JÖNKÖPING UNIVERSITY

School of Engineering

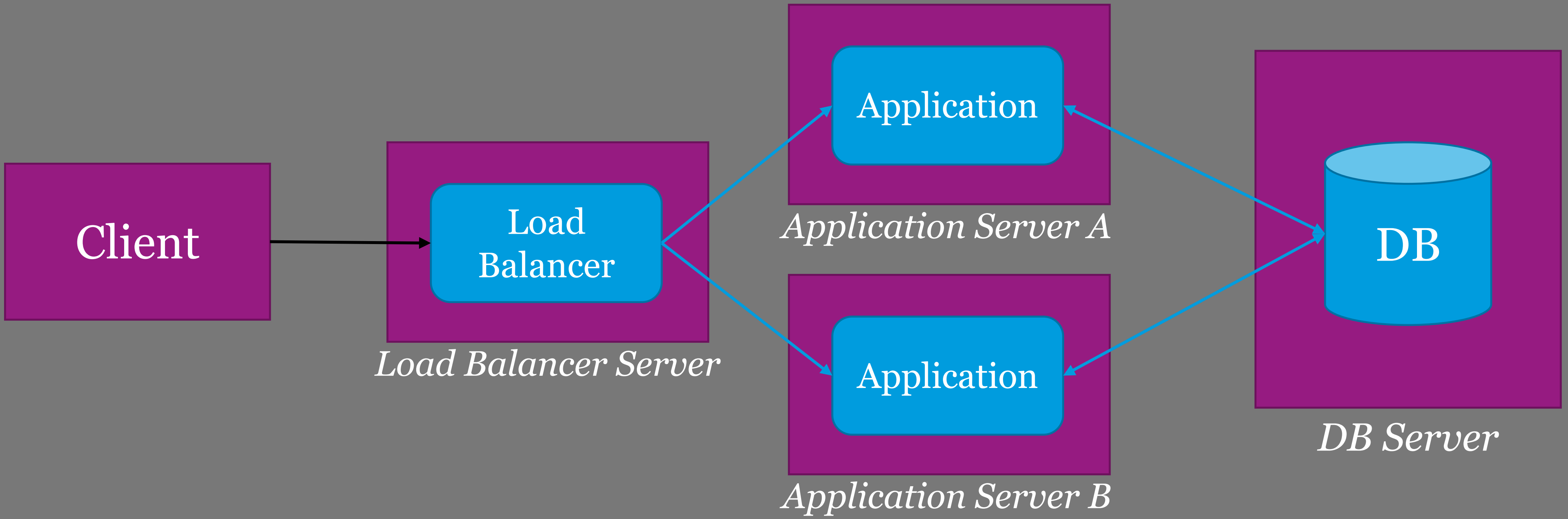
SCALING DATABASES

Peter Larsson-Green

Lecturer at Jönköping University

Spring 2019

HORIZONTAL SCALING WITH A LOAD BALANCER



Relational databases are hard to scale because they support ACID transactions.

DANGEROUS EXAMPLE

name	amount
Alice	100
Bob	100

accounts

Transfer $20 from Alice's account to Bob's account.

- First reduce Alice's amount by $20:
  - `UPDATE accounts SET amount = amount - 20 WHERE name = "Alice"`
- Then increase Bob's amount by $20:
  - `UPDATE accounts SET amount = amount + 20 WHERE name = "Bob"`

What's the problem?

- What if the second query is never executed (e.g. DB has crashed)?
  - $20 lost, leaving the database in an invalid state.

GOOD EXAMPLE

name	amount
Alice	100
Bob	100

accounts

Transfer $20 from Alice's account to Bob's account.

- Use an SQL transaction to group queries together:

```
BEGIN TRANSACTION
UPDATE accounts SET amount = amount - 20 WHERE name = "Alice"
UPDATE accounts SET amount = amount + 20 WHERE name = "Bob"
COMMIT
```

  - The DB will execute all queries, or none.

DANGEROUS EXAMPLE

name	amount
Alice	100
Bob	100

accounts

Require all names to be unique.

```
app.post("/accounts", function(req, res){
  const name = req.body.name
  const query = "SELECT name FROM accounts WHERE name = ?"
  db.get(query, [name], function(error, account){
    if(account == undefined){
      const query = "INSERT INTO accounts (name, amount) VALUES (?, 0)"
      db.run(query, [name])
    }
  })
})
```

Another client might have crated an account with the same name before this query is executed!

