ArangoDB

One Engine, one Query Language. Multiple Data Models.

json { } + r a p h g + key value

About me

¡Hola, me llamo Jan!

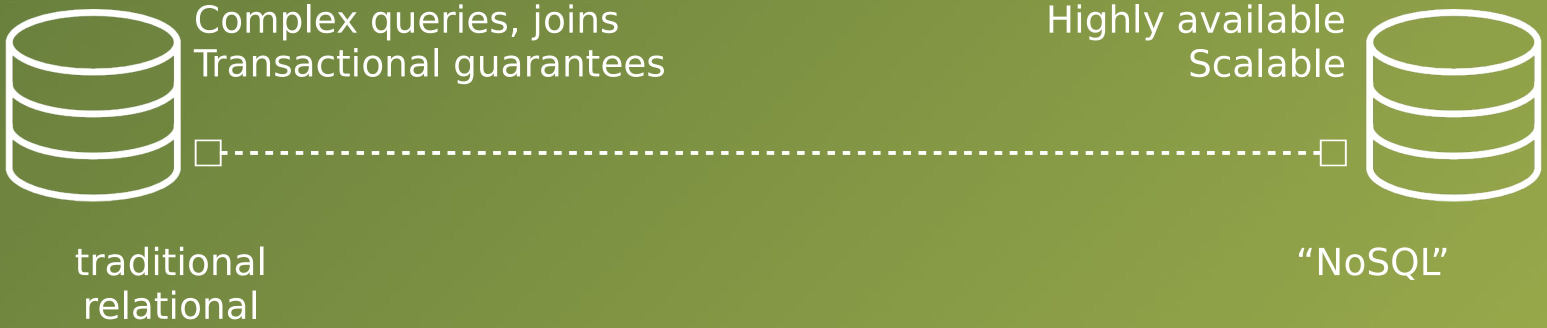
I am working for ArangoDB Inc. in Colonia, DE

I am one of the developers of ArangoDB,
the distributed, multi-model database

Running complex queries in a distributed system

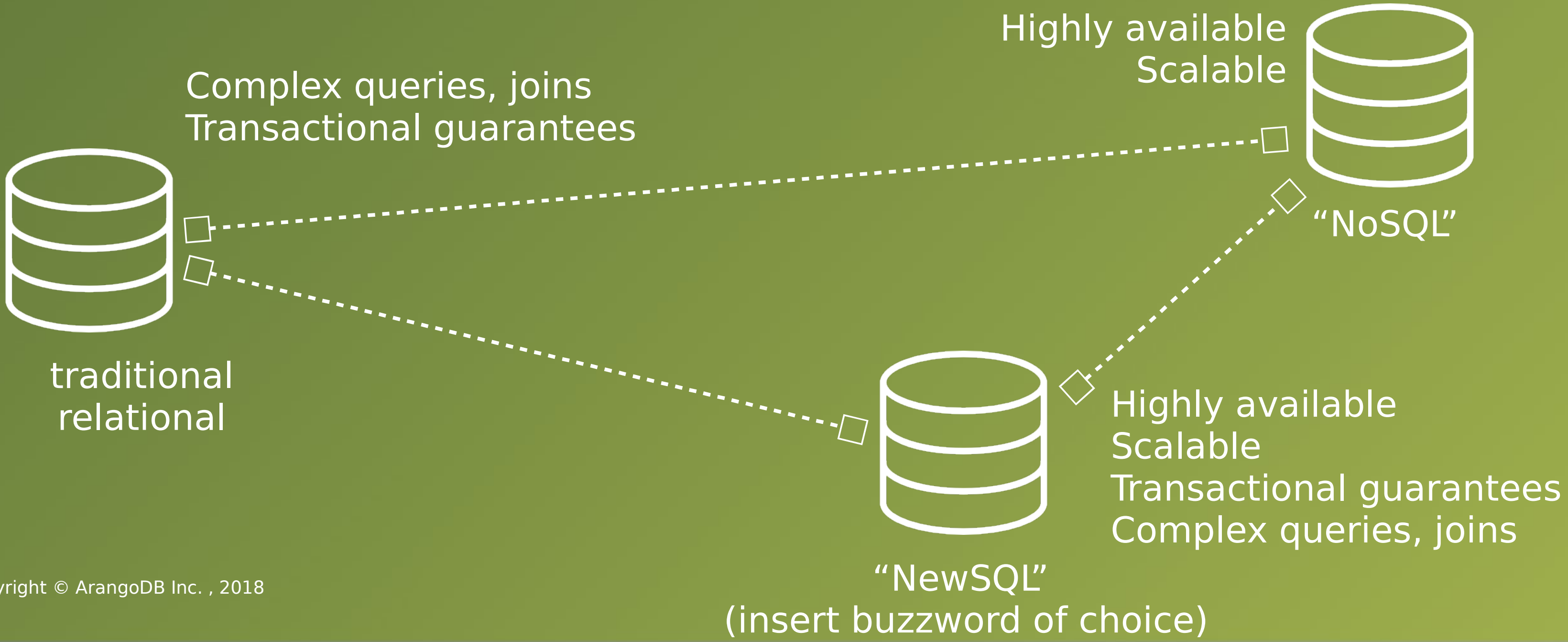
Database tradeoffs

Until recently, there was a tradeoff to consider when choosing an OLTP database



Database trends

In the last few years, there has been a trend towards distributed databases adopting complex query functionality and transactions



Agenda

- Distributed databases primer
- Organizing queries in a distributed database
- Distributed ACID transactions
- Q & A

Today I will only consider OLTP databases
Sorry, no Spark/Hadoop!

