Inside ArangoDB's new Sharding

Max Neunhöffer

NoSQL matters 2014

Cologne, 30 April 2014

What is sharding?

- Use multiple servers and distribute the data.

What is sharding?

- Use multiple servers and distribute the data.
- Let the whole cluster appear as one big database.

What is sharding?

- Use multiple servers and distribute the data.
- Let the whole cluster appear as one big database.
- Heterogeneous and decentralised infrastructure possible.

What is sharding?

- Use multiple servers and distribute the data.
- Let the whole cluster appear as one big database.
- Heterogeneous and decentralised infrastructure possible.

Why do we need it?

- Big data — not enough space

What is sharding?

- Use multiple servers and distribute the data.
- Let the whole cluster appear as one big database.
- Heterogeneous and decentralised infrastructure possible.

Why do we need it?

- Big data — not enough space
- Many (write) requests — not enough bandwidth

What is sharding?

- Use multiple servers and distribute the data.
- Let the whole cluster appear as one big database.
- Heterogeneous and decentralised infrastructure possible.

Why do we need it?

- Big data — not enough space
- Many (write) requests — not enough bandwidth
- Be elastic — add or remove resources swiftly

What is sharding?

- Use multiple servers and distribute the data.
- Let the whole cluster appear as one big database.
- Heterogeneous and decentralised infrastructure possible.

Why do we need it?

- Big data — not enough space
- Many (write) requests — not enough bandwidth
- Be elastic — add or remove resources swiftly

Horizontal Scaling — scale out

Do you need sharding???

Do you need sharding???

Probably not!

Do you need sharding???

Probably not! — at this stage

Do you need sharding???

Probably not! — at this stage

If at all possible, you probably want

- to buy one or two big enough machines,

Do you need sharding???

Probably not! — at this stage

If at all possible, you probably want

- to buy one or two big enough machines,
- providing a reserve of about a factor of 3 – 10.

Do you need sharding???

Probably not! — at this stage

If at all possible, you probably want

- to buy one or two big enough machines,
- providing a reserve of about a factor of 3 – 10.
- Possibly use replication for reliability.

Do you need sharding???

Probably not! — at this stage

If at all possible, you probably want

- to buy one or two big enough machines,
- providing a reserve of about a factor of 3 – 10.
- Possibly use replication for reliability.

However: Everybody wants to be the next WhatsApp . . .

Do you need sharding???

Probably not! — at this stage

If at all possible, you probably want

- to buy one or two big enough machines,
- providing a reserve of about a factor of 3 – 10.
- Possibly use replication for reliability.

However: Everybody wants to be the next WhatsApp . . .

$\Longrightarrow$ You want to use a database that can do sharding.

- is a multi-model database (document store & graph database),

ArangoDB

- is a multi-model database (document store & graph database),
- is open source and free (Apache 2 license),

ArangoDB

- is a multi-model database (document store & graph database),
- is open source and free (Apache 2 license),
- offers convenient queries (via HTTP/REST and AQL),

ArangoDB

- is a multi-model database (document store & graph database),
- is open source and free (Apache 2 license),
- offers convenient queries (via HTTP/REST and AQL),
- offers high performance and is memory efficient,

ArangoDB

- is a multi-model database (document store & graph database),
- is open source and free (Apache 2 license),
- offers convenient queries (via HTTP/REST and AQL),
- offers high performance and is memory efficient,
- uses JavaScript throughout (V8 built into server),

ArangoDB

- is a multi-model database (document store & graph database),
- is open source and free (Apache 2 license),
- offers convenient queries (via HTTP/REST and AQL),
- offers high performance and is memory efficient,
- uses JavaScript throughout (V8 built into server),
- doubles as a web and application server
  (API extendable by JavaScript code in the Foxx framework),

ArangoDB

- is a multi-model database (document store & graph database),
- is open source and free (Apache 2 license),
- offers convenient queries (via HTTP/REST and AQL),
- offers high performance and is memory efficient,
- uses JavaScript throughout (V8 built into server),
- doubles as a web and application server
  (API extendable by JavaScript code in the Foxx framework),
- offers many drivers for a wide range of languages,

ArangoDB

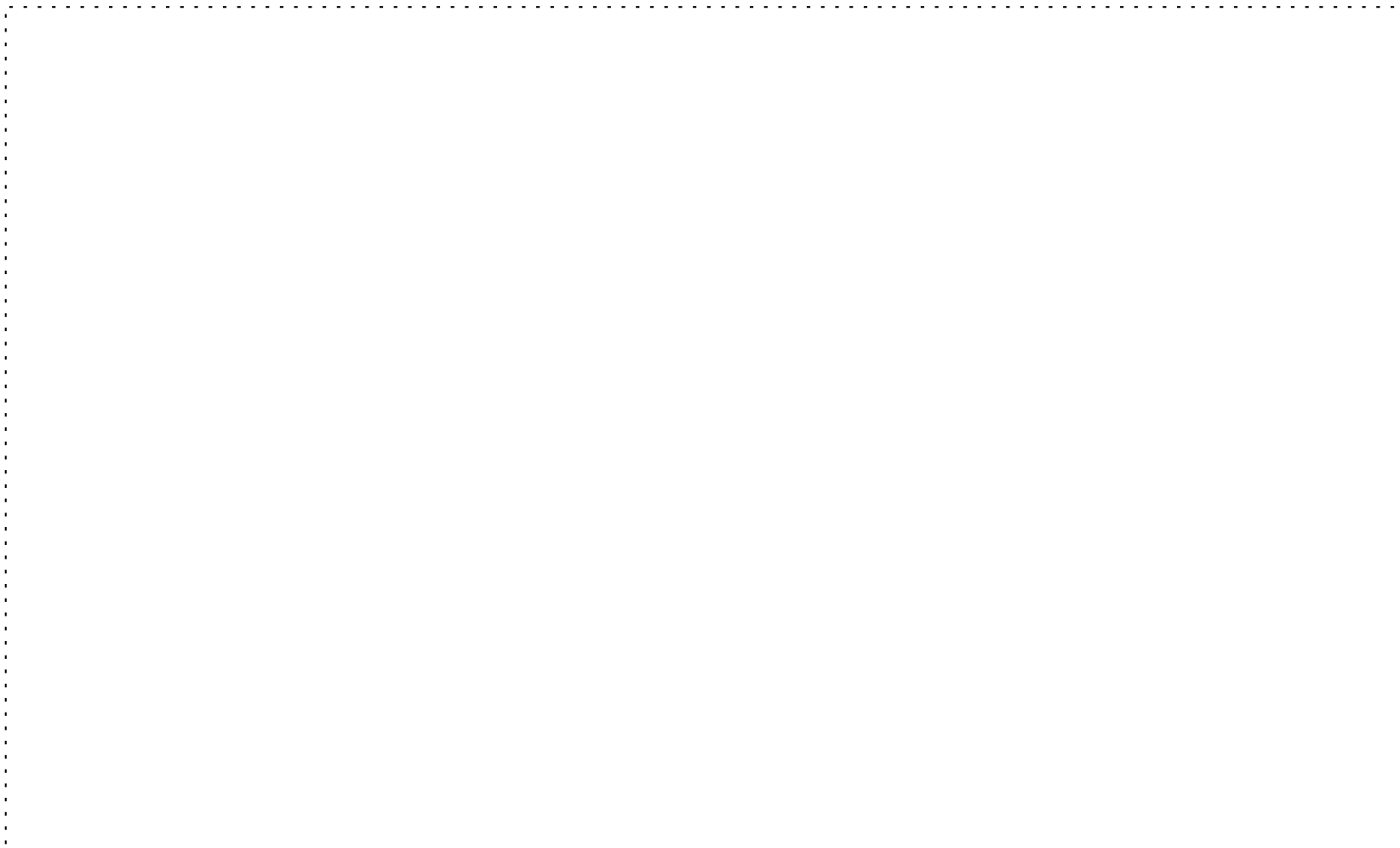
- is a multi-model database (document store & graph database),
- is open source and free (Apache 2 license),
- offers convenient queries (via HTTP/REST and AQL),
- offers high performance and is memory efficient,
- uses JavaScript throughout (V8 built into server),
- doubles as a web and application server
  (API extendable by JavaScript code in the Foxx framework),
- offers many drivers for a wide range of languages,
- is easy to use with web frontend and good documentation,