GOOD EXAMPLE

name	amount
Alice	100
Bob	100

accounts

Require all names to be unique.

- Use a UNIQUE constraint on the name column.

```
CREATE TABLE accounts(
  name TEXT,
  amount INTEGER,
  CONSTRAINT name_must_be_unique UNIQUE (name)
)
```

GOOD EXAMPLE

name	amount
Alice	100
Bob	100

accounts

Require all names to be unique.

- Use a UNIQUE constraint on the name column.

```
app.post("/accounts", function(req, res){
  const name = req.body.name
  const query = "INSERT INTO accounts (name, amount) VALUES (?, 0)"
  db.run(query, [name], function(error){
    if(error && error.message == "SQLITE_CONSTRAINT: UNIQUE constraint failed: accounts.name"){
      /* name already taken... */ }
    }
  })
})
```

RELATIONAL DB: ADVANTAGE

Relational databases support ACID operations:

- Atomicity:
  - Operations are fully completed, or fully aborted (a sequence of queries can be grouped into a transaction).
- Consistency:
  - All constraints, cascades (and similar) should be honored.
- Isolation:
  - If multiple transactions are executed simultaneously, the should be executed independently of each other.
- Durability:
  - Errors (including power failures) should not leave the database in a bad state.

RELATIONAL DB: DISADVANTAGE

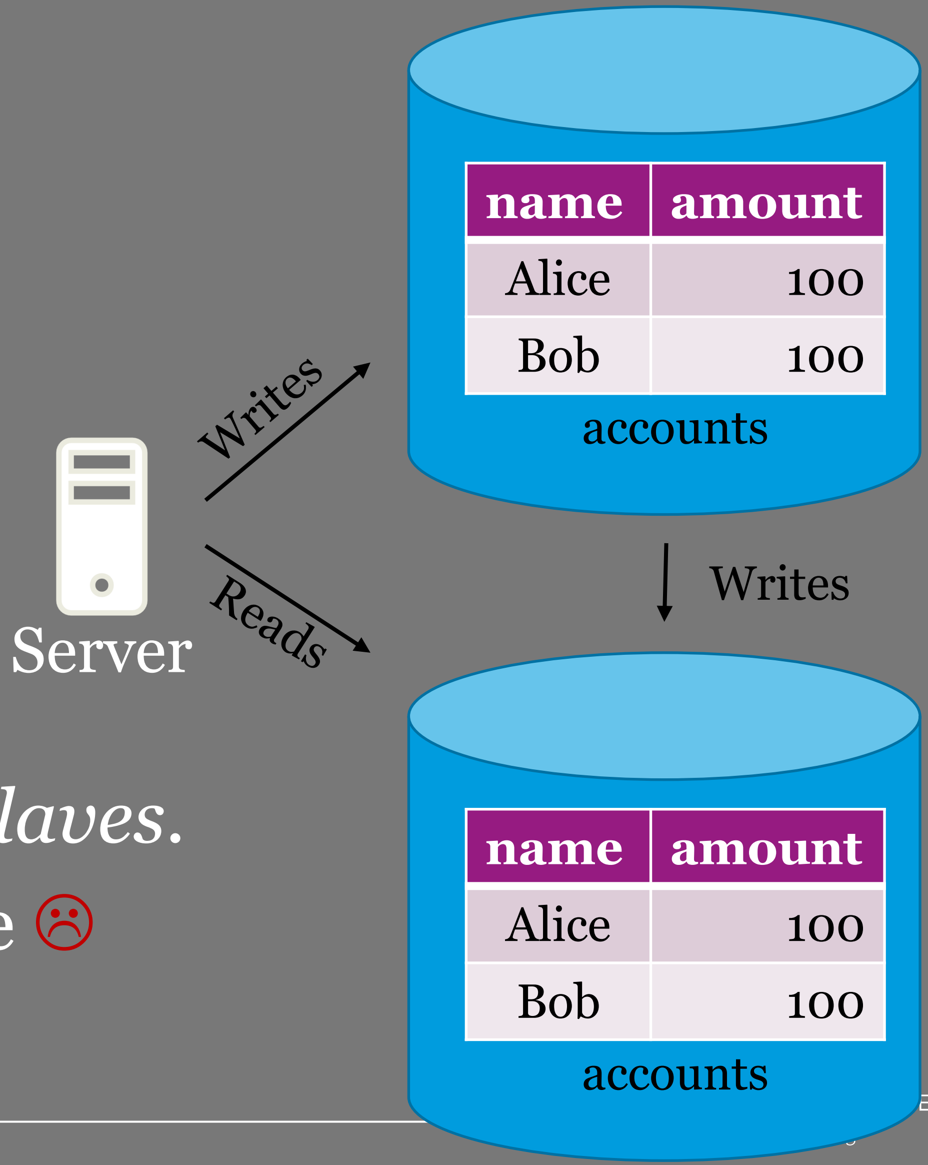
Primarily one downside with relational databases:

- Hard to scale!
  - Contains a lot of data.
  - Need to process many queries.

RELATIONAL DB: SCALING APPROACH

Example 1: Use replicas

- Can read from anyone ☺
- Need to write to all ☹



name	amount
Alice	100
Bob	100

name	amount
Alice	100
Bob	100

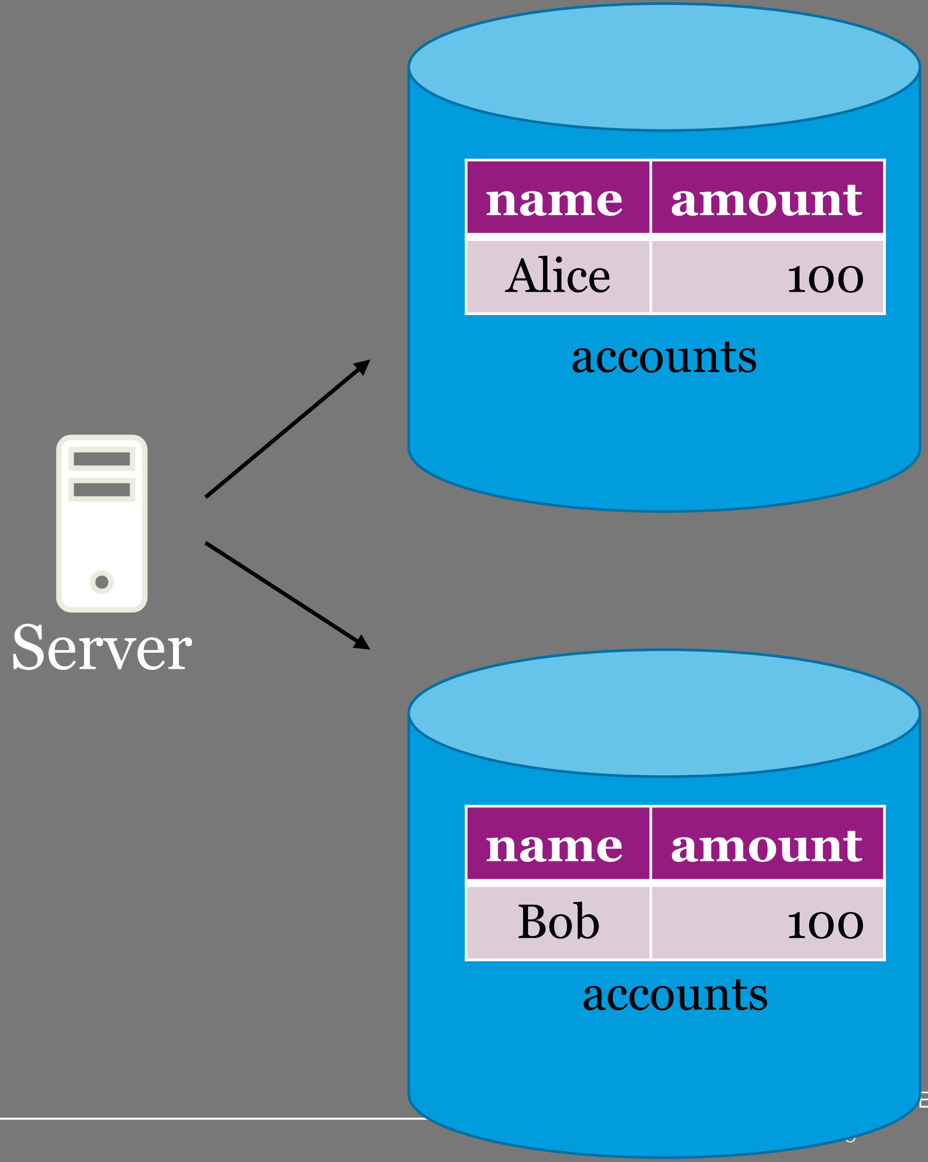
Special case: *write master with read slaves*.

