

AQL under the hood

ArangoDB workshop at JPL, 24–27 October 2016

Max Neunhöffer

Documents and collections

```
{
  "_key": "123456",
  "_id": "chars/123456",
  "name": "Duck",
  "firstname": "Donald",
  "dob": "1934-11-13",
  "hobbies": ["Golf",
              "Singing",
              "Running"],
  "home": {
    "town": "Duck town",
    "street": "Lake Road",
    "number": 17},
  "species": "duck"
}
```

Documents and collections

```
{
  "_key": "123456",
  "_id": "chars/123456",
  "name": "Duck",
  "firstname": "Donald",
  "dob": "1934-11-13",
  "hobbies": ["Golf",
              "Singing",
              "Running"],
  "home": {
    "town": "Duck town",
    "street": "Lake Road",
    "number": 17},
  "species": "duck"
}
```

When I say “document”,
I mean “JSON”.

Documents and collections

```
{
  "_key": "123456",
  "_id": "chars/123456",
  "name": "Duck",
  "firstname": "Donald",
  "dob": "1934-11-13",
  "hobbies": ["Golf",
              "Singing",
              "Running"],
  "home": {
    "town": "Duck town",
    "street": "Lake Road",
    "number": 17},
  "species": "duck"
}
```

When I say “document”,
I mean “JSON”.

A “collection” is a set of
documents in a DB.

Documents and collections

```
{
  "_key": "123456",
  "_id": "chars/123456",
  "name": "Duck",
  "firstname": "Donald",
  "dob": "1934-11-13",
  "hobbies": ["Golf",
              "Singing",
              "Running"],
  "home": {
    "town": "Duck town",
    "street": "Lake Road",
    "number": 17},
  "species": "duck"
}
```

When I say “document”,
I mean “JSON”.

A “collection” is a set of
documents in a DB.

The DB can inspect the
values, allowing for
secondary indexes.

Documents and collections

```
{
  "_key": "123456",
  "_id": "chars/123456",
  "name": "Duck",
  "firstname": "Donald",
  "dob": "1934-11-13",
  "hobbies": ["Golf",
              "Singing",
              "Running"],
  "home": {
    "town": "Duck town",
    "street": "Lake Road",
    "number": 17},
  "species": "duck"
}
```

When I say “document”,
I mean “JSON”.

A “collection” is a set of
documents in a DB.

The DB can inspect the
values, allowing for
secondary indexes.

Or one can just treat the
DB as a key/value store.

Documents and collections

```
{
  "_key": "123456",
  "_id": "chars/123456",
  "name": "Duck",
  "firstname": "Donald",
  "dob": "1934-11-13",
  "hobbies": ["Golf",
              "Singing",
              "Running"],
  "home": {
    "town": "Duck town",
    "street": "Lake Road",
    "number": 17},
  "species": "duck"
}
```

When I say “document”,
I mean “JSON”.

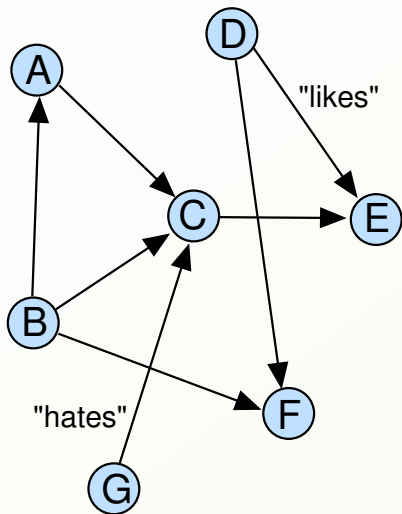
A “collection” is a set of
documents in a DB.

The DB can inspect the
values, allowing for
secondary indexes.

Or one can just treat the
DB as a key/value store.

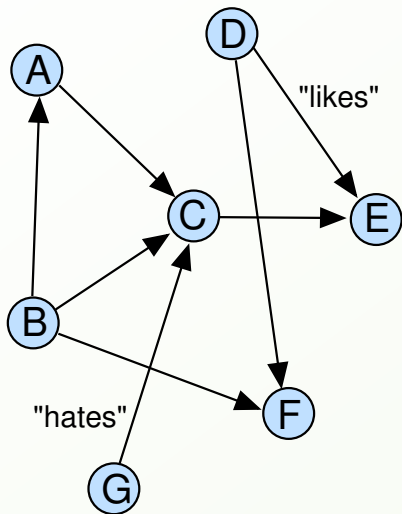
Sharding: the data of a
collection is distributed
between multiple servers.

Graphs

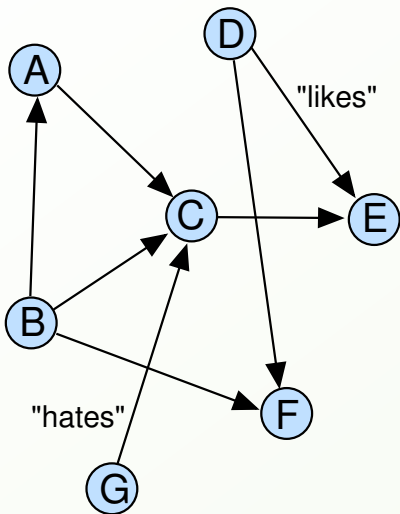


Graphs

A **graph** consists of **vertices** and **edges**.