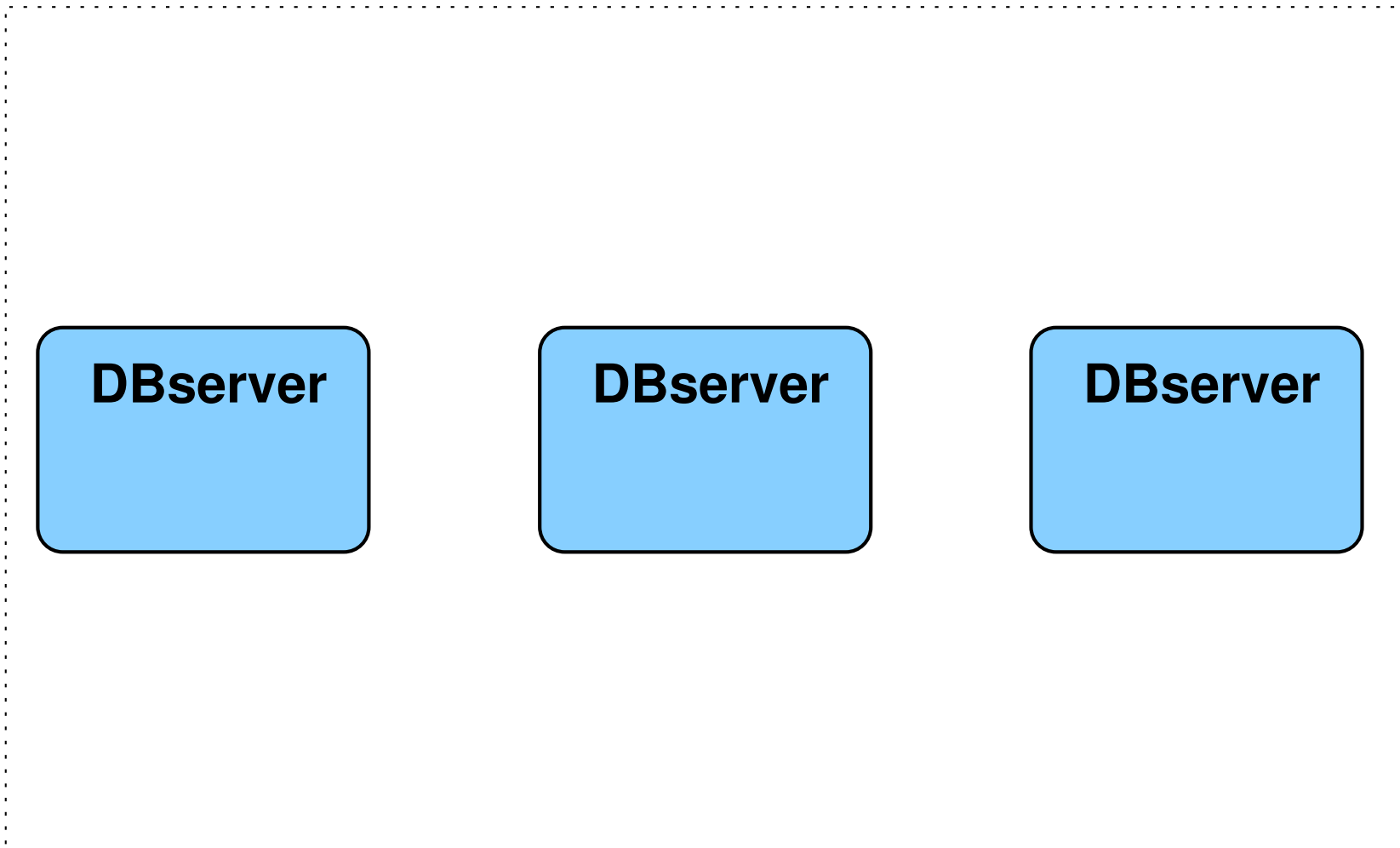
ArangoDB

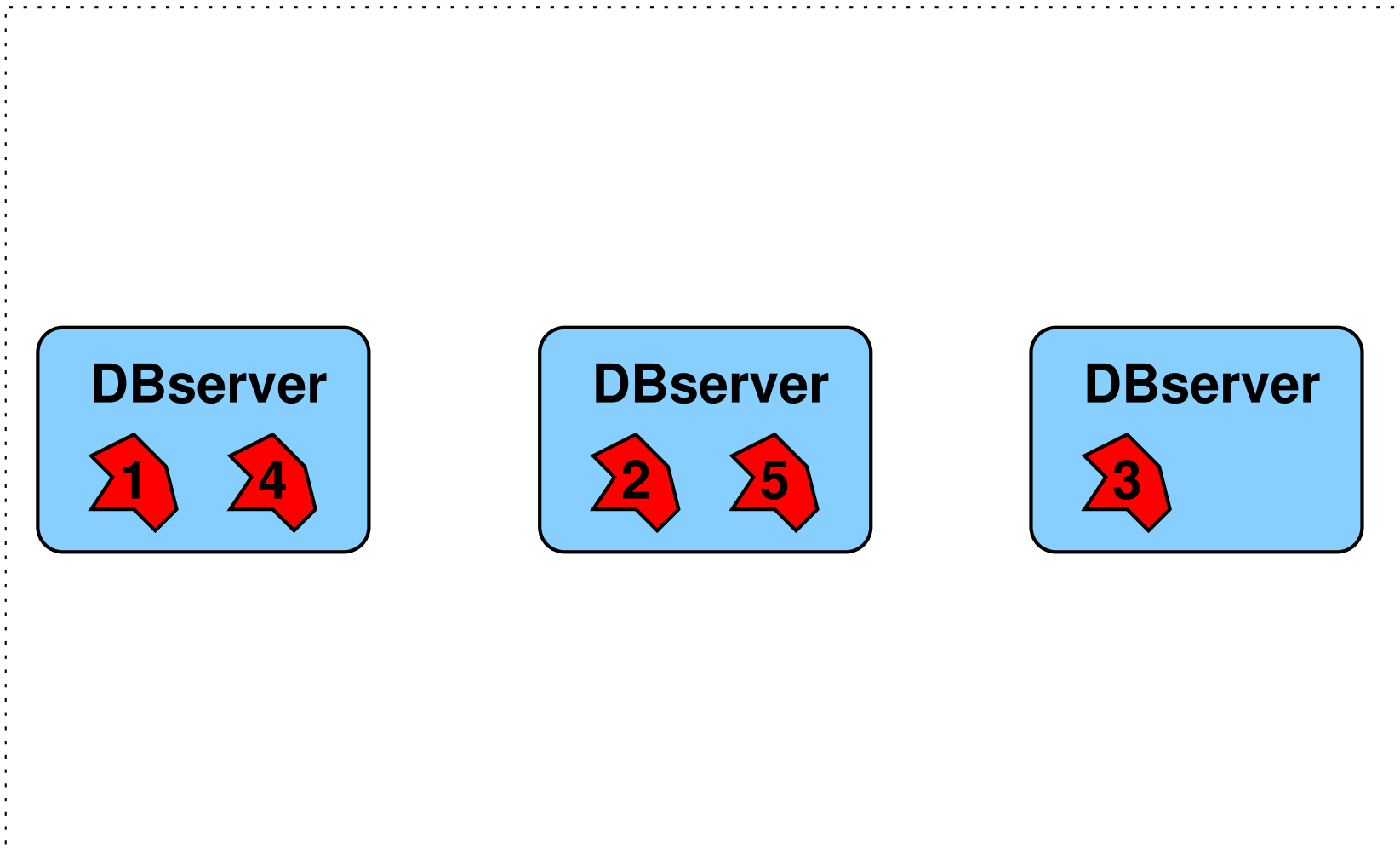
- is a multi-model database (document store & graph database),
- is open source and free (Apache 2 license),
- offers convenient queries (via HTTP/REST and AQL),
- offers high performance and is memory efficient,
- uses JavaScript throughout (V8 built into server),
- doubles as a web and application server
  (API extendable by JavaScript code in the Foxx framework),
- offers many drivers for a wide range of languages,
- is easy to use with web frontend and good documentation,
- enjoys good professional as well as community support, and ...