- Data we read might not be up-to-date ☹

RELATIONAL DB: SCALING APPROACH

Example 2: Distribute the data

- Hard to scale when you need to use multiple DB at the same time ☹



name	amount
Alice	100

accounts

name	amount
Bob	100

accounts

RELATIONAL DB: SCALING APPROACH

No matter how you do it, it is hard to support ACID operations in a decentralized database.

- The CAP-theorem…

THE NOSQL APPROACH

- Support scaling ☺
- Drop ACID operations ☹

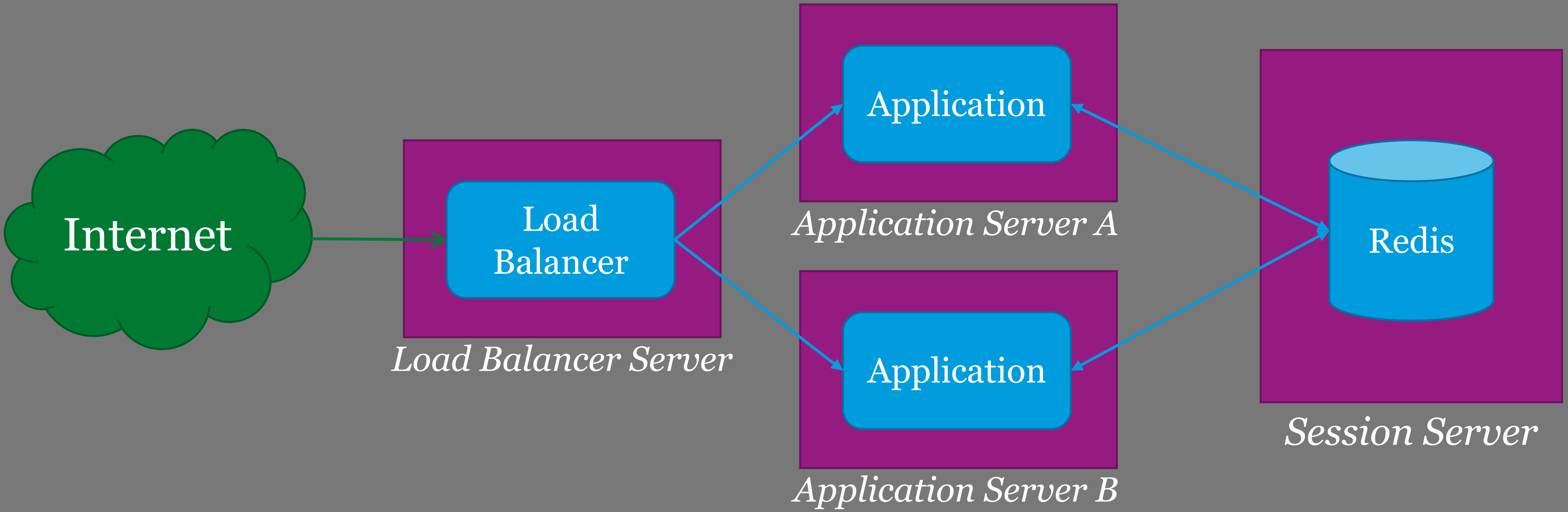
NOSQL: KEY-VALUE DATABASES

For example Redis.

- Supported operations:
    - Create: `SET("The key", "The value")`
    - Retrieve: `GET("The key")` → `"The value"`
    - Update: `SET("The key", "The value")`
    - Delete: `DEL("The key")`

NOSQL: KEY-VALUE DATABASES

Good use-case: sharing sessions across multiple servers.

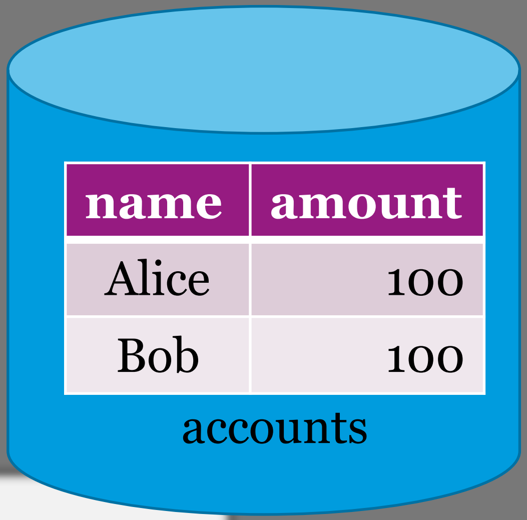


NOSQL: DOCUMENT DATABASE

For example MongoDB.