Distributed databases primer

Distributed databases

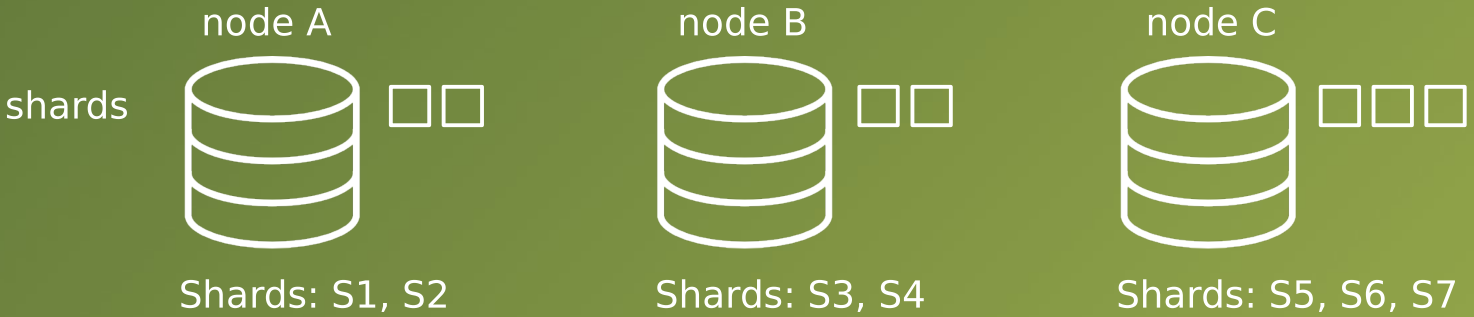
A distributed database is a cluster of database nodes

The overall dataset is partitioned into smaller chunks (“shards”)

Adding new nodes to the database increases its capacity (scale out)

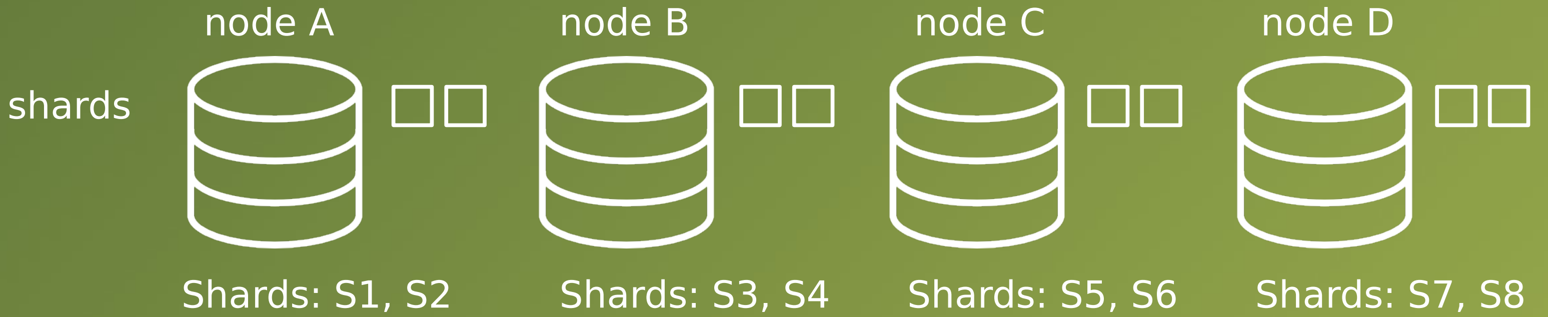
Sharding example

3 nodes (A, B, C), 7 shards (S1, S2, S3, S4, S5, S6, S7)



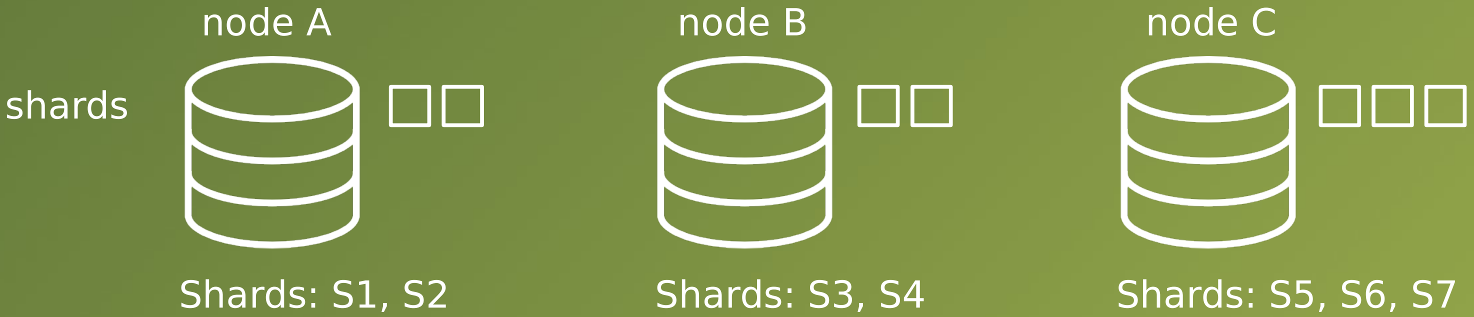
Adding a node = increased capacity

4 nodes (A, B, C, D), 8 shards (S1, S2, S3, S4, S5, S6, S7, S8)



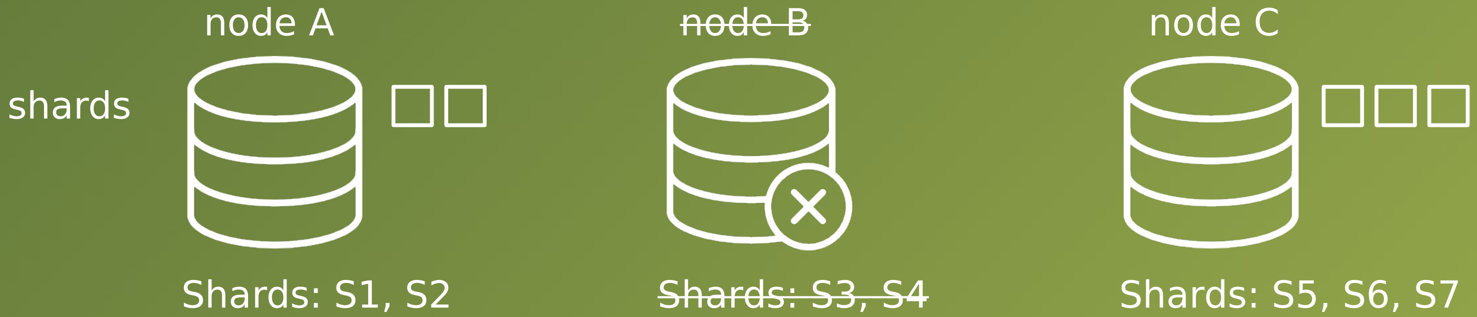
What about data loss?