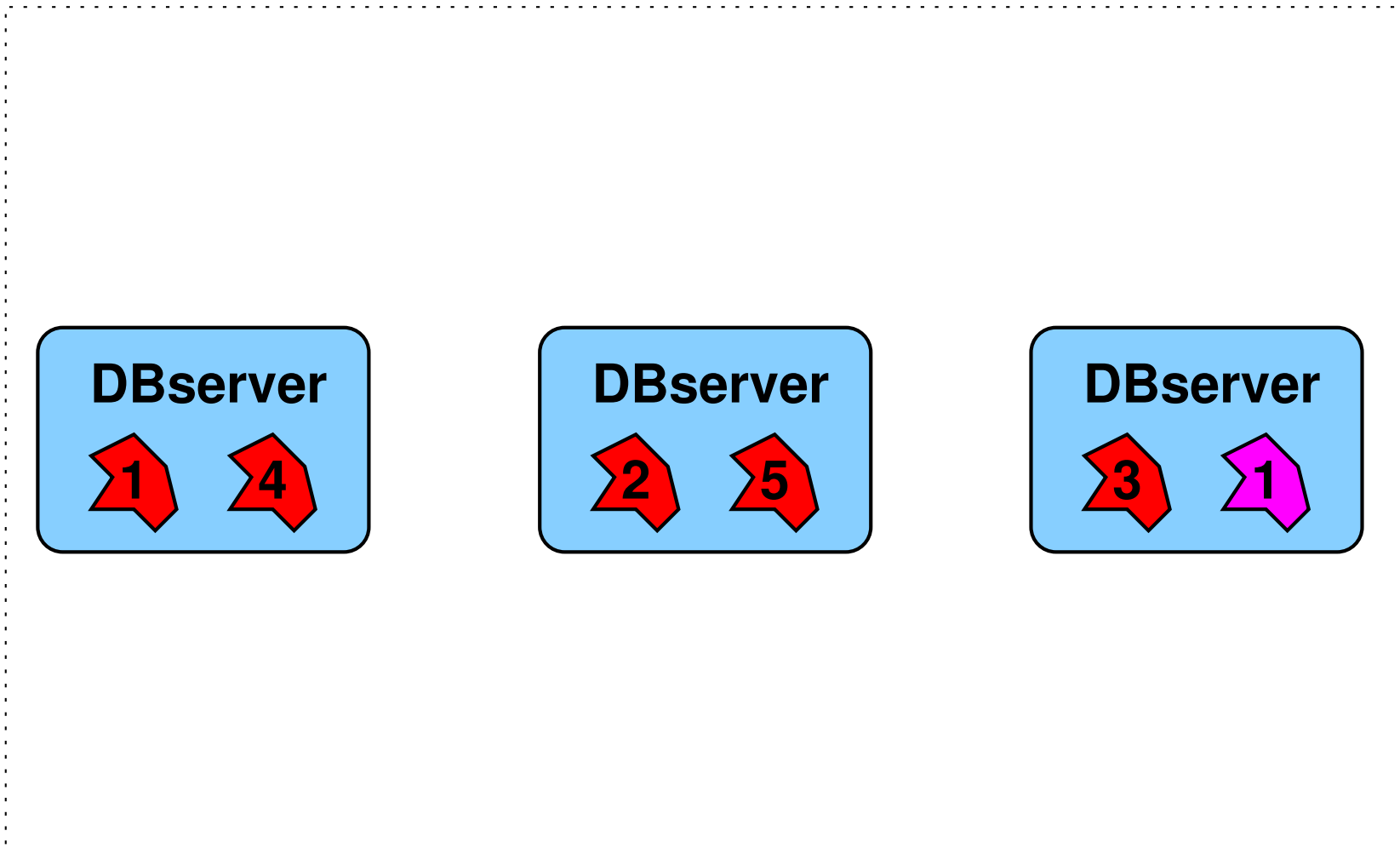
ArangoDB

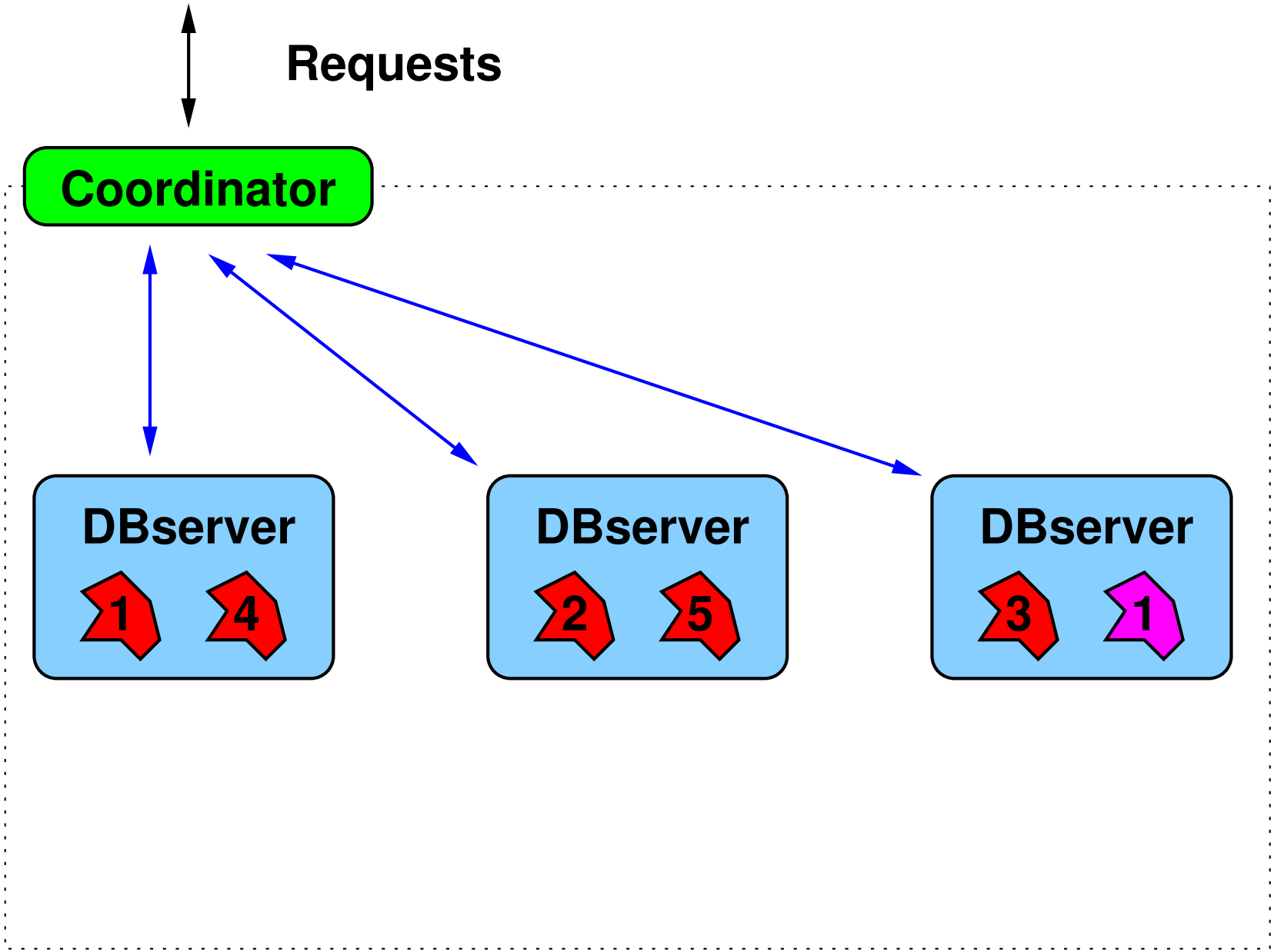
- is a multi-model database (document store & graph database),
- is open source and free (Apache 2 license),
- offers convenient queries (via HTTP/REST and AQL),
- offers high performance and is memory efficient,
- uses JavaScript throughout (V8 built into server),
- doubles as a web and application server
  (API extendable by JavaScript code in the Foxx framework),
- offers many drivers for a wide range of languages,
- is easy to use with web frontend and good documentation,
- enjoys good professional as well as community support, and ...
- ... recently added sharding in Version 2.0.