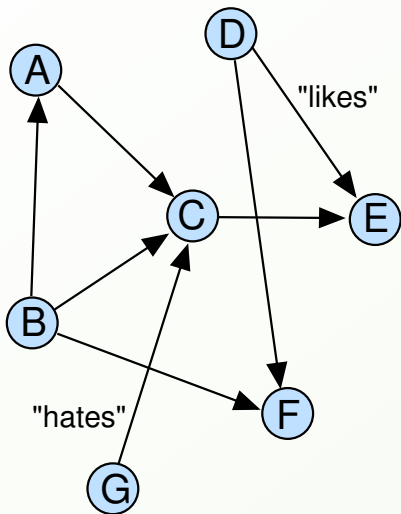
Graphs



A **graph** consists of **vertices** and **edges**.

Graphs **model relations**, can be directed or undirected.

Graphs

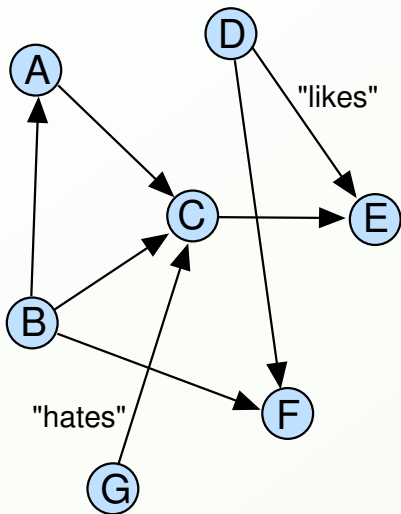


A **graph** consists of **vertices** and **edges**.

Graphs **model relations**, can be directed or undirected.

Vertices and **edges** are documents.

Graphs



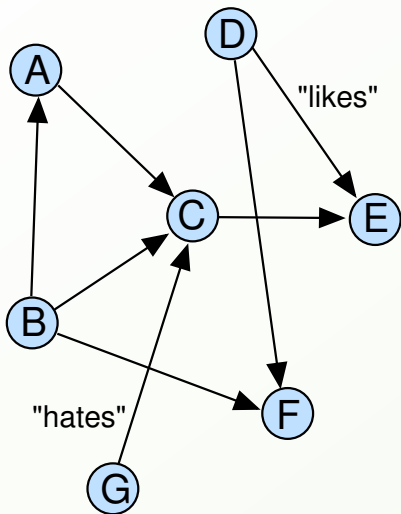
A **graph** consists of **vertices** and **edges**.

Graphs **model relations**, can be **directed** or **undirected**.

Vertices and **edges** are documents.

Every edge has a **_from** and a **_to** attribute.

Graphs



A **graph** consists of **vertices** and **edges**.

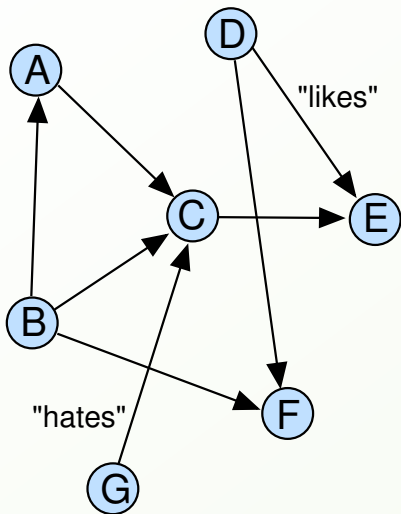
Graphs **model relations**, can be **directed** or **undirected**.

Vertices and **edges** are **documents**.

Every edge has a **_from** and a **_to** attribute.

The database offers **queries** and **transactions** dealing with **graphs**.

Graphs



A **graph** consists of **vertices** and **edges**.

Graphs **model relations**, can be **directed** or **undirected**.

Vertices and **edges** are documents.

Every edge has a **_from** and a **_to** attribute.

The database offers **queries** and **transactions** dealing with **graphs**.

For example, **paths in the graph** are interesting.

Query 1

Fetch all documents in a collection

```
FOR p IN people  
  RETURN p
```

Query 1

Fetch all documents in a collection

```
FOR p IN people  
  RETURN p
```

```
[ { "name": "Schmidt", "firstname": "Helmut",  
    "hobbies": ["Smoking"] },  
  { "name": "Neunhöffer", "firstname": "Max",  
    "hobbies": ["Piano", "Golf"] },  
  ...  
]
```


Query 1

Fetch all documents in a collection

```
FOR p IN people  
  RETURN p
```

```
[ { "name": "Schmidt", "firstname": "Helmut",  
    "hobbies": ["Smoking"] },  
  { "name": "Neunhöffer", "firstname": "Max",  
    "hobbies": ["Piano", "Golf"] },  
  ...  
]
```

(Actually, a cursor is returned.)