3 nodes (A, B, C), 7 shards (S1, S2, S3, S4, S5, S6, S7)



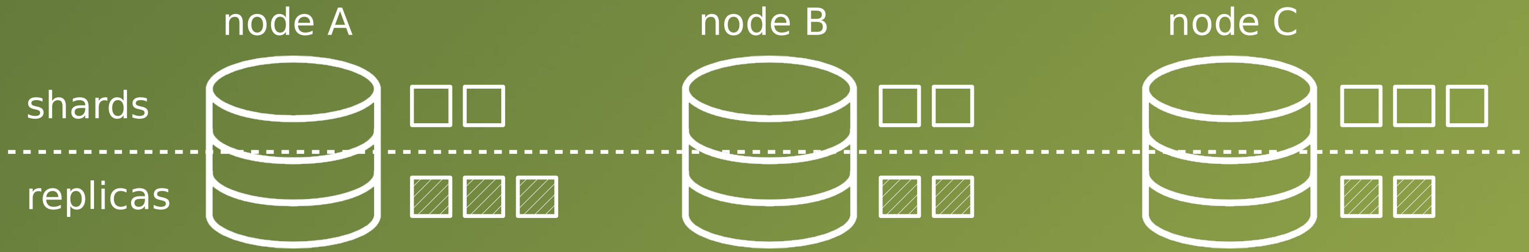
Node failure = data loss

3 nodes (A, B, C), 7 shards (S1, S2, S3, S4, S5, S6, S7)



Shards example with replicas

3 nodes (A, B, C), 7 shards (S1, S2, S3, S4, S5, S6, S7)



Shards: S1, S2

Replicas: S4, S6, S7

Shards: S3, S4

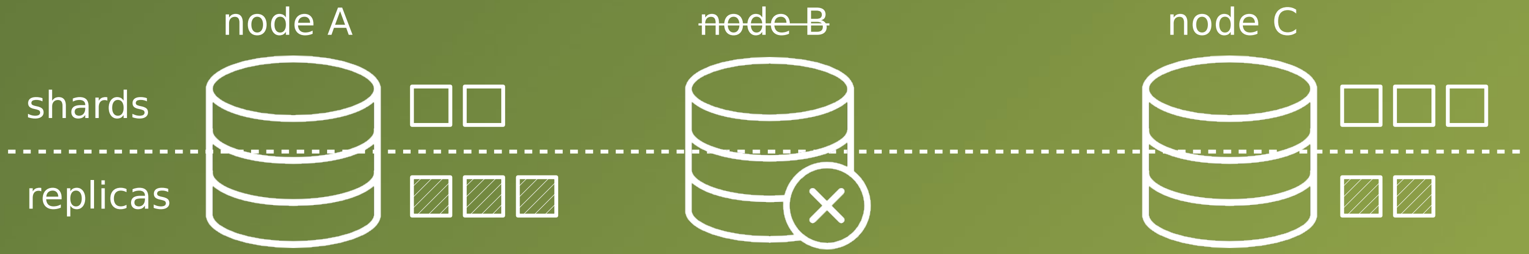
Replicas: S2, S5

Shards: S5, S6, S7

Replicas: S1, S3

Node failure with a replica setup

3 nodes (A, B, C), 7 shards (S1, S2, S3, S4, S5, S6, S7)



Shards: S1, S2

Replicas: S4, S6, S7

~~Shards: S3, S4~~

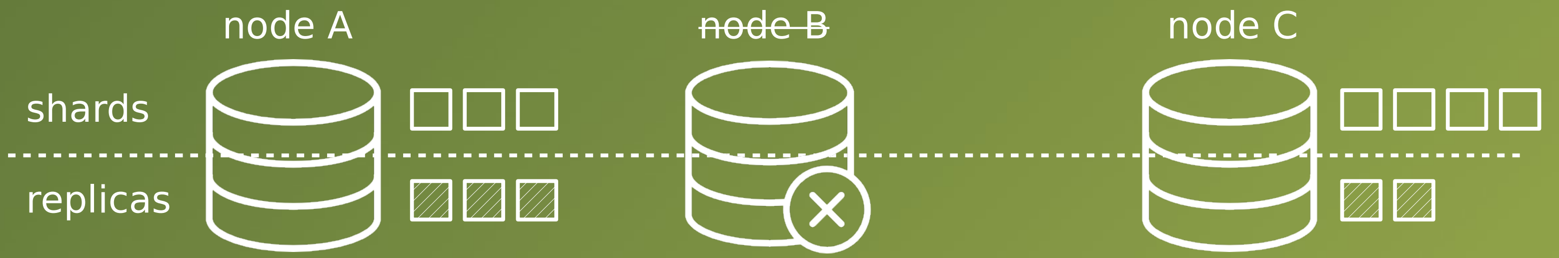
~~Replicas: S2, S5~~

Shards: S5, S6, S7

Replicas: S1, S3

Promoting replicas

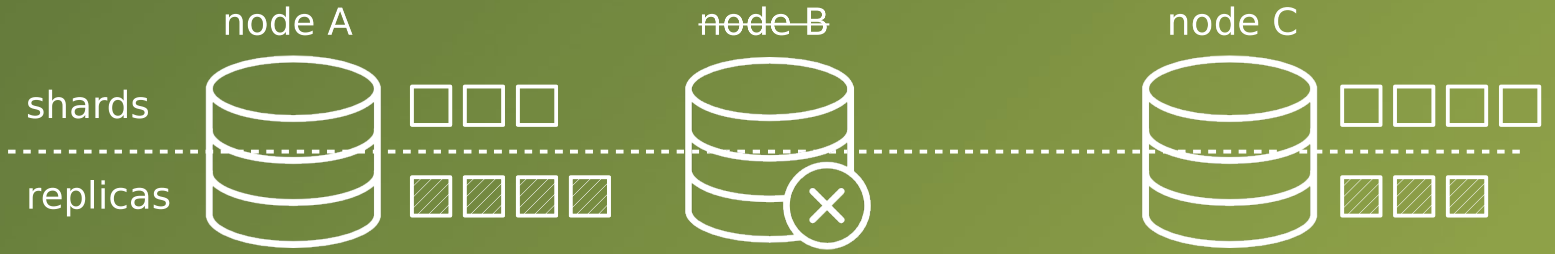
2 nodes (A, C), 7 shards (S1, S2, S3, S4, S5, S6, S7)



node A	~~node B~~	node C
Shards: S1, S2, **S4**	~~Shards: S3, S4~~	Shards: **S3**, S5, S6, S7
Replicas: ~~S4~~, S6, S7	~~Replicas: S2, S5~~	Replicas: S1, ~~S3~~