- A unit of data is called a *document*.
  - Kind of like a row in a table in a relational database.
- A collection of documents is called a *collection*.
  - Kind of like a table in a relational database.
- Documents can be nested.

NOSQL: DOCUMENT DATABASE



name	amount
Alice	100
Bob	100

accounts

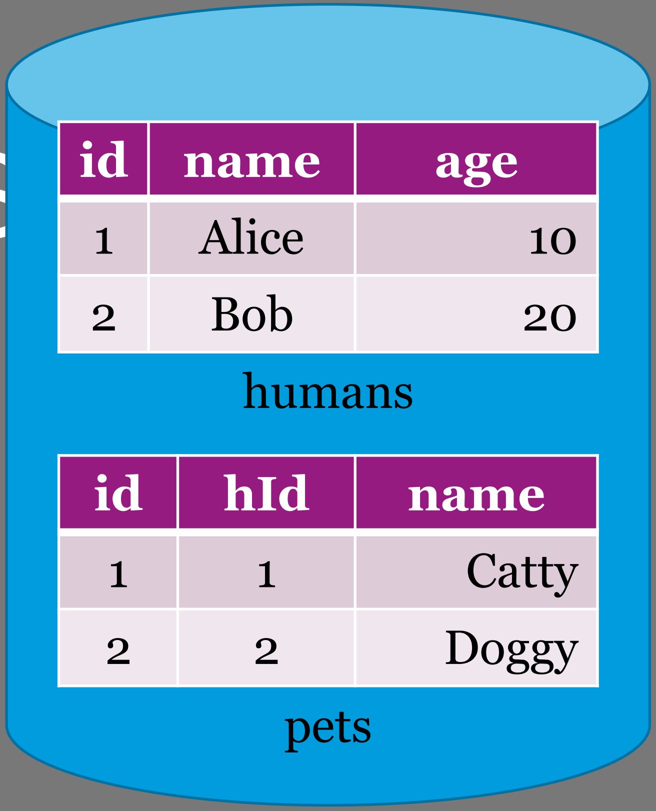
Example: Storing accounts.

```
const db = connectToDatabase()
const accounts = db.collection("accounts")
accounts.insert({name: "Alice", amount: 100})
accounts.insert({name: "Bob", amount: 100})
```

NOSQL: DOCUMENT DATABAS

Example: Storing humans and pets.

```
const humans = db.collection("humans")
humans.insert({
  name: "Alice",
  age: 10,
  pets: [{name: "Catty"}]
})
humans.insert({
  name: "Bob",
  age: 20,
  pets: [{name: "Doggy"}]
})
```



