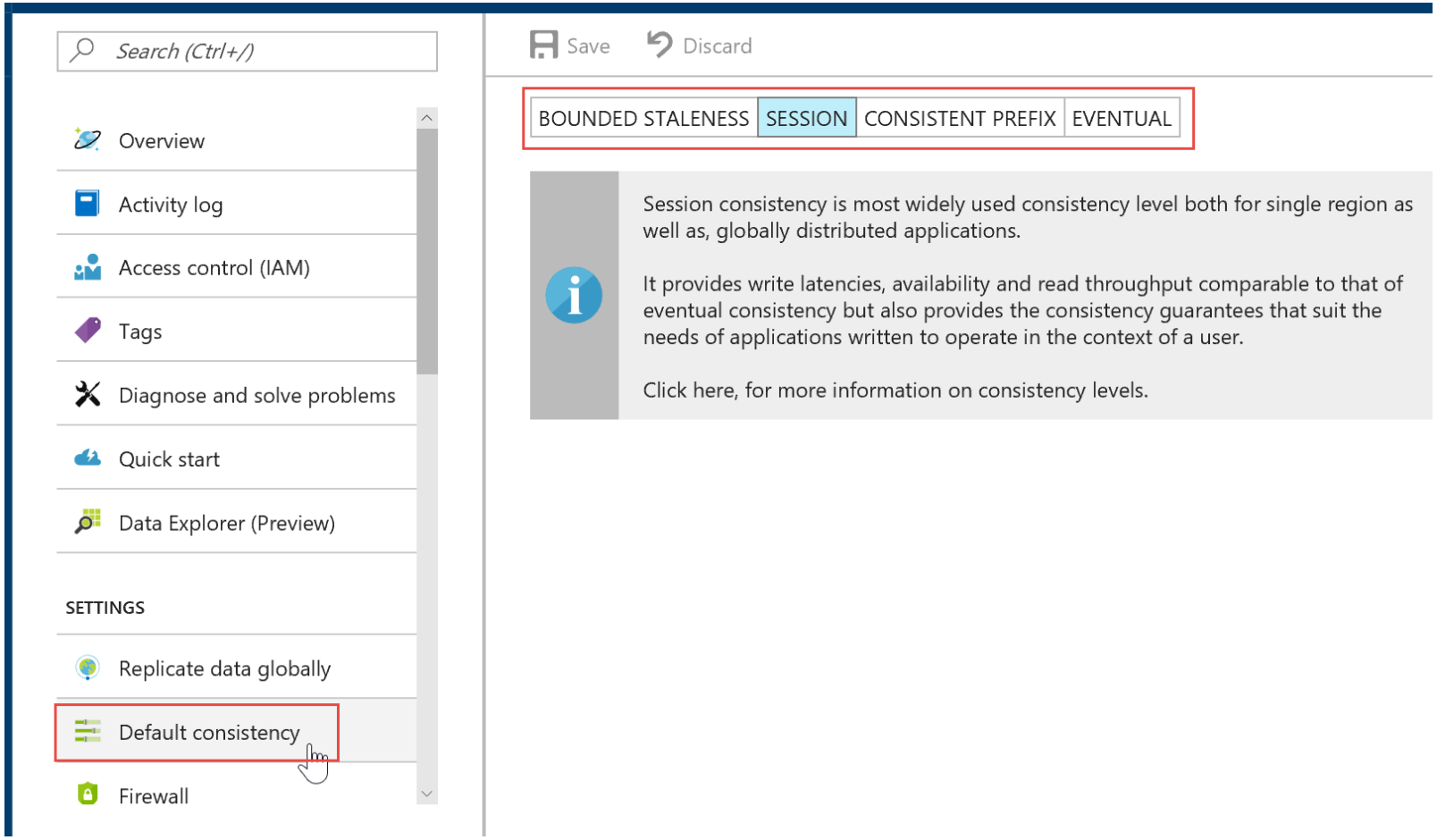
Multi-model

Different APIs



5 consistencies

- Strong
- Bounded-staleness
- Session
- Consistent Prefix
- Eventual