Creating new replicas

2 nodes (A, C), 7 shards (S1, S2, S3, S4, S5, S6, S7)



Shards: S1, S2, S4

Replicas: **S3**, **S5**, S6, S7

~~Shards: S3, S4~~

~~Replicas: S2, S5~~

Shards: S3, S5, S6, S7

Replicas: S1, **S2**, **S4**

Organizing queries in a distributed database

Query coordination

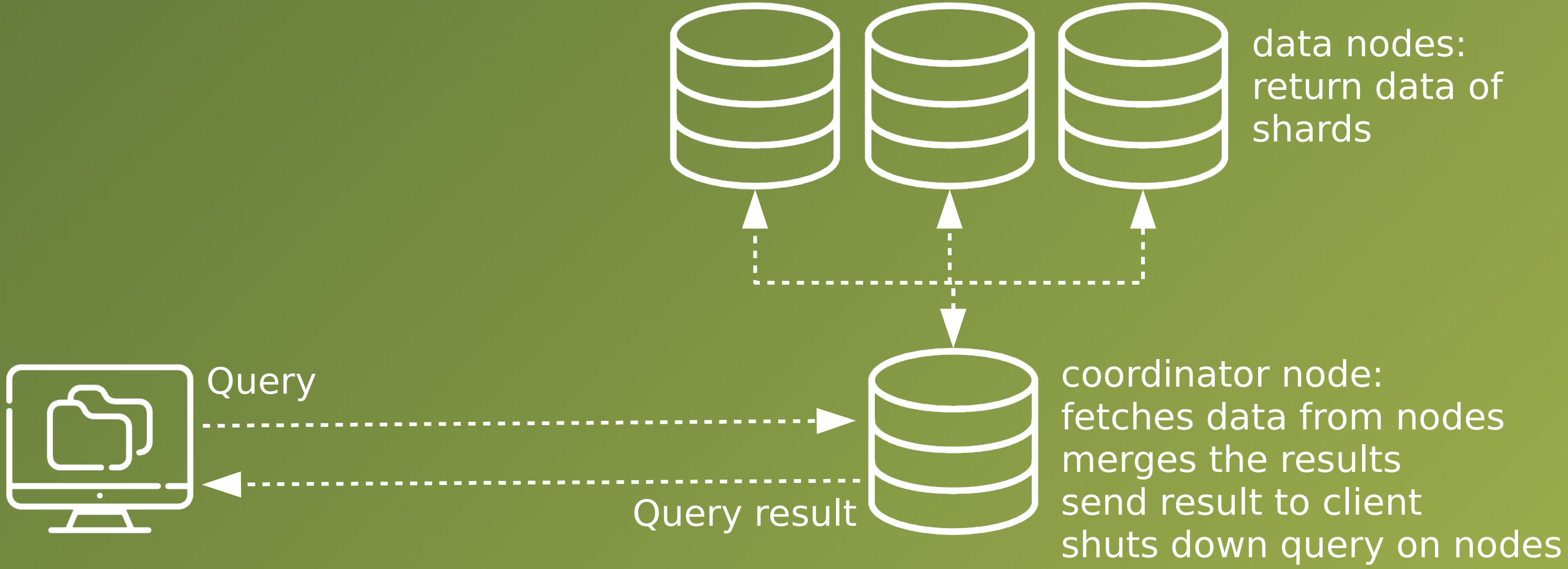
A typical distributed query will involve multiple nodes, and requires communication between them

There is normally a coordinating node for per query, which is responsible for

- triggering data processing steps on the other nodes
- putting together the partial results from the other nodes
- sending the merged result back to the client
- shutting down the query on the other nodes

Query coordination example

3 data nodes



Distributed query considerations

For each inter-node communication, there will be a network roundtrip (latency++)

One of the major goals when running distributed queries is to minimize the amount of network communication, e.g. by

- restricting the query to as few shards as possible
- pushing filter conditions to the shards
- pre-aggregating data on the shards

Operations on different shards can also be executed in parallel to reduce overall latency

ArangoDB query examples

Now following are some example queries from ArangoDB

ArangoDB is a multi-model NoSQL database, which supports documents, graphs and key-values

It can be run in single-server or distributed (cluster) mode

ArangoDB provides its own query language AQL, which is similar to SQL, but has a different syntax

Query example (filter)

A simple ArangoDB query with a filter condition:

```
FOR u IN users
  FILTER u.active == true
  RETURN u
```

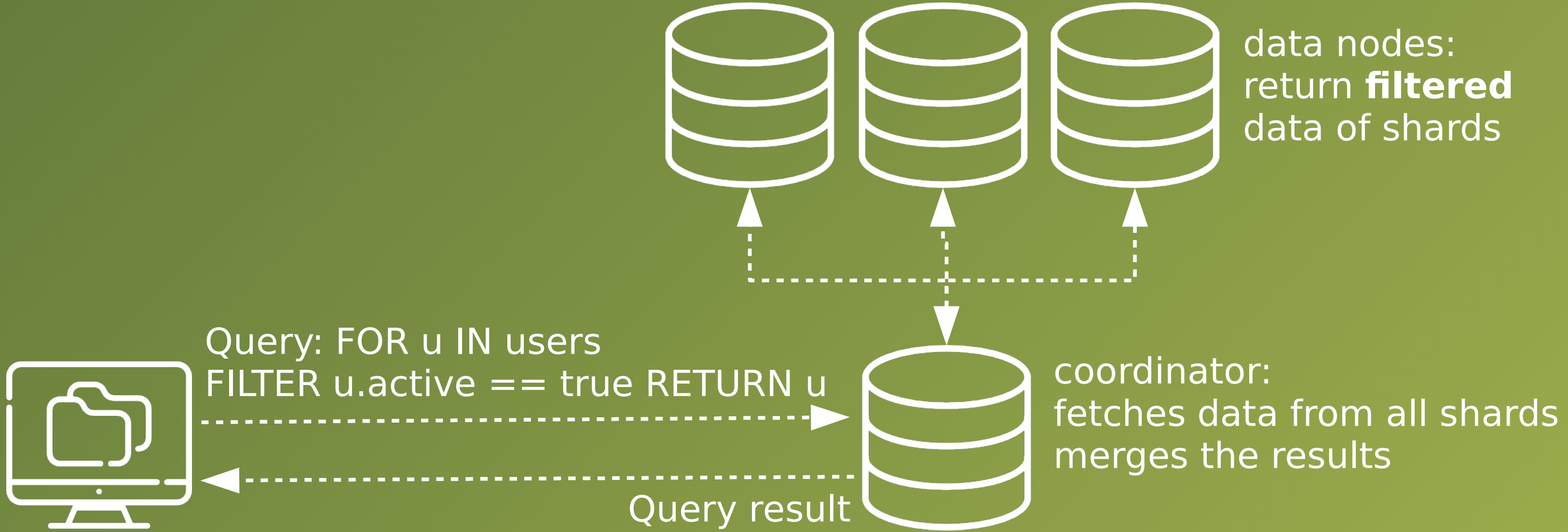
which is equivalent to SQL's

```
SELECT * FROM users u WHERE u.active = 1
```