Query 2

Use filtering, sorting and limit

```
FOR p IN people
  FILTER p.age >= @minage
  SORT p.name, p.firstname
  LIMIT @nrlimit
  RETURN { name: CONCAT(p.name, ", ",
                        p.firstname),
          age : p.age }
```

Query 2

Use filtering, sorting and limit

```
FOR p IN people
  FILTER p.age >= @minage
  SORT p.name, p.firstname
  LIMIT @nrlimit
  RETURN { name: CONCAT(p.name, ", ",
                        p.firstname),
          age : p.age }
```

```
[ { "name": "Neunhöffer, Max", "age": 44 },
  { "name": "Schmidt, Helmut", "age": 95 },
  ...
]
```

Query 3

Aggregation and functions

```
FOR p IN people
  COLLECT a = p.age INTO L
  FILTER a >= @minage
  RETURN { "age": a, "number": LENGTH(L) }
```

Query 3

Aggregation and functions

```
FOR p IN people
  COLLECT a = p.age INTO L
  FILTER a >= @minage
  RETURN { "age": a, "number": LENGTH(L) }
```

```
[ { "age": 18, "number": 10 },
  { "age": 19, "number": 17 },
  { "age": 20, "number": 12 },
  ...
]
```

Query 4

Joins

```
FOR p IN @@peoplecollection
  FOR h IN houses
    FILTER p._key == h.owner
    SORT h.streetname, h.housename
    RETURN { housename: h.housename,
             streetname: h.streetname,
             owner: p.name,
             value: h.value }
```

Query 4

Joins

```
FOR p IN @@peoplecollection
  FOR h IN houses
    FILTER p._key == h.owner
    SORT h.streetname, h.housename
    RETURN { housename: h.housename,
             streetname: h.streetname,
             owner: p.name,
             value: h.value }
```

```
[ { "housename": "Firlefanfanz",
    "streetname": "Meyer street",
    "owner": "Hans Schmidt", "value": 423000
  },
  ...
]
```

Query 5

Modifying data

```
FOR e IN events
  FILTER e.timestamp < "2014-09-01T09:53+0200"
  INSERT e IN oldevents
```

```
FOR e IN events
  FILTER e.timestamp < "2014-09-01T09:53+0200"
  REMOVE e._key IN events
```


Query 6

Graph queries

```
FOR x IN GRAPH_SHORTEST_PATH(  
  "routeplanner", "germanCity/Cologne",  
  "frenchCity/Paris", {weight: "distance"} )  
RETURN { begin    : x.startVertex,  
        end      : x.vertex,  
        distance : x.distance,  
        nrPaths  : LENGTH(x.paths) }
```

Query 6

Graph queries

```
FOR x IN GRAPH_SHORTEST_PATH(  
  "routeplanner", "germanCity/Cologne",  
  "frenchCity/Paris", {weight: "distance"} )  
RETURN { begin    : x.startVertex,  
        end      : x.vertex,  
        distance : x.distance,  
        nrPaths  : LENGTH(x.paths) }
```

```
[ { "begin": "germanCity/Cologne",  
  "end"   : { "_id": "frenchCity/Paris", ... },  
  "distance": 550,  
  "nrPaths": 10 },  
  ...  
]
```

Life of a query

1. Text and query parameters come from user

Life of a query

1. Text and query parameters come from user
2. Parse text, produce **abstract syntax tree (AST)**

Life of a query

1. Text and query parameters come from user
2. Parse text, produce **abstract syntax tree (AST)**
3. Substitute query parameters

Life of a query

1. Text and query parameters come from user
2. Parse text, produce **abstract syntax tree (AST)**
3. Substitute query parameters
4. First optimisation: constant expressions, etc.

Life of a query

1. Text and query parameters come from user
2. Parse text, produce **abstract syntax tree (AST)**
3. Substitute query parameters
4. First optimisation: constant expressions, etc.
5. Translate AST into an **execution plan (EXP)**

Life of a query

1. Text and query parameters come from user
2. Parse text, produce **abstract syntax tree (AST)**
3. Substitute query parameters
4. First optimisation: constant expressions, etc.
5. Translate AST into an **execution plan (EXP)**
6. Optimise one EXP, produce many, potentially better EXPs

Life of a query

1. Text and query parameters come from user
2. Parse text, produce **abstract syntax tree (AST)**
3. Substitute query parameters
4. First optimisation: constant expressions, etc.
5. Translate AST into an **execution plan (EXP)**
6. Optimise one EXP, produce many, potentially better EXPs
7. Reason about **distribution** in cluster

Life of a query

1. Text and query parameters come from user
2. Parse text, produce **abstract syntax tree (AST)**
3. Substitute query parameters
4. First optimisation: constant expressions, etc.
5. Translate AST into an **execution plan (EXP)**
6. Optimise one EXP, produce many, potentially better EXPs
7. Reason about distribution in cluster
8. Optimise distributed EXPs