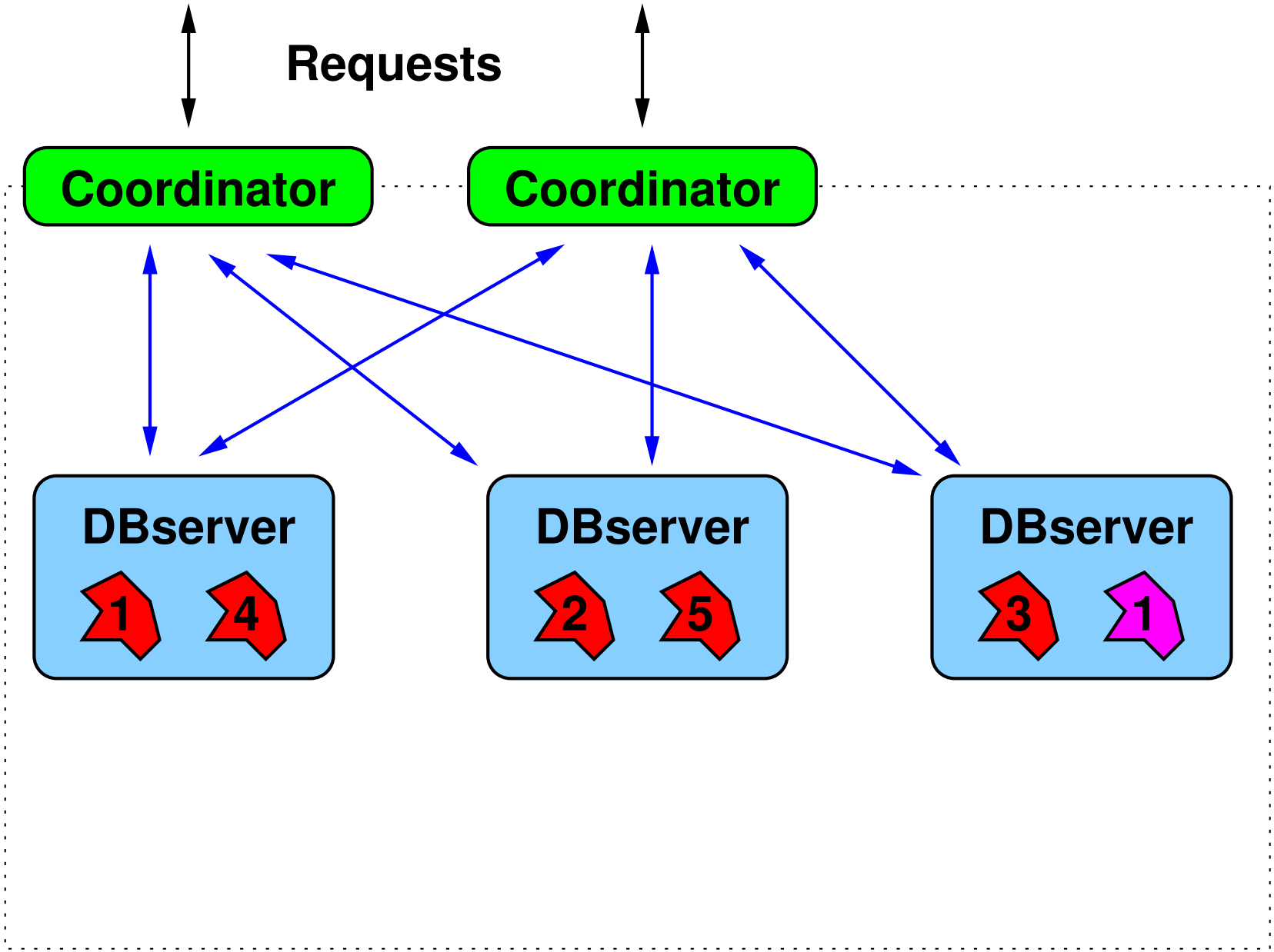


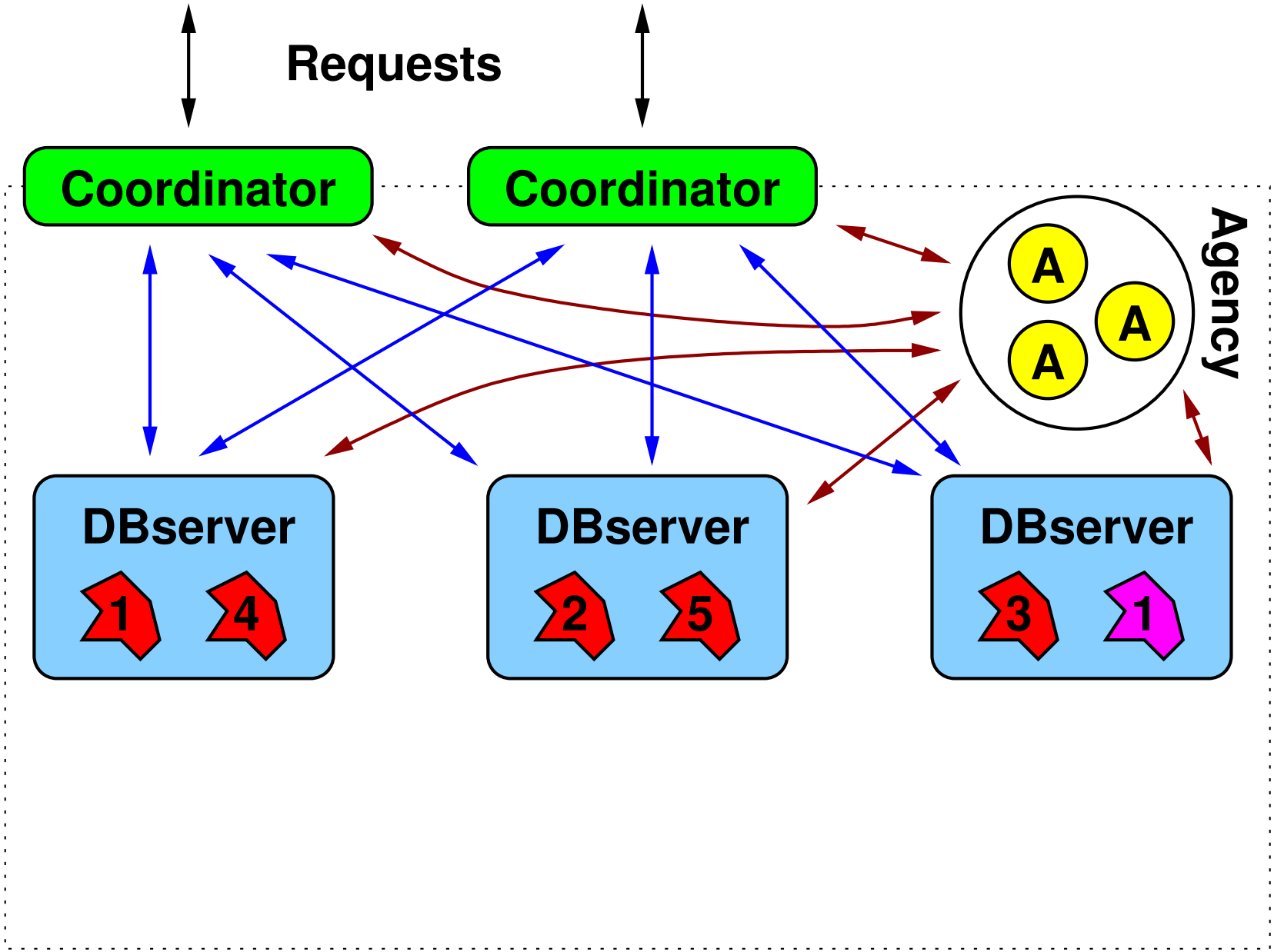

DBserver
DBserver
DBserver

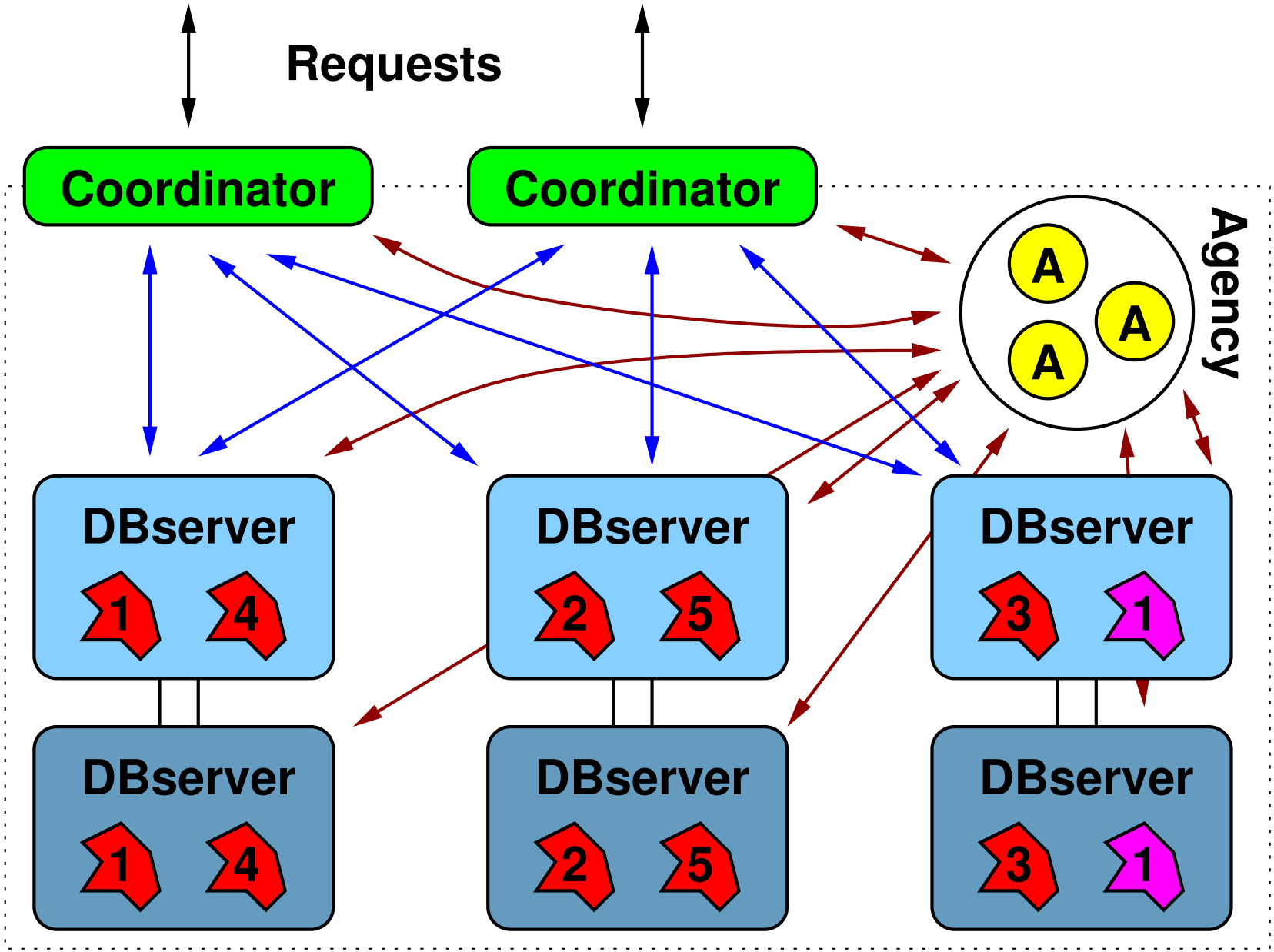

DBserver
1
4
DBserver
2
5
DBserver
3


DBserver
1
4
DBserver
2
5
DBserver
3
1

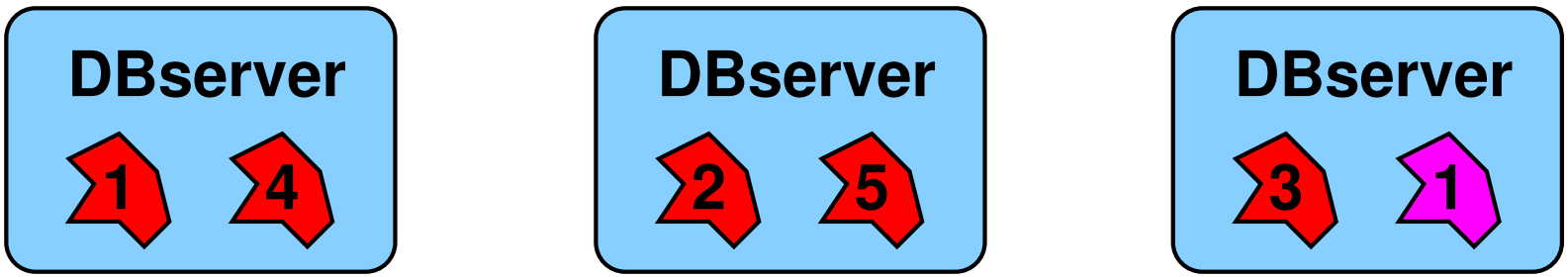

Requests
Coordinator
DBserver
1
4
DBserver
2
5
DBserver
3
1


Requests
Coordinator
Coordinator
DBserver
1
4
DBserver
2
5
DBserver
3
1


Requests
Coordinator
Coordinator
Agency
A
A
A
DBserver
1
4
DBserver
2
5
DBserver
3
1