The coordinator will push the filter condition to the shards, so they will only return data that satisfies the filter condition

Query example (filter)

3 data nodes



Query example (filter on shard key)

Now a query using a filter on a shard key attribute:

```
FOR u IN users
  FILTER u._key == “jsteemann”
  RETURN u
```

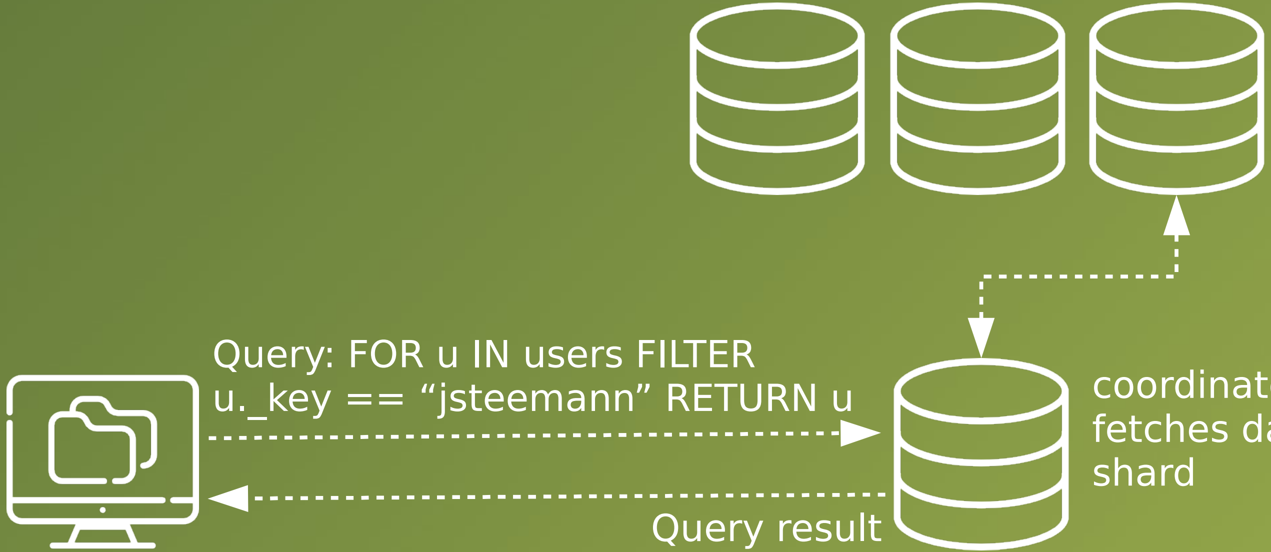
which is equivalent to SQL’s

```
SELECT * FROM users u WHERE u._key = “jsteemann”
```

The coordinator will restrict to query to the one shard the data is located on, push the filter condition to the shard and fetch the results from there

Query example (filter on shard key)

3 data nodes


single data node:
returns **filtered**
data of shard
Query: FOR u IN users FILTER
u._key == “jsteemann” RETURN u
coordinator:
fetches data from **single**
shard
Query result

Query example (sorting)

Another ArangoDB query, now with a sort condition and a projection:

```
FOR u IN users
  SORT u.name
  RETURN u.name
```

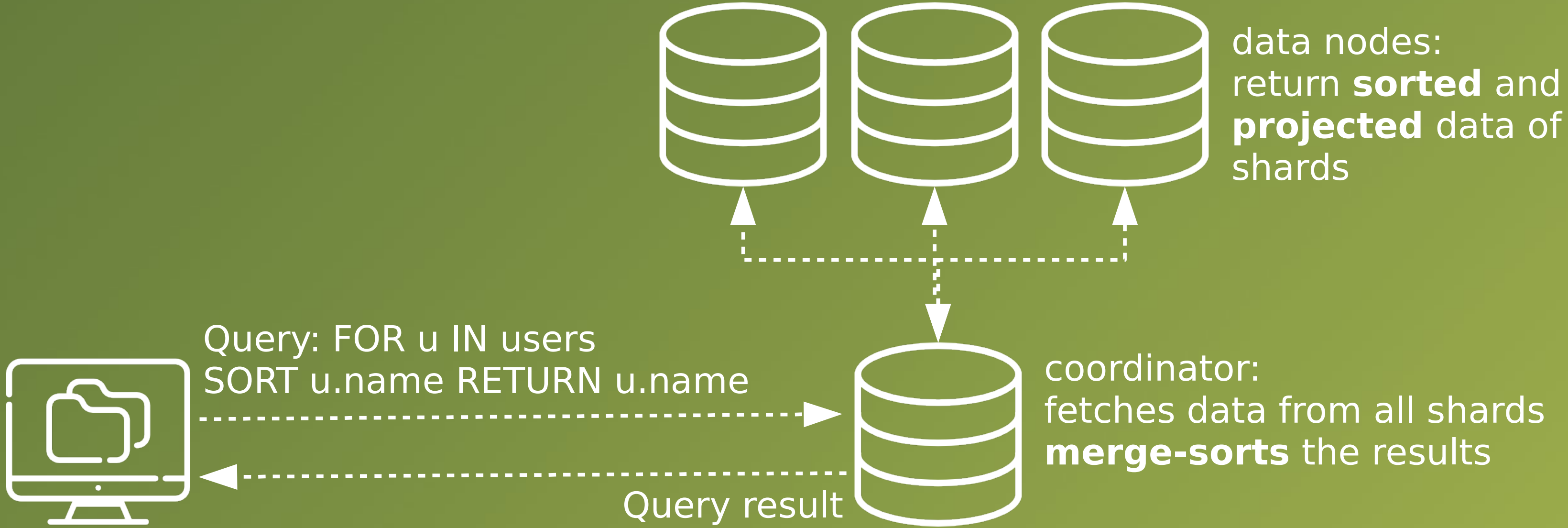
which is equivalent to SQL’s

```
SELECT u.name FROM users u ORDER BY u.name
```