Life of a query

1. Text and query parameters come from user
2. Parse text, produce **abstract syntax tree (AST)**
3. Substitute query parameters
4. First optimisation: constant expressions, etc.
5. Translate AST into an **execution plan (EXP)**
6. Optimise one EXP, produce many, potentially better EXPs
7. Reason about distribution in cluster
8. Optimise distributed EXPs
9. Estimate costs for all EXPs, and sort by ascending cost

Life of a query

1. Text and query parameters come from user
2. Parse text, produce **abstract syntax tree (AST)**
3. Substitute query parameters
4. First optimisation: constant expressions, etc.
5. Translate AST into an **execution plan (EXP)**
6. Optimise one EXP, produce many, potentially better EXPs
7. Reason about distribution in cluster
8. Optimise distributed EXPs
9. Estimate costs for all EXPs, and sort by ascending cost
10. Instantiate “cheapest” plan, i.e. set up execution engine

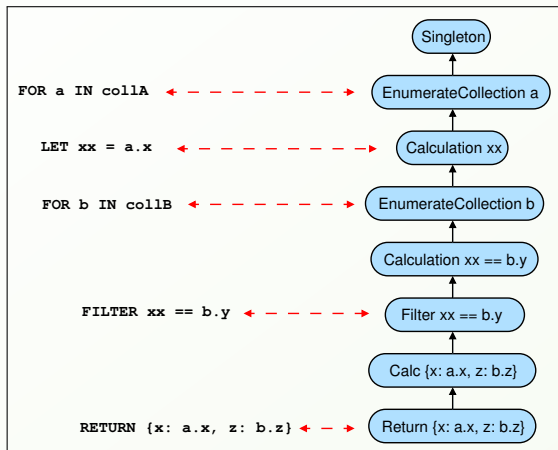
Life of a query

1. Text and query parameters come from user
2. Parse text, produce **abstract syntax tree (AST)**
3. Substitute query parameters
4. First optimisation: constant expressions, etc.
5. Translate AST into an **execution plan (EXP)**
6. Optimise one EXP, produce many, potentially better EXPs
7. Reason about **distribution** in cluster
8. Optimise **distributed** EXPs
9. Estimate costs for all EXPs, and sort by ascending cost
10. Instantiate “cheapest” plan, i.e. set up execution engine
11. Distribute and link up engines on different servers

Life of a query

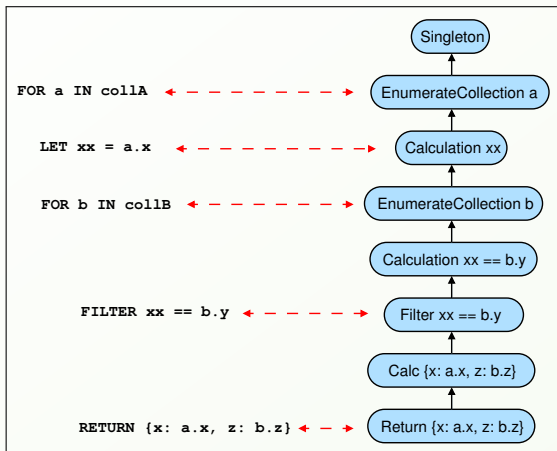
1. Text and query parameters come from user
2. Parse text, produce **abstract syntax tree (AST)**
3. Substitute query parameters
4. First optimisation: constant expressions, etc.
5. Translate AST into an **execution plan (EXP)**
6. Optimise one EXP, produce many, potentially better EXPs
7. Reason about **distribution** in cluster
8. Optimise **distributed** EXPs
9. Estimate costs for all EXPs, and sort by ascending cost
10. Instantiate “cheapest” plan, i.e. set up execution engine
11. Distribute and link up engines on different servers
12. Execute plan, provide **cursor API**