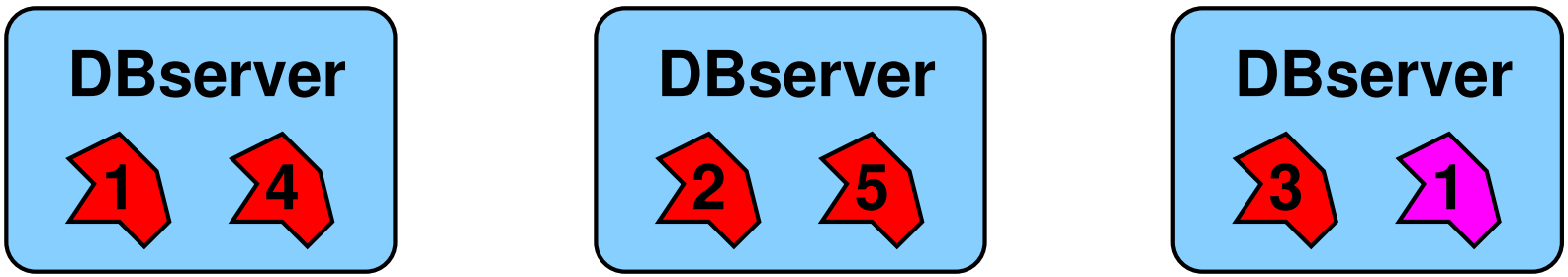

Requests
Coordinator
Coordinator
Agency
A
A
A
DBserver
1
4
DBserver
2
5
DBserver
3
1
DBserver
1
4
DBserver
2
5
DBserver
3
1

Sharded collections



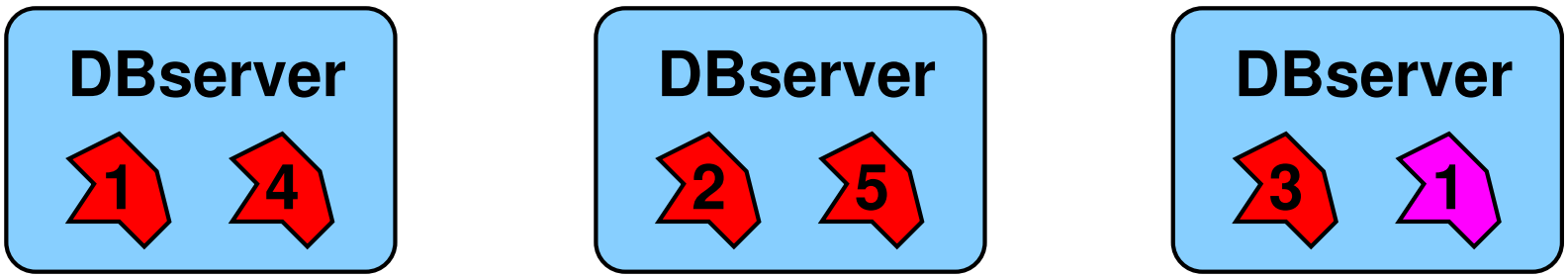
- A sharded collection consists of local collections (the shards) *distributed over the DBservers*.

Sharded collections



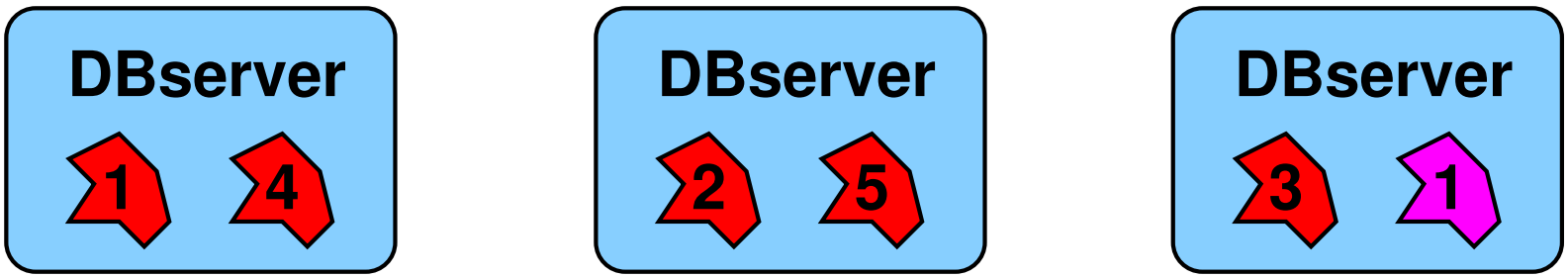
- A sharded collection consists of local collections (the shards) *distributed over the DBservers*.
- Each document in the collection is stored in exactly one shard.

Sharded collections



- A sharded collection consists of local collections (the shards) *distributed over the DBservers*.
- Each document in the collection is stored in exactly one shard.
- The shard is chosen by shard keys, which are one or more attributes in the documents (default: `_key` only).

Sharded collections



- A sharded collection consists of local collections (the shards) *distributed over the DBservers*.
- Each document in the collection is stored in exactly one shard.
- The shard is chosen by shard keys, which are one or more attributes in the documents (default: `_key` only).
- The coordinators redirect incoming queries to the right DBserver(s).

Reading a document in a sharded collection

Assume that we use `_key` as only shard key.

Reading a document in a sharded collection

Assume that we use `_key` as only shard key.

The **coordinator**

1. receives a GET request via HTTP with a `_key`,

Reading a document in a sharded collection

Assume that we use `_key` as only shard key.

The **coordinator**

1. receives a GET request via HTTP with a `_key`,
2. determines which shard is responsible,

Reading a document in a sharded collection

Assume that we use `_key` as only shard key.

The **coordinator**

1. receives a GET request via HTTP with a `_key`,
2. determines which shard is responsible,
3. forwards the GET to the responsible DBserver,
   (which retrieves the document (or fails)),

Reading a document in a sharded collection

Assume that we use `_key` as only shard key.

The **coordinator**

1. receives a GET request via HTTP with a `_key`,
2. determines which shard is responsible,
3. forwards the GET to the responsible DBserver,
   (which retrieves the document (or fails)),
4. receives the HTTP response, and

Reading a document in a sharded collection

Assume that we use `_key` as only shard key.

The **coordinator**

1. receives a GET request via HTTP with a `_key`,
2. determines which shard is responsible,
3. forwards the GET to the responsible DBserver, (which retrieves the document (or fails)),
4. receives the HTTP response, and
5. returns it to the client.

Reading a document in a sharded collection

Assume that we use `_key` as only shard key.

The **coordinator**

1. receives a GET request via HTTP with a `_key`,
2. determines which shard is responsible,
3. forwards the GET to the responsible DBserver, (which retrieves the document (or fails)),
4. receives the HTTP response, and
5. returns it to the client.

More difficult case

If we use other shard keys than `_key`, it needs to ask every shard!

Updating a document in a sharded collection

Assume we use some shard keys other than `_key`. The coordinator

1. receives a PUT or PATCH request with a changed document,

Updating a document in a sharded collection

Assume we use some shard keys other than `_key`. The coordinator

1. receives a PUT or PATCH request with a changed document,
2. determines which shard is responsible,

Updating a document in a sharded collection

Assume we use some shard keys other than `_key`. The coordinator

1. receives a PUT or PATCH request with a changed document,
2. determines which shard is responsible,
3. forwards the request to the responsible DBserver,
   (which tries to replace/update the document) and

Updating a document in a sharded collection

Assume we use some shard keys other than `_key`. The coordinator

1. receives a PUT or PATCH request with a changed document,
2. determines which shard is responsible,
3. forwards the request to the responsible DBserver, (which tries to replace/update the document) and
4. receives the HTTP response.

Updating a document in a sharded collection

Assume we use some shard keys other than `_key`. The coordinator

1. receives a PUT or PATCH request with a changed document,
2. determines which shard is responsible,
3. forwards the request to the responsible DBserver, (which tries to replace/update the document) and
4. receives the HTTP response.
5. If the answer is “do not know document”, the coordinator must still ask all shards, understand the answers, and

Updating a document in a sharded collection

Assume we use some shard keys other than `_key`. The coordinator

1. receives a PUT or PATCH request with a changed document,
2. determines which shard is responsible,
3. forwards the request to the responsible DBserver, (which tries to replace/update the document) and
4. receives the HTTP response.
5. If the answer is "do not know document", the coordinator must still ask all shards, understand the answers, and
6. finally responds to the client.

Updating a document in a sharded collection

Assume we use some shard keys other than `_key`. The coordinator

1. receives a PUT or PATCH request with a changed document,
2. determines which shard is responsible,
3. forwards the request to the responsible DBserver, (which tries to replace/update the document) and
4. receives the HTTP response.
5. If the answer is "do not know document", the coordinator must still ask all shards, understand the answers, and
6. finally responds to the client.

Easier case

If we use `_key` as only shard key, it only needs to ask one shard!

Querying byExample

The **coordinator**

1. receives a PUT request via HTTP with an example document,

Querying byExample

The **coordinator**

1. receives a PUT request via HTTP with an example document,
2. forwards the PUT to all shards,
   (which run the query and each build a cursor),

Querying byExample

The **coordinator**

1. receives a PUT request via HTTP with an example document,
2. forwards the PUT to all shards,
   (which run the query and each build a cursor),
3. receives all the HTTP responses,

Querying byExample

The **coordinator**

1. receives a PUT request via HTTP with an example document,
2. forwards the PUT to all shards,
   (which run the query and each build a cursor),
3. receives all the HTTP responses,
4. builds a meta-cursor keeping the info about all internal cursors,

Querying byExample

The **coordinator**

1. receives a PUT request via HTTP with an example document,
2. forwards the PUT to all shards,
   (which run the query and each build a cursor),
3. receives all the HTTP responses,
4. builds a meta-cursor keeping the info about all internal cursors,
5. responds to the client and

Querying byExample

The **coordinator**

1. receives a PUT request via HTTP with an example document,
2. forwards the PUT to all shards,
   (which run the query and each build a cursor),
3. receives all the HTTP responses,
4. builds a meta-cursor keeping the info about all internal cursors,
5. responds to the client and
6. subsequently serves the meta-cursor using all internal cursors
   one after another.