The coordinator will push the sort condition and the projection to all shards, and combines the locally sorted results from the shards into a totally ordered result (using merge-sort)

Query example (sorting)

3 data nodes


data nodes:
return **sorted** and **projected** data of shards
Query: FOR u IN users SORT u.name RETURN u.name
coordinator:
fetches data from all shards
merge-sorts the results
Query result

Query example (aggregation)

One more ArangoDB query, now using aggregation:

```
FOR u IN users
  COLLECT year = DATE_YEAR(u.dob)
  AGGREGATE count = COUNT(u.dob)
  RETURN { year, count }
```

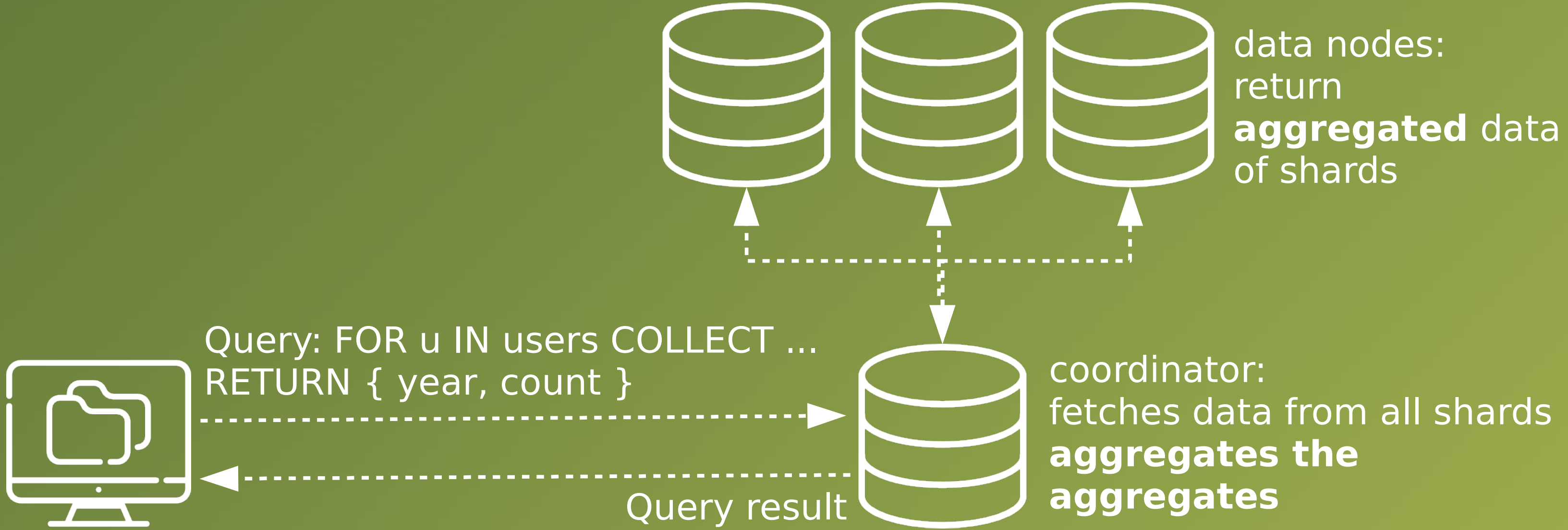
which is equivalent to SQL's

```
SELECT YEAR(u.dob) AS year, COUNT(u.dob) AS count
FROM users u GROUP BY year
```

The coordinator will push the aggregation to all shards, and combines the already aggregated results from the shards into a single result

Query example (aggregation)

3 data nodes



Query example (join)

One final ArangoDB query, now with an equi-join:

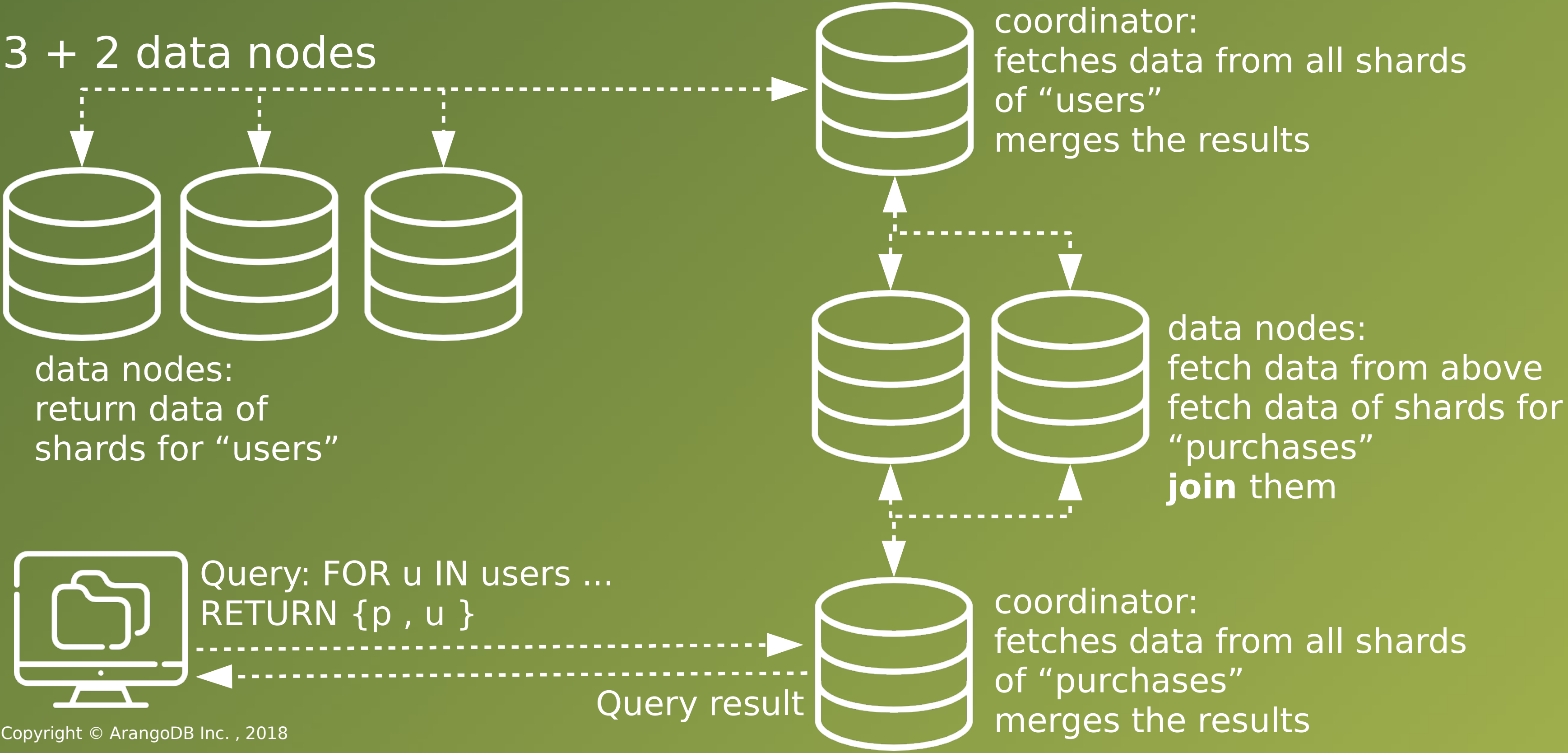
```
FOR u IN users FOR p IN purchases
  FILTER u._key == p.user
  RETURN { user: u, purchase: p }
```