Execution plans



Query → EXP

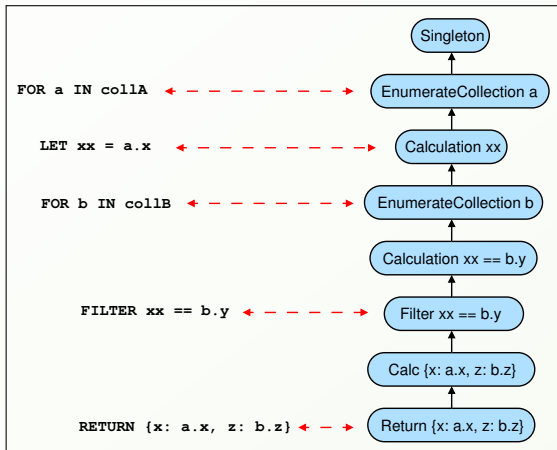
Execution plans



Query → EXP

Black arrows are dependencies

Execution plans

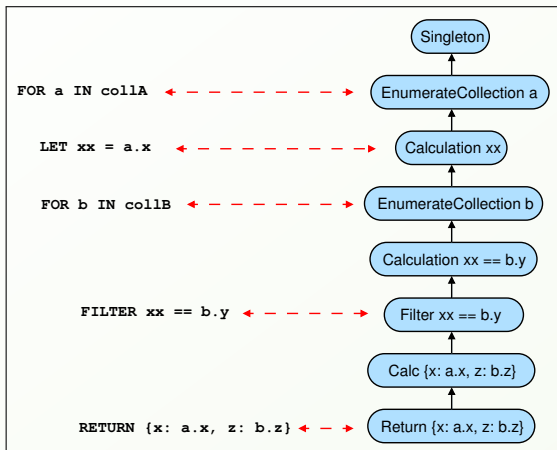


Query → EXP

Black arrows are dependencies

Think of a pipeline

Execution plans



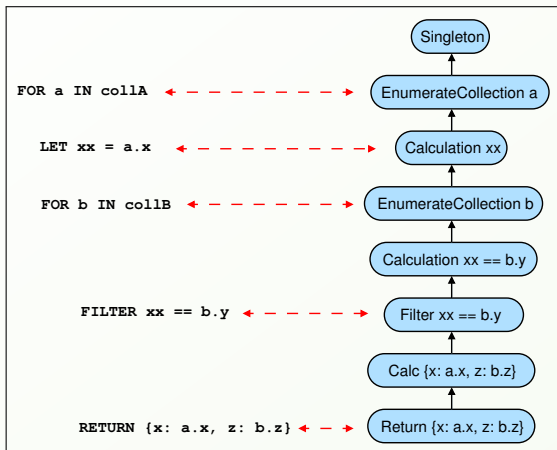
Query → EXP

Black arrows are dependencies

Think of a pipeline

Each node provides a cursor API

Execution plans



Query → EXP

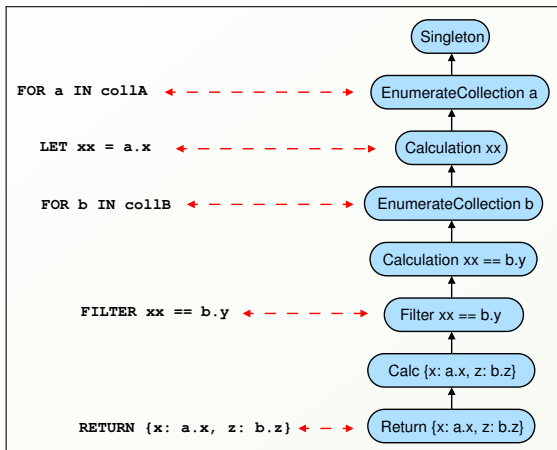
Black arrows are dependencies

Think of a pipeline

Each node provides a cursor API

Blocks of "Items" travel through the pipeline

Execution plans



Query → EXP

Black arrows are dependencies

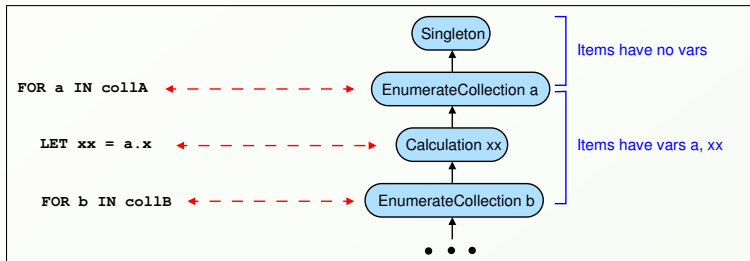
Think of a pipeline

Each node provides a cursor API

Blocks of "Items" travel through the pipeline

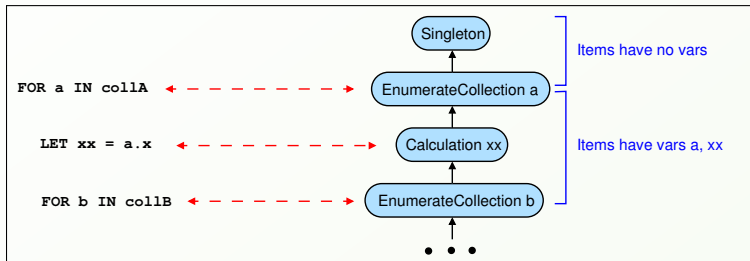
What is an "item"???

Pipeline and items



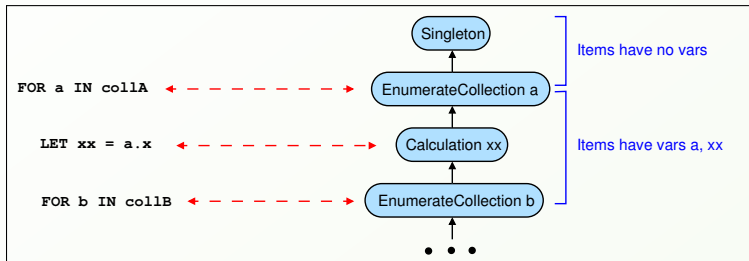
- **Items** are the **thingies** traveling through the pipeline.