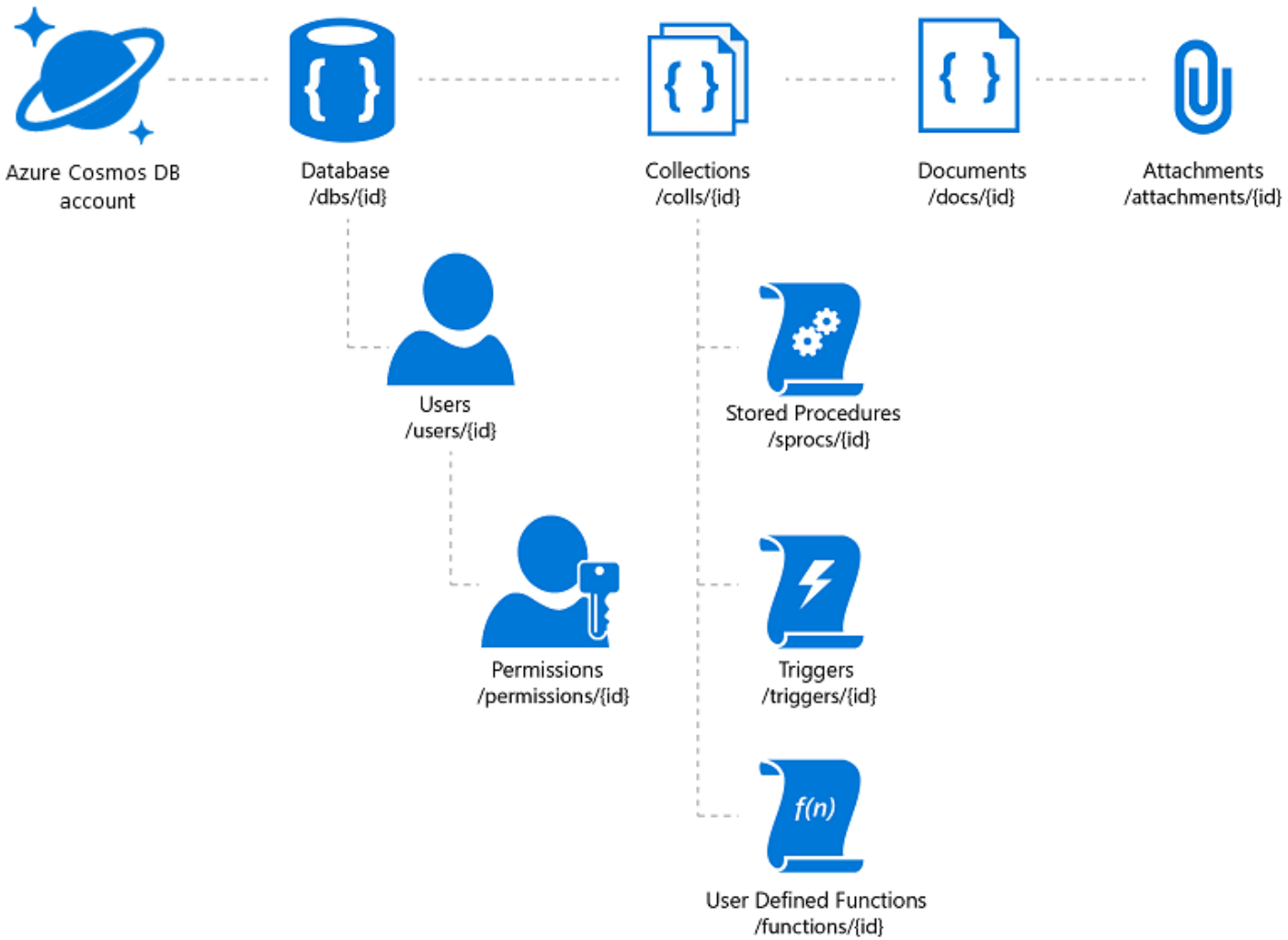
Cosmos db uses ARS [Atom-Record-Sequence]

- An atom is a primitive type.
- A record is a struct
- A sequence is an array of either atoms, records or structs.