which is equivalent to SQL's

```
SELECT u.* AS user, p.* AS purchase
FROM users u, purchases p WHERE u._key = p.user
```

The coordinator will query all shards of the "purchases" collection, and these will reach out to the coordinator again to get data from all shards of the "users" collection

Query example (join)



Distributed ACID transactions

Benefits of transactions

With transactions, complex operations on multiple data items can be executed in an all-or-nothing fashion

If something goes wrong, the database will do an automatic cleanup of partially executed operations

With transactions, the database will ensure consistency of data and protect us from anomalies, no matter if there are other concurrent operations on the same data

Key take-away: transactions make application developers’ lifes easier

Distributed databases with transactions

Some distributed databases also support ACID transactions
or have plans to add them:

- Google Cloud Spanner (Database as a service)
- CockroachDB
- FoundationDB
- FaunaDB (closed source)
- ...
- MongoDB (announced for future releases, with limitations)

Atomicity

While a distributed transaction is ongoing, it may make modifications on different nodes

These changes need to be ineffective (hidden) until the transaction actually commits

On commit, the transaction’s changes must become instantly visible on all nodes at the same time

Atomicity

Distributed databases normally store the status of transactions (pending, committed, aborted) in a private section of the key space, e.g:

Key	Value
T0	commited
T1	aborted
T2	pending

When a transaction commits, its status key is atomically updated from “pending” to “committed”

Consistency – designated leaders

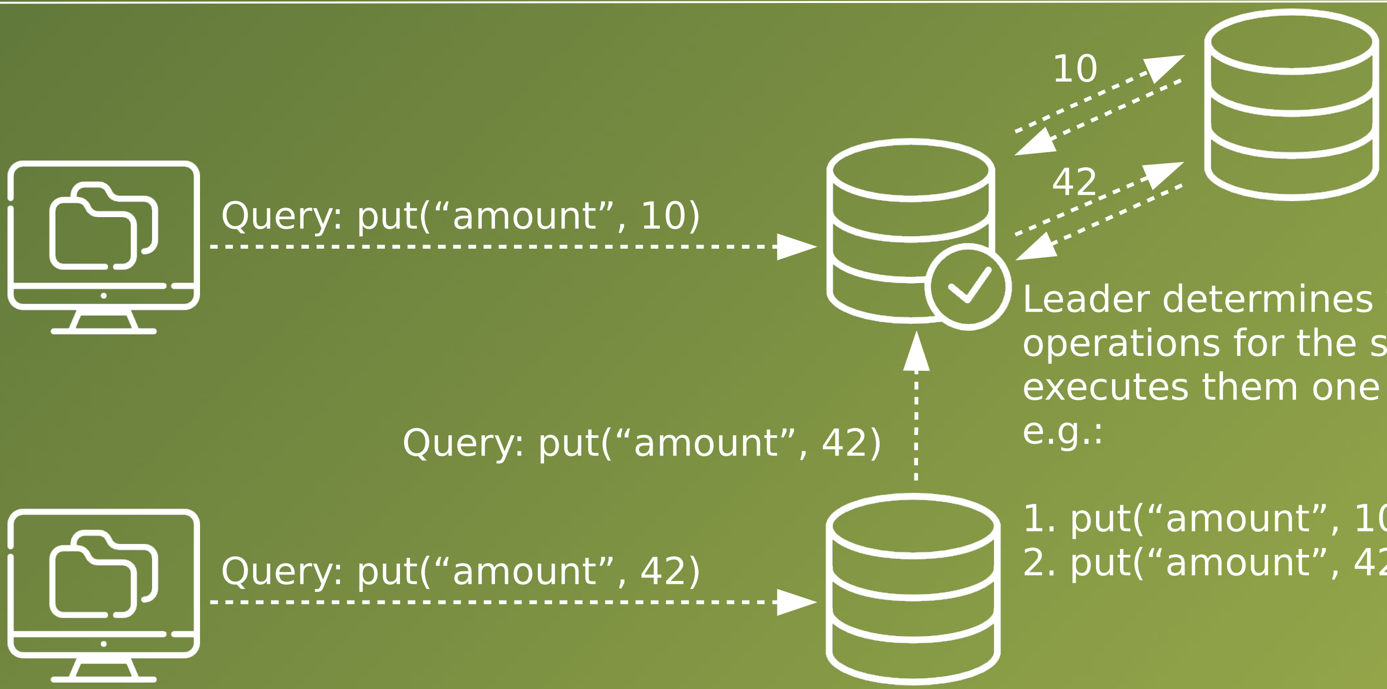
Databases that provide consistency normally serialize all write operations for a key on the designated “leader” node for its shard