Querying byExample

The **coordinator**

1. receives a PUT request via HTTP with an example document,
2. forwards the PUT to all shards,
   (which run the query and each build a cursor),
3. receives all the HTTP responses,
4. builds a meta-cursor keeping the info about all internal cursors,
5. responds to the client and
6. subsequently serves the meta-cursor using all internal cursors
   one after another.

AQL queries are much more complicated, but handled similarly.

Consistency

Question

Which state does the client see?

Consistency

Question

Which state does the client see?

Answer (currently!)

Each DBserver takes a snapshot for its internal cursor!

Consistency

Question

Which state does the client see?

Answer (currently!)

Each DBserver takes a snapshot for its internal cursor!

Problem

It is hard to take consistent snapshots on different servers!

Consistency

Question

Which state does the client see?

Answer (currently!)

Each DBserver takes a snapshot for its internal cursor!

Problem

It is hard to take consistent snapshots on different servers!

Solution

Eventually, we will use Multi Version Concurrency Control (MVCC) to provide a consistent snapshot across the whole database.

Consistency

Question

Which state does the client see?

Answer (currently!)

Each DBserver takes a snapshot for its internal cursor!

Problem

It is hard to take consistent snapshots on different servers!

Solution (at least most of it)

Eventually, we will use Multi Version Concurrency Control (MVCC) to provide a consistent snapshot across the whole database.

Consistency

Question

Which state does the client see?

Answer (currently!)

Each DBserver takes a snapshot for its internal cursor!

Problem

It is hard to take consistent snapshots on different servers!

Solution (at least most of it)

Eventually, we will use Multi Version Concurrency Control (MVCC) to provide a consistent snapshot across the whole database.
This brings us to TRANSACTIONS . . .

Transactions . . .

Transactions . . .

. . . are hard in a distributed database.

Transactions . . .

. . . are hard in a distributed database.

Questions

What state does a transaction see?

Transactions . . .

. . . are hard in a distributed database.

Questions

What state does a transaction see?
What can it modify?

Transactions . . .

. . . are hard in a distributed database.

Questions

What state does a transaction see?
What can it modify?
Who sees these modifications when?

Transactions . . .

. . . are hard in a distributed database.

Questions

What state does a transaction see?
What can it modify?
Who sees these modifications when?
What means “when” and what is time anyway?

Transactions . . .

. . . are hard in a distributed database.

Questions

What state does a transaction see?
What can it modify?
Who sees these modifications when?
What means “when” and what is time anyway?

$\Longrightarrow$ Need global transaction management:

Transactions . . .

. . . are hard in a distributed database.

Questions

What state does a transaction see?
What can it modify?
Who sees these modifications when?
What means “when” and what is time anyway?

$\Longrightarrow$ Need global transaction management:

- a transaction sees a snapshot of the DB at its start time and

Transactions . . .

. . . are hard in a distributed database.

Questions

What state does a transaction see?
What can it modify?
Who sees these modifications when?
What means “when” and what is time anyway?

$\Longrightarrow$ Need global transaction management:

- a transaction sees a snapshot of the DB at its start time and
- may change all docs that have not been changed since then.

Transactions . . .

. . . are hard in a distributed database.

Questions

What state does a transaction see?
What can it modify?
Who sees these modifications when?
What means “when” and what is time anyway?

$\Longrightarrow$ Need global transaction management:

- a transaction sees a snapshot of the DB at its start time and
- may change all docs that have not been changed since then.
- $\Longrightarrow$ every document revision has an associated transaction, and

Transactions . . .

. . . are hard in a distributed database.

Questions

What state does a transaction see?
What can it modify?
Who sees these modifications when?
What means “when” and what is time anyway?

$\Longrightarrow$ Need global transaction management:

- a transaction sees a snapshot of the DB at its start time and
- may change all docs that have not been changed since then.
- $\Longrightarrow$ every document revision has an associated transaction, and
- the transaction manager must be able to order transactions by their start time.

Transactions . . .

. . . are hard in a distributed database.

Questions

What state does a transaction see?
What can it modify?
Who sees these modifications when?
What means “when” and what is time anyway?

$\Longrightarrow$ Need global transaction management:

- a transaction sees a snapshot of the DB at its start time and
- may change all docs that have not been changed since then.
- $\Longrightarrow$ every document revision has an associated transaction, and
- the transaction manager must be able to order transactions by their start time.
- It is only asked, when two transactions try to modify the same doc.

Our approach to transactions in a cluster

Our approach to transactions in a cluster

- **In Version 2.0**, transactions in a cluster are just fake.

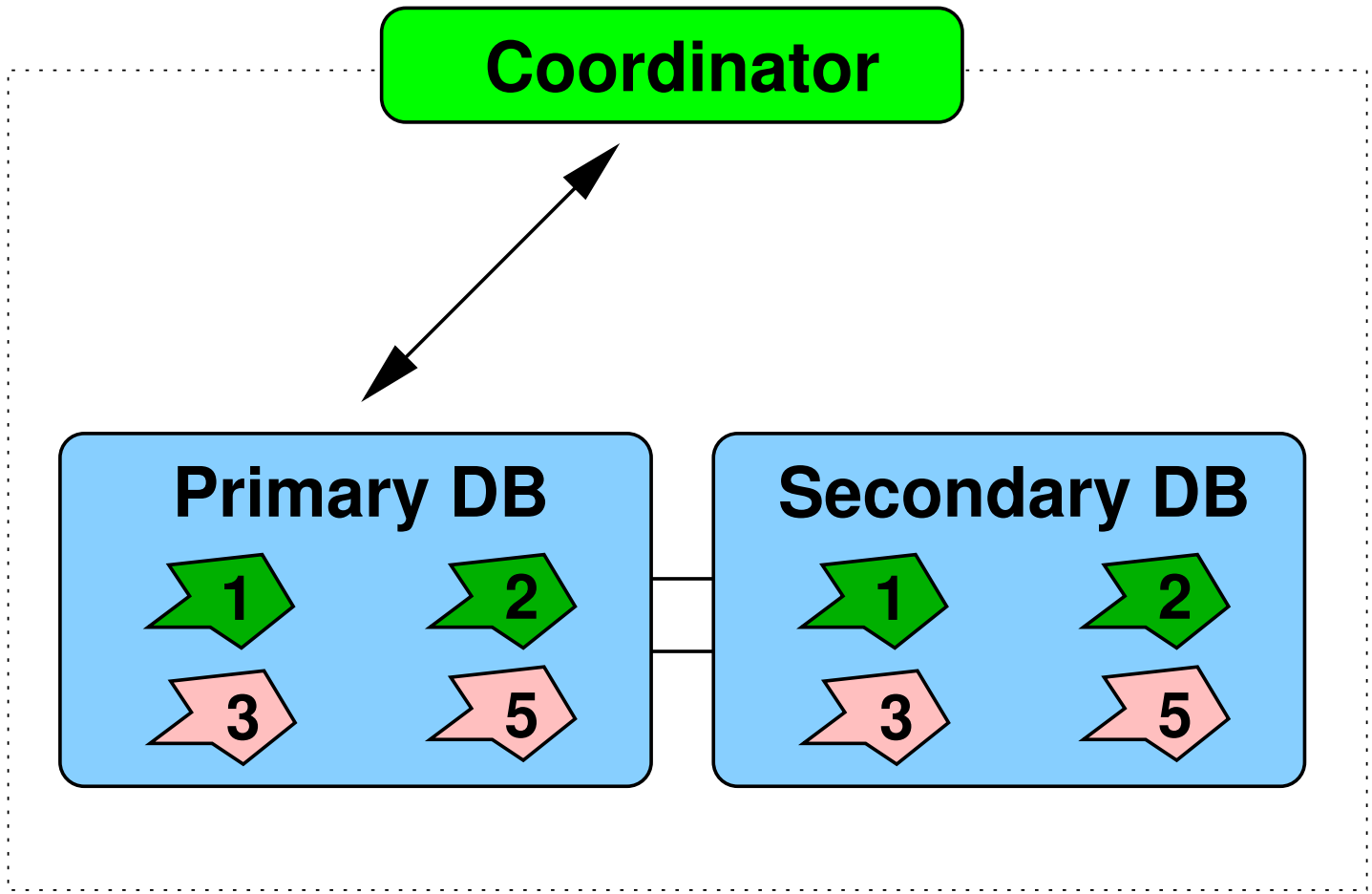
Our approach to transactions in a cluster

- **In Version 2.0**, transactions in a cluster are just fake.
- **Later**, we will temporarily relax isolation: “eventual atomicity”.