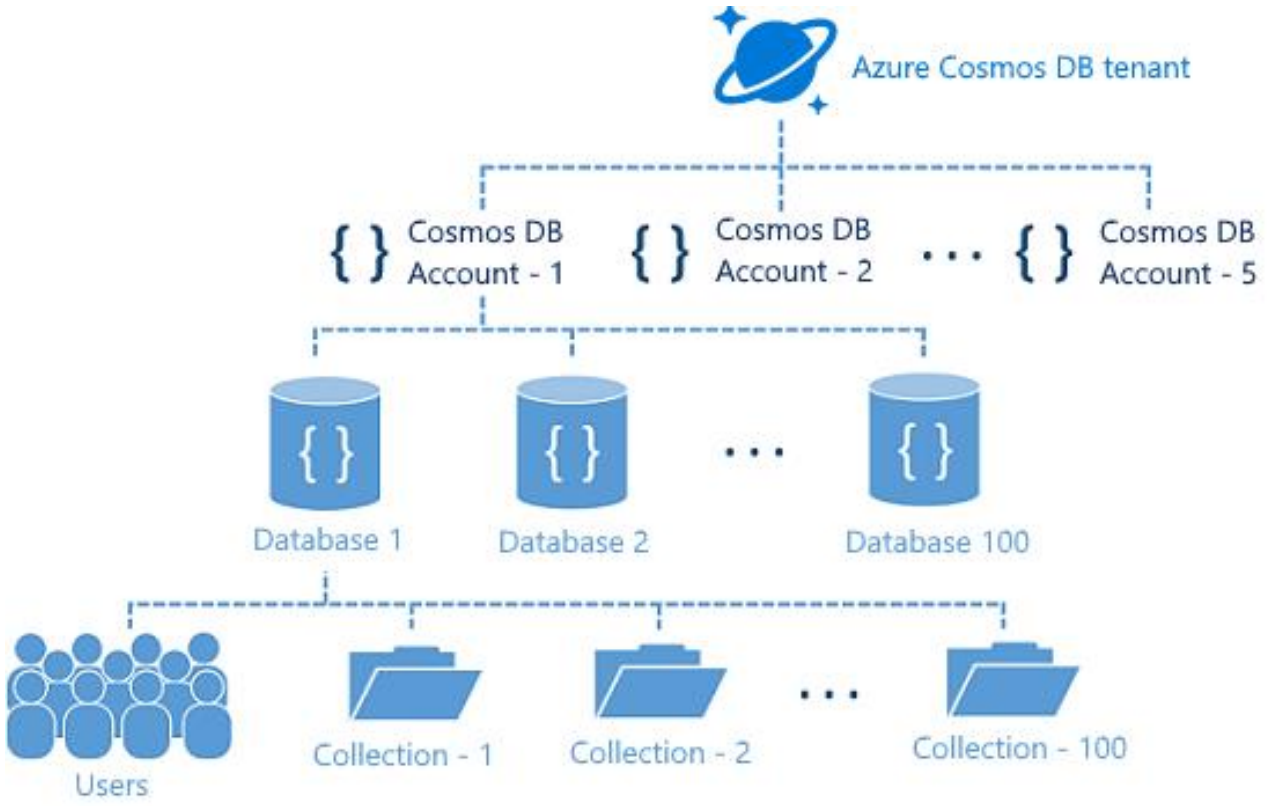
Other key features

- Powered by Azure => easily managed with a click of a button
- Easily distributive (e.g. one region for write & many regions for read out of box)
- Very high performance (typically under 15 ms indexed writes)
- Indexed automatically with Bw-trees (instead of B-tree)

Under the hood (Hierarchical resource model)



Under the hood (Database)



Use cases

- Internet of things and telematics
- Retail and marketing (storing catalog data and for event sourcing)
- Gaming (customized and personalized content like in-game stats, social media integration, and high-score leaderboards)
- Social Applications (all user generated content from posts to tags)
- And more..

Questions

- What are the key features of Cosmos DB?
- How to manage consistency in Cosmos DB?