id	name	age
1	Alice	10
2	Bob	20

humans

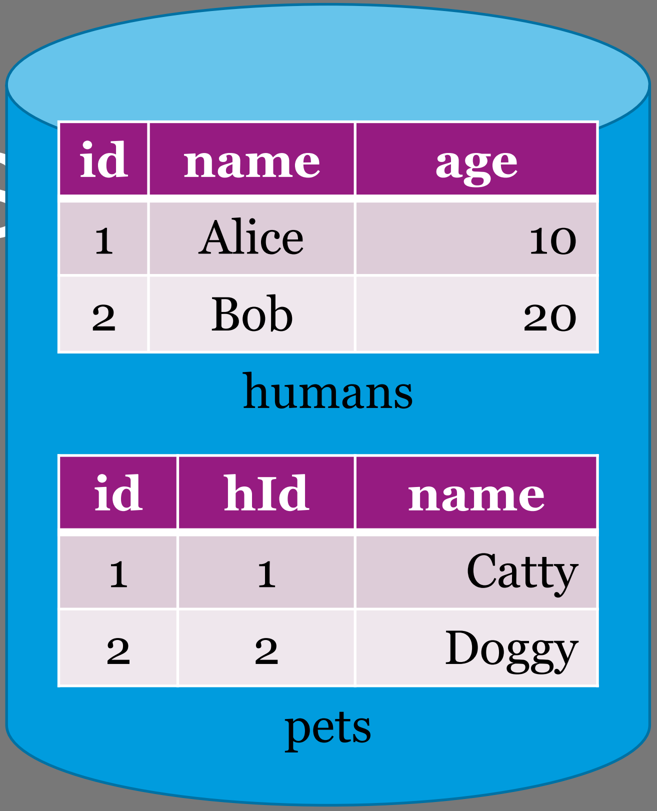
id	hId	name
1	1	Catty
2	2	Doggy

pets

Fast to fetch a human with its pets ☺

No easy way to fetch a specific pet ☹

NOSQL: DOCUMENT DATABAS



id	name	age
1	Alice	10
2	Bob	20

humans

id	hId	name
1	1	Catty
2	2	Doggy

pets

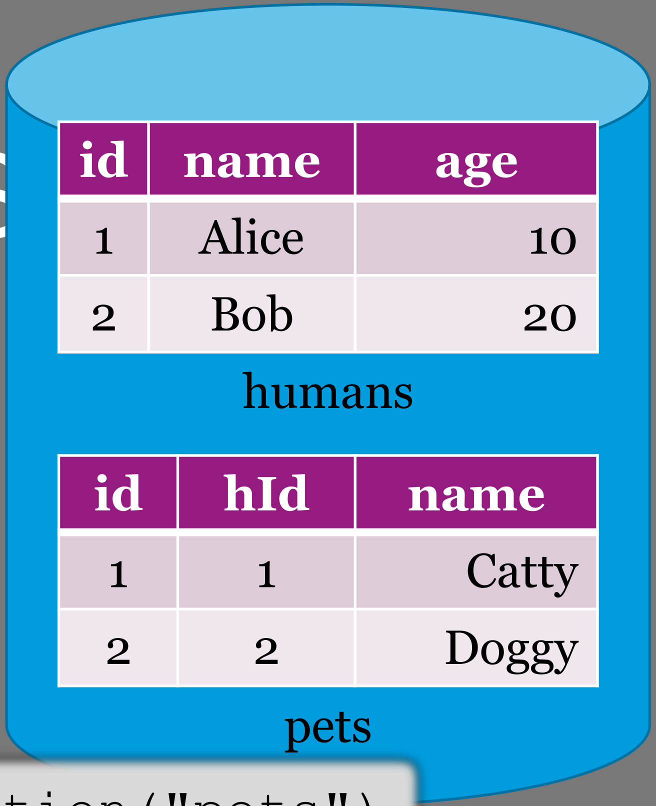
Example: Storing humans and pets.

```
const humans = db.collection("humans")
humans.insert({id: 1, name: "Alice", age: 10})
humans.insert({id: 2, name: "Bob", age: 20})
const pets = db.collection("pets")
pets.insert({id: 1, hId: 1, name: "Catty"})
pets.insert({id: 2, hId: 2, name: "Doggy"})
```

Like a relational database, but without ACID operations ☹

NOSQL: DOCUMENT DATABAS

Example: Storing humans and pets.



id	name	age
1	Alice	10
2	Bob	20

humans

id	hId	name
1	1	Catty
2	2	Doggy

pets

```
const humans = db.collection("humans")
humans.insert({
  name: "Alice",
  age: 10,
  pets: [{name: "Catty"}]
})
humans.insert({
  name: "Bob",
  age: 20,
  pets: [{name: "Doggy"}]
})
```

```
const pets = db.collection("pets")
pets.insert({
  name: "Catty",
  human: {name: "Alice", age: 10}
})
pets.insert({
  name: "Doggy",
  human: {name: "Bob", age: 20}
})
```

NOSQL LIMITS

```
{
  name: "Jönköping",
  population: 860000,
  age: 350
}
```

Firestore:

- *You can only perform range comparisons (<, <=, >, >=) on a single field, and you can include at most one array_contains clause in a compound query.*

```
cities.where("population", ">=", 1000).where("age", ">", 100)
```

- *The comparison can be <, <=, ==, >, >=, or array_contains.*

```
cities.where("population", "!=", 1000)
```

RELATIONAL DB VS NOSQL

Many big websites still use relational databases.

- Stack Overflow uses Microsoft SQL Server.

Most websites will work just fine with a relational database.

- Use a NoSQL database only if you have to or if don't have relational data.

USE-CASES FOR NOSQL

Examples:

- Google indexing web pages.
- Smartphone apps collecting data.