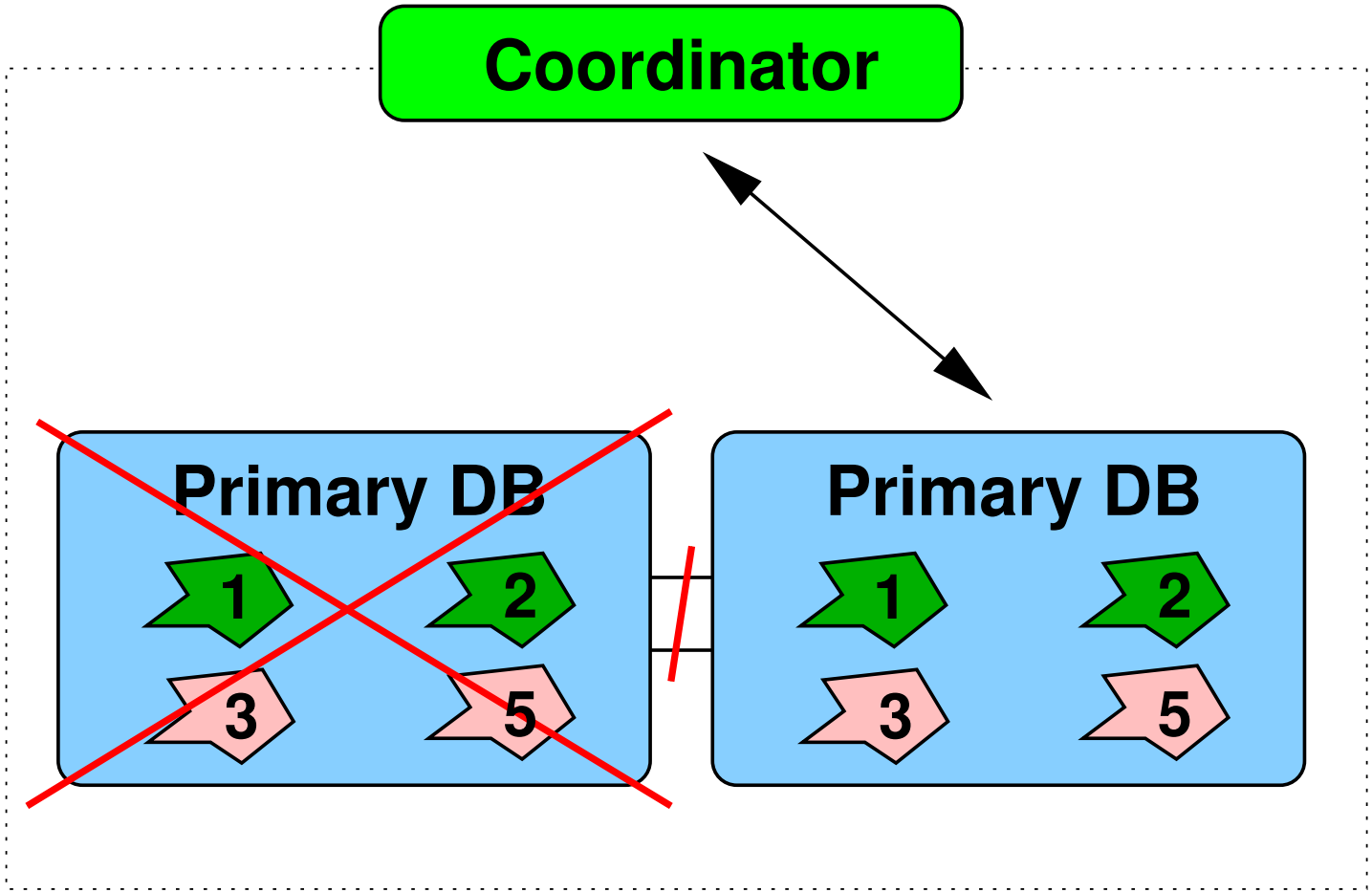
Our approach to transactions in a cluster

- **In Version 2.0**, transactions in a cluster are just fake.
- **Later**, we will temporarily relax isolation: “eventual atomicity”.
- **Eventually**, we will implement full transaction semantics in the MVCC sense.