Pipeline and items



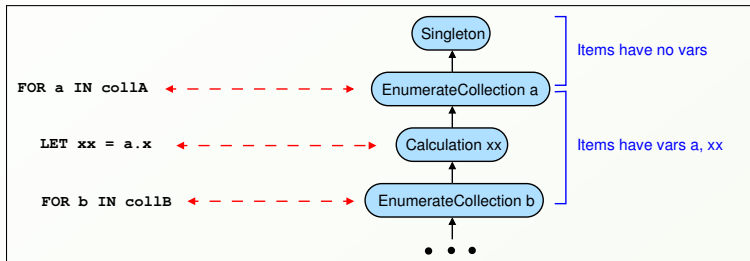
- ▶ **Items** are the **thingies** traveling through the pipeline.
- ▶ An **item** holds **values** of those variables in the current frame

Pipeline and items



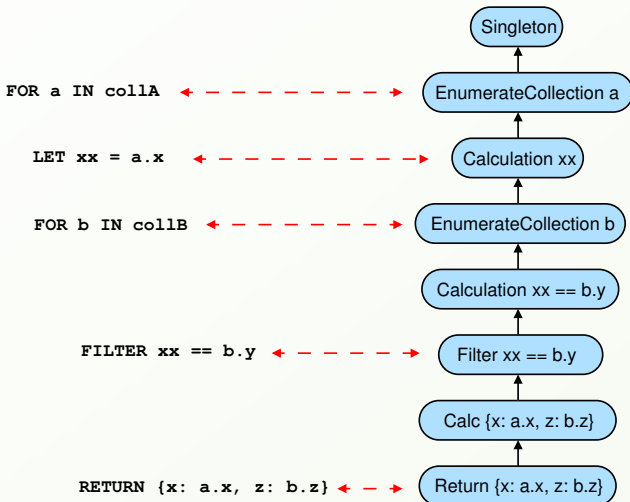
- ▶ **Items** are the **thingies** traveling through the pipeline.
- ▶ An **item** holds **values** of those **variables** in the current frame
- ▶ **Thus:** Items **look differently** in different parts of the plan

Pipeline and items



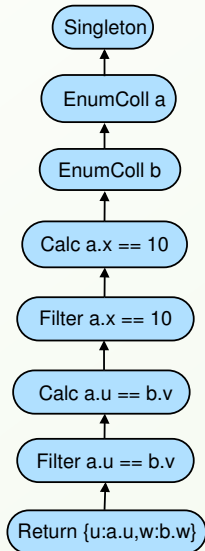
- ▶ **Items** are the **thingies** traveling through the pipeline.
- ▶ An **item** holds **values** of those **variables** in the current frame
- ▶ **Thus:** Items **look differently** in different parts of the plan
- ▶ We always deal with **blocks of items** for performance reasons

Execution plans



Move filters up

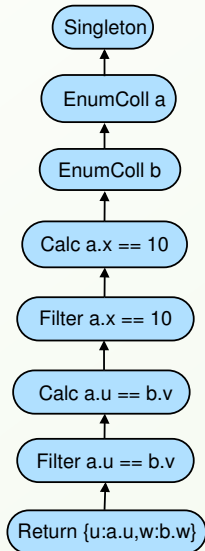
```
FOR a IN collA
  FOR b IN collB
    FILTER a.x == 10
    FILTER a.u == b.v
    RETURN {u:a.u,w:b.w}
```



Move filters up

```
FOR a IN collA
  FOR b IN collB
    FILTER a.x == 10
    FILTER a.u == b.v
    RETURN {u:a.u,w:b.w}
```

The **result** and **behaviour** does **not** **change**, if the first FILTER is **pulled** out of the inner FOR.

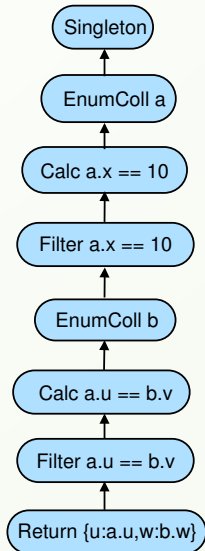


Move filters up

```
FOR a IN collA
  FILTER a.x < 10
  FOR b IN collB
    FILTER a.u == b.v
  RETURN {u:a.u,w:b.w}
```

The **result** and **behaviour** does **not change**, if the first FILTER is **pulled** out of the inner FOR.

However, the **number of items traveling in the pipeline** is decreased.