The state of data on the leader shard then is a consistent ”source of truth” for that shard

Write operations are replicated from leaders to replicas in the same order as applied on the leader

Replicas are thus exact copies of the leader shards and can take over any time

Leader-only writes



Leader determines the order of the operations for the same key and executes them one after the other, e.g.:

1. put(“amount”, 10)
2. put(“amount”, 42)

Shard leadership

Shard leaders can change over time, e.g. in case of node failures, planned maintenance

It is necessary that all nodes in the cluster have the same view on who is the current leader for a specific shard, and which are the shard’s current replicas

Consensus protocols

The nodes in the cluster normally use a “consensus protocol” to exchange status messages

Paxos and RAFT are the most commonly used consensus protocols in distributed databases

These protocols are designed to handle network partitions and node failures, and will work reliably if a majority of nodes is still available and can still exchange messages with each other

Ordering transactions

To ensure consistency, transactions that modify the same data must be put into an unambiguous order

Having an unambiguous global order allows having a cross-node consistent view on the data

This is hard to achieve because the transactions can start on different nodes in parallel

Ordering transactions using timestamps

Each transaction is assigned a timestamp when it is started

This same timestamp will be used later as the transaction’s commit timestamp

The timestamps of transactions will be used for ordering them

Rule: a transaction with a lower timestamp happened before a transaction with a higher timestamp

Clock skew

Timestamps created by different nodes are not reliably comparable due to clock skew

The solution to make them comparable in most cases is to define an “uncertainty interval” (which is the maximum tolerable clock skew)

If the timestamp difference is outside of the “uncertainty interval”, two timestamps are safely comparable

Two timestamps with a difference inside the uncertainty interval are not comparable safely, and the relative order of them is unknown

Consistency using timestamps

If the transactions could have influence on each other, this is an (actual or a potential) read or write conflict, and one of the transactions must be aborted or restarted

A transaction restart also means assigning a new, higher timestamp

Isolation

To ensure isolation, a running transaction must not overwrite or remove data that another ongoing transaction may still see

Write operations are stored in a multi-version data structure, which can handle multiple values for the same key at the same time

Any transaction that reads or writes a key needs to find the “correct” version of it

Isolation – multi-versioning

Key	Transaction ID	Value
“amount”	T0	10
”amount”	T1	42
”name”	T17	”test”
”page”	T2	”index.html”
”page”	T50	<removed>

Any operation can identify whether it can “see” an operation from another transaction, simply by looking up the status and timestamp of the corresponding transaction

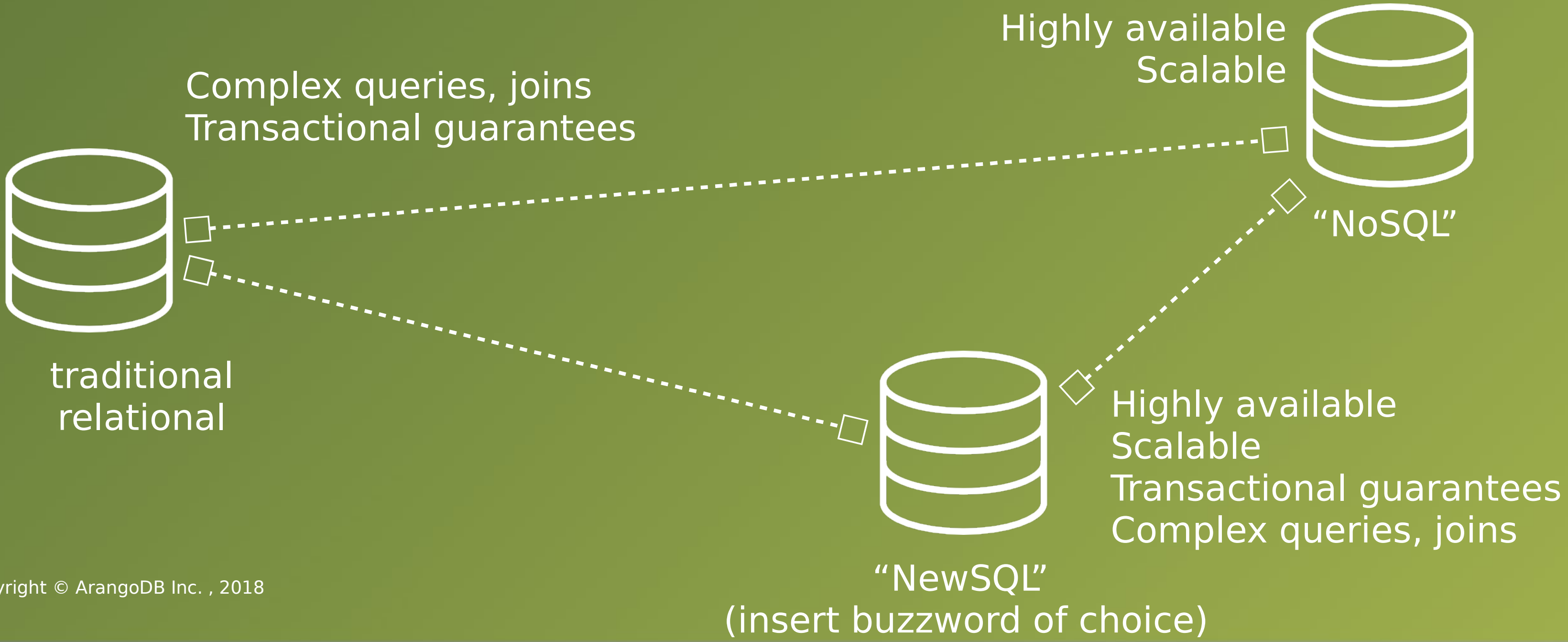
Durability

To ensure durability, every write operation (and also transaction status changes) needs to be persisted on multiple nodes (leader + replicas)

A commit is only considered successful if acknowledged by a configurable number of nodes

Database trends

In the last few years, there has been a trend towards distributed databases adopting complex query functionality and transactions



¡Muchas gracias!

¿Hay preguntas?

Links / credits

Please star ArangoDB on Github:
https://github.com/arangodb/arangodb

Participate in ArangoDB’s community survey to win a t-shirt:
https://arangodb.com/community-survey/

#arangodb | jan@arangodb.com

Icons made by Freepik (www.freepik.com) from www.flaticon.com, licensed by CC 3.0 BY