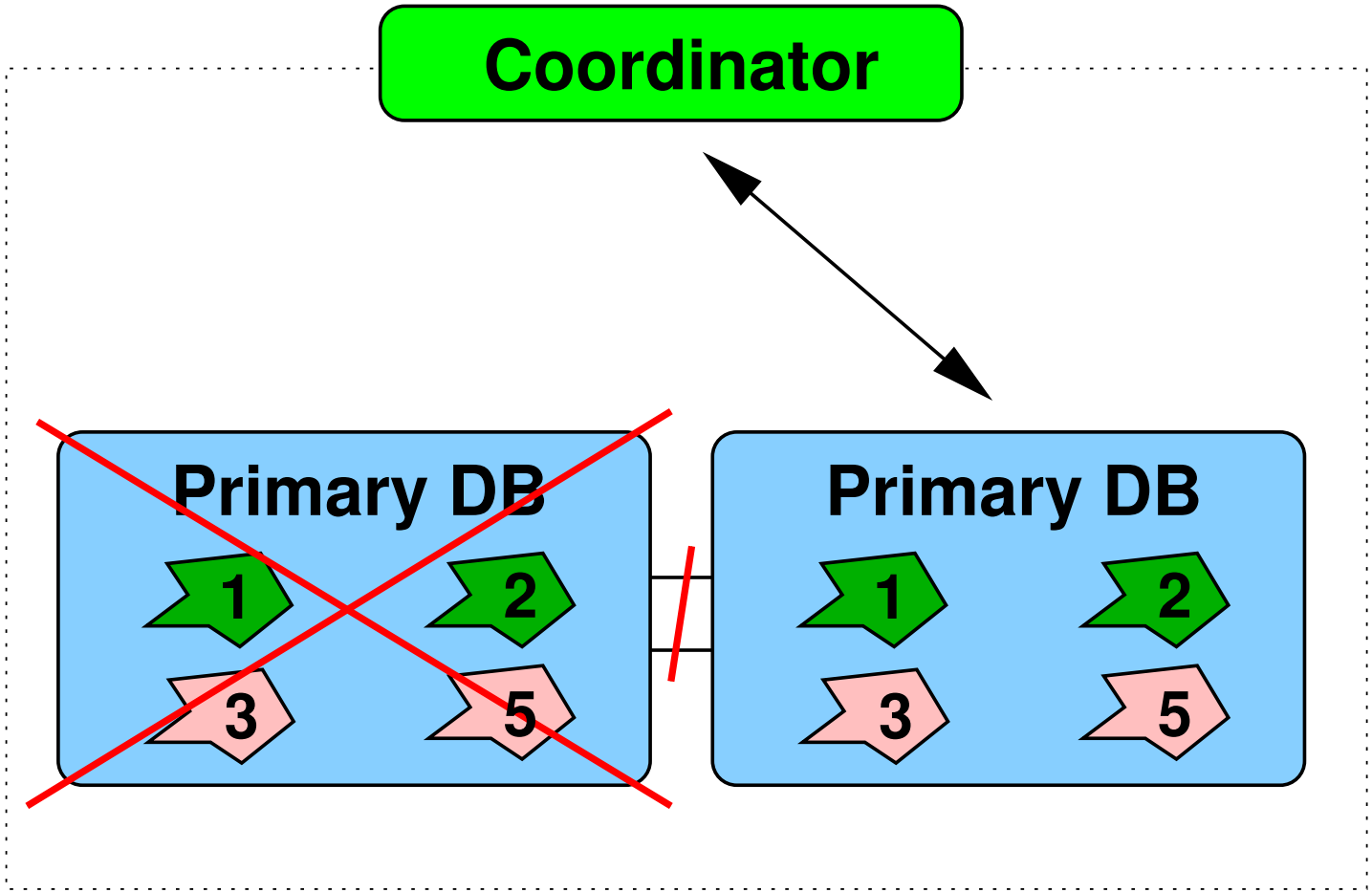
Fault tolerance and automatic failover



Fault tolerance and automatic failover



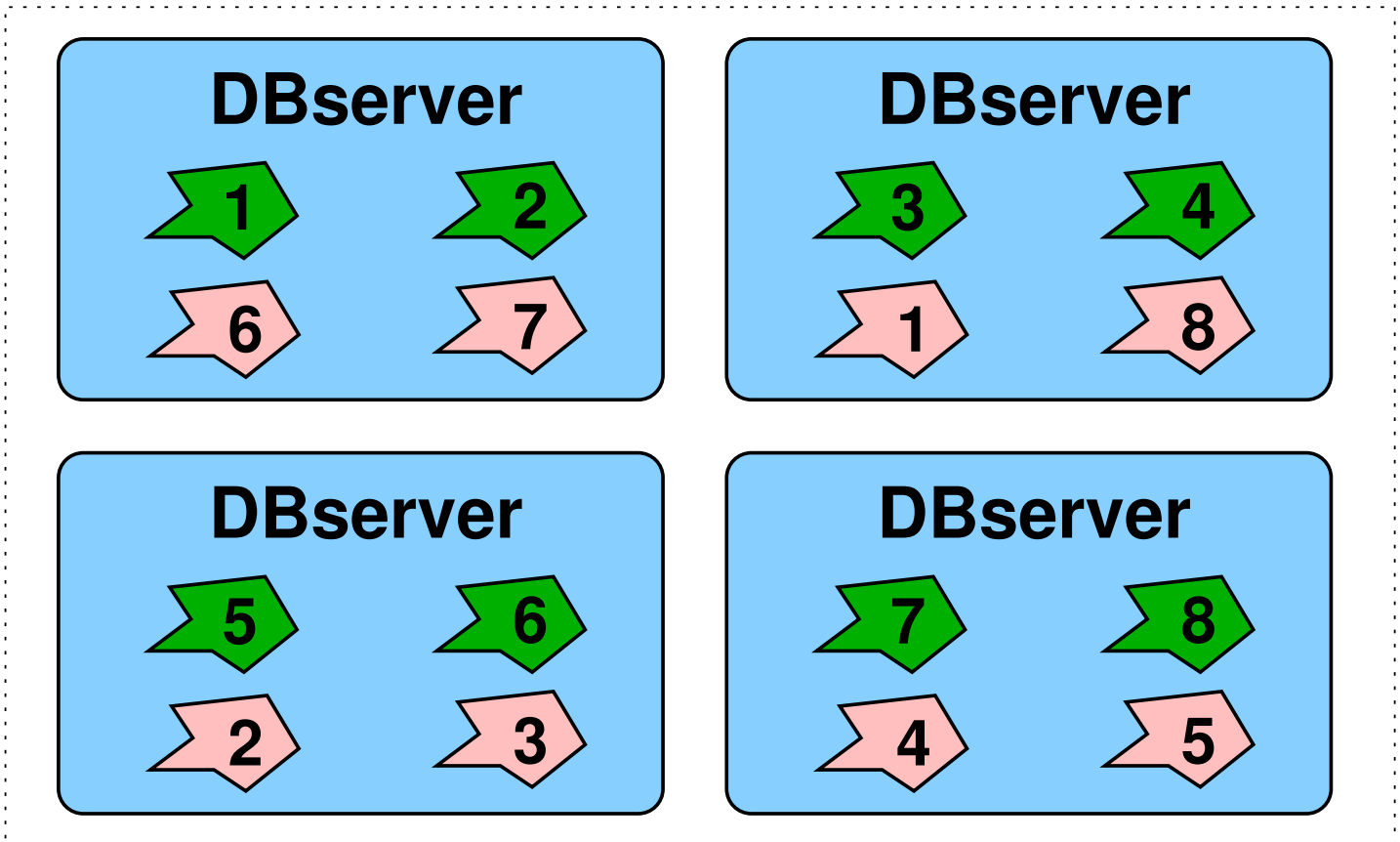
Fault tolerance and automatic failover



Synchronisation and failover organisation is done via the agency.

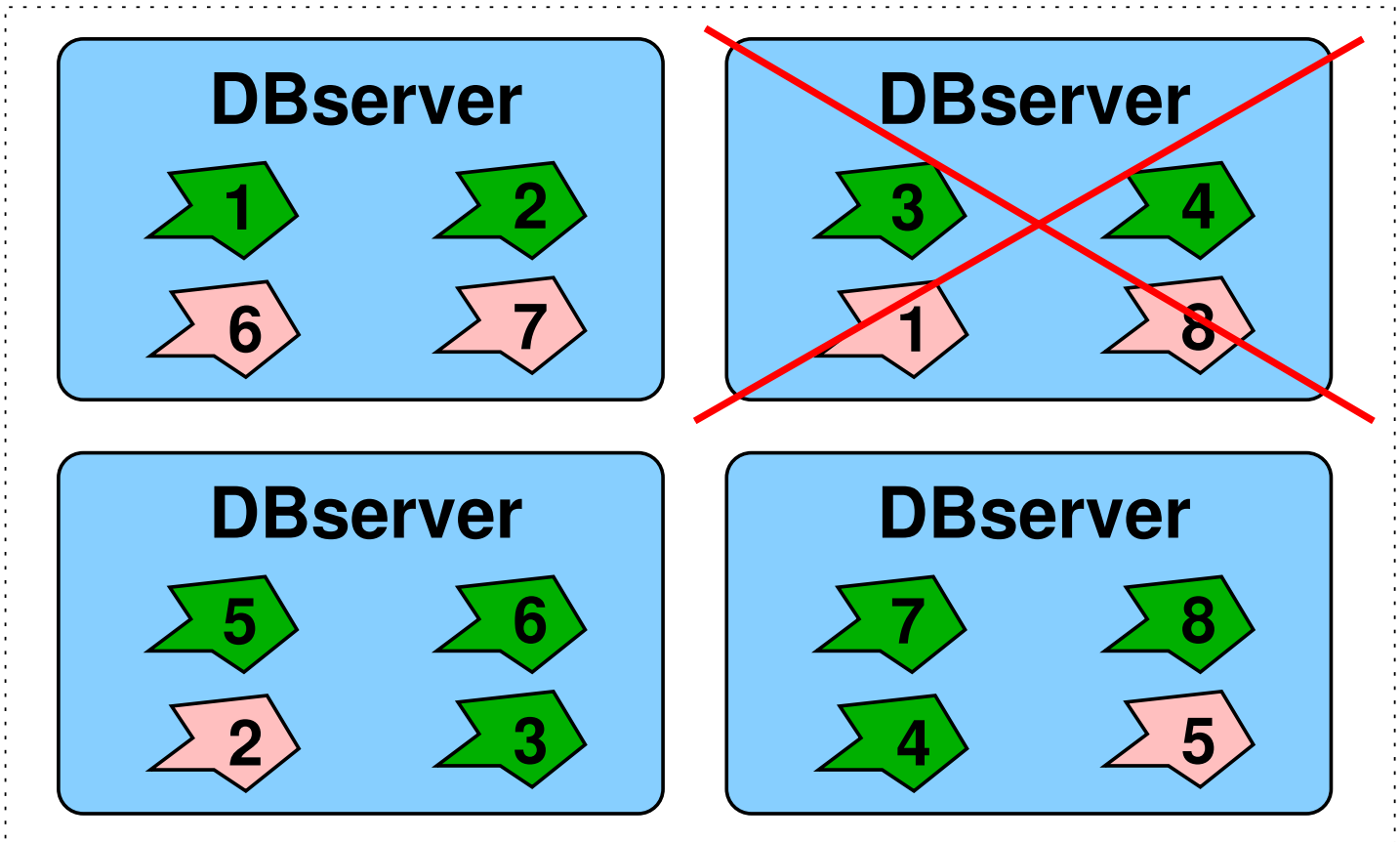
Distributed failover

Green: primary shard instances **Pink:** secondary shard instances



Distributed failover

Green: primary shard instances **Pink:** secondary shard instances



If any one server fails, all shards are still there, and the load is still relatively evenly distributed.

Done in Version 2.0 (released end of February 2014)

- Primary DBservers, coordinators, sharded collections

Done in Version 2.0 (released end of February 2014)

- Primary DBservers, coordinators, sharded collections
- Maintenance of databases and collections

Done in Version 2.0 (released end of February 2014)

- Primary DBservers, coordinators, sharded collections
- Maintenance of databases and collections
- CRUD, SimpleQueries, AQL (potentially inefficient)

Done in Version 2.0 (released end of February 2014)

- Primary DBservers, coordinators, sharded collections
- Maintenance of databases and collections
- CRUD, SimpleQueries, AQL (potentially inefficient)
- Cluster planner and kickstarter libraries

Done in Version 2.0 (released end of February 2014)

- Primary DBservers, coordinators, sharded collections
- Maintenance of databases and collections
- CRUD, SimpleQueries, AQL (potentially inefficient)
- Cluster planner and kickstarter libraries
- Graphical user interface to cluster planning/launching/shutdown

Done in Version 2.0 (released end of February 2014)

- Primary DBservers, coordinators, sharded collections
- Maintenance of databases and collections
- CRUD, SimpleQueries, AQL (potentially inefficient)
- Cluster planner and kickstarter libraries
- Graphical user interface to cluster planning/launching/shutdown
- Graphical dash board for cluster state monitoring

Done in Version 2.0 (released end of February 2014)

- Primary DBservers, coordinators, sharded collections
- Maintenance of databases and collections
- CRUD, SimpleQueries, AQL (potentially inefficient)
- Cluster planner and kickstarter libraries
- Graphical user interface to cluster planning/launching/shutdown
- Graphical dash board for cluster state monitoring
- Authentication with and in cluster, including SSL

Todo (see roadmap for later this year)

- Dump and restore for clusters

Todo (see roadmap for later this year)

- Dump and restore for clusters
- Secondary DBservers, synchronous replication, automatic failover

Todo (see roadmap for later this year)

- Dump and restore for clusters
- Secondary DBservers, synchronous replication, automatic failover
- Performance tuning for AQL

Todo (see roadmap for later this year)

- Dump and restore for clusters
- Secondary DBservers, synchronous replication, automatic failover
- Performance tuning for AQL
- Elastic resizing of clusters, automatic redistribution of shards

Todo (see roadmap for later this year)

- Dump and restore for clusters
- Secondary DBservers, synchronous replication, automatic failover
- Performance tuning for AQL
- Elastic resizing of clusters, automatic redistribution of shards
- Writing AQL requests

Todo (see roadmap for later this year)

- Dump and restore for clusters
- Secondary DBservers, synchronous replication, automatic failover
- Performance tuning for AQL
- Elastic resizing of clusters, automatic redistribution of shards
- Writing AQL requests
- More flexible authentication using tokens

Todo (see roadmap for later this year)

- Dump and restore for clusters
- Secondary DBservers, synchronous replication, automatic failover
- Performance tuning for AQL
- Elastic resizing of clusters, automatic redistribution of shards
- Writing AQL requests
- More flexible authentication using tokens
- Eventually atomic transactions

Todo (see roadmap for later this year)

- Dump and restore for clusters
- Secondary DBservers, synchronous replication, automatic failover
- Performance tuning for AQL
- Elastic resizing of clusters, automatic redistribution of shards
- Writing AQL requests
- More flexible authentication using tokens
- Eventually atomic transactions
- Fully isolated transactions

Todo (see roadmap for later this year)

- Dump and restore for clusters
- Secondary DBservers, synchronous replication, automatic failover
- Performance tuning for AQL
- Elastic resizing of clusters, automatic redistribution of shards
- Writing AQL requests
- More flexible authentication using tokens
- Eventually atomic transactions
- Fully isolated transactions
- Distributed failover

Todo (see roadmap for later this year)

- Dump and restore for clusters
- Secondary DBservers, synchronous replication, automatic failover
- Performance tuning for AQL
- Elastic resizing of clusters, automatic redistribution of shards
- Writing AQL requests
- More flexible authentication using tokens
- Eventually atomic transactions
- Fully isolated transactions
- Distributed failover

Stay tuned for new releases later this year!