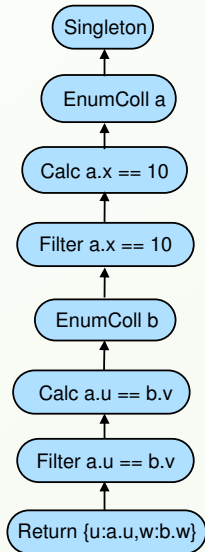
Move filters up

```
FOR a IN collA
  FILTER a.x < 10
  FOR b IN collB
    FILTER a.u == b.v
    RETURN {u:a.u,w:b.w}
```

The **result** and **behaviour** does **not change**, if the first FILTER is **pulled** out of the inner FOR.

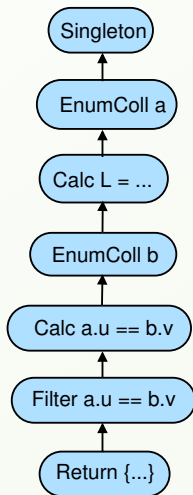
However, the **number of items traveling in the pipeline is decreased**.

Note that the two FOR statements could be interchanged!



Remove unnecessary calculations

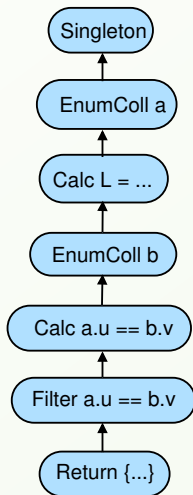
```
FOR a IN collA
  LET L = LENGTH(a.hobbies)
  FOR b IN collB
    FILTER a.u == b.v
    RETURN {h:a.hobbies,w:b.w}
```



Remove unnecessary calculations

```
FOR a IN collA
  LET L = LENGTH(a.hobbies)
  FOR b IN collB
    FILTER a.u == b.v
    RETURN {h:a.hobbies,w:b.w}
```

The **Calculation** of L is unnecessary!



Remove unnecessary calculations

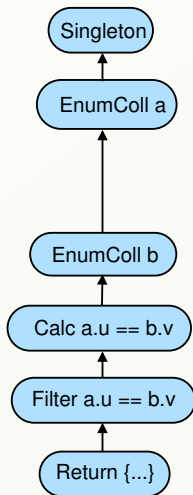
```
FOR a IN collA
```

```
  FOR b IN collB
```

```
    FILTER a.u == b.v
```

```
    RETURN {h:a.hobbies,w:b.w}
```

The **Calculation** of **L** is unnecessary!
(since it cannot throw an exception).



Remove unnecessary calculations

```
FOR a IN collA
```

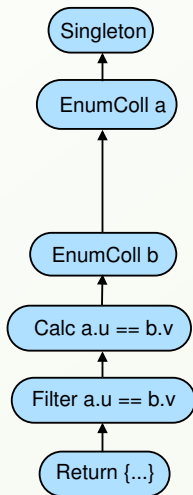
```
  FOR b IN collB
```

```
    FILTER a.u == b.v
```

```
    RETURN {h:a.hobbies,w:b.w}
```

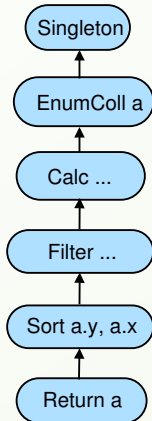
The **Calculation** of **L** is unnecessary!
(since it cannot throw an exception).

Therefore we can just leave it out.



Use index for FILTER and SORT

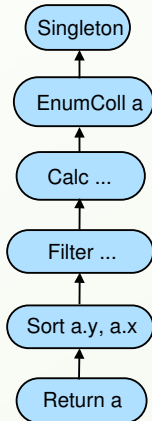
```
FOR a IN collA
  FILTER a.x > 17 &&
    a.x <= 23 &&
    a.y == 10
  SORT a.y, a.x
RETURN a
```



Use index for FILTER and SORT

```
FOR a IN collA
  FILTER a.x > 17 &&
    a.x <= 23 &&
    a.y == 10
  SORT a.y, a.x
  RETURN a
```

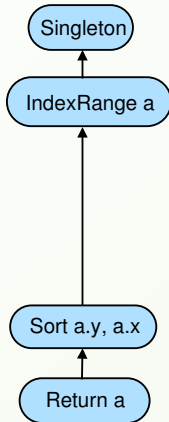
Assume **collA** has a **skiplist index** on "y"
and "x" (in this order),



Use index for FILTER and SORT

```
FOR a IN collA
  FILTER a.x > 17 &&
    a.x <= 23 &&
    a.y == 10
  SORT a.y, a.x
  RETURN a
```

Assume **collA** has a **skiplist index** on "y" and "x" (in this order), then we can **read off** the half-open **interval** between
 { y: 10, x: 17 } and
 { y: 10, x: 23 }
from the **skiplist index**.



Use index for FILTER and SORT

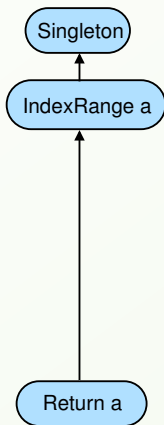
```
FOR a IN collA
  FILTER a.x > 17 &&
    a.x <= 23 &&
    a.y == 10
  SORT a.y, a.x
  RETURN a
```

Assume **collA** has a **skiplist index** on “y” and “x” (in this order), then we can **read off** the half-open **interval** between

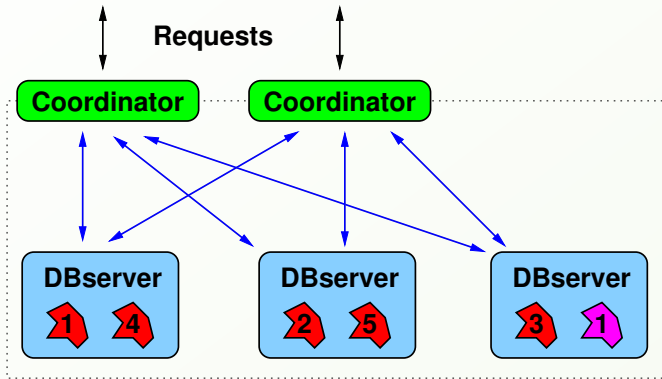
{ y: 10, x: 17 } and
{ y: 10, x: 23 }

from the **skiplist index**.

The result will **automatically be sorted** by y and then by x.

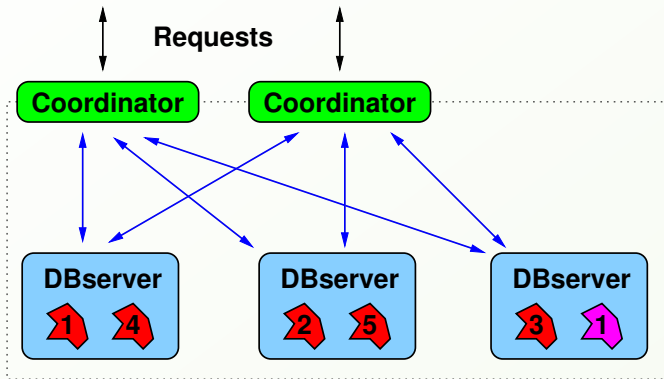


Data distribution in a cluster



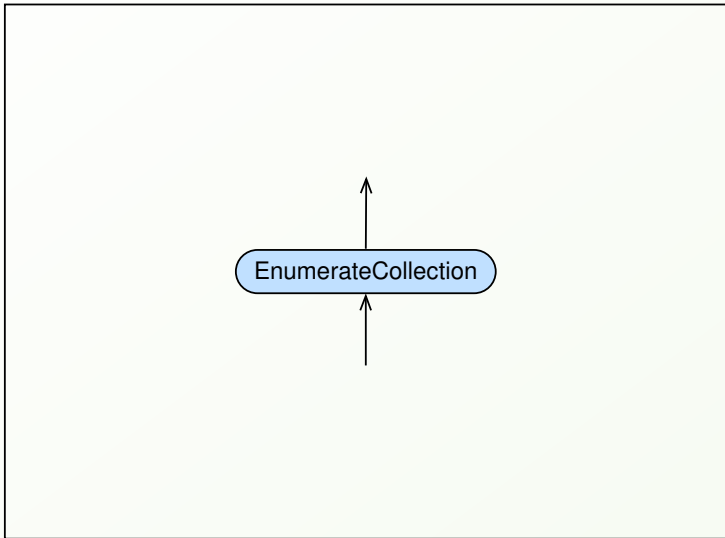
- ▶ The **shards** of a collection are distributed across the DB servers.

Data distribution in a cluster

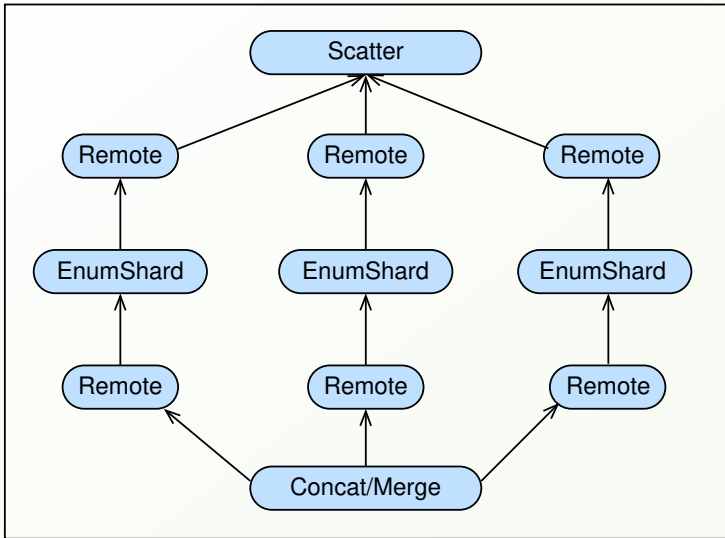


- ▶ The **shards** of a collection are distributed across the DB servers.
- ▶ The **coordinators** receive queries and organise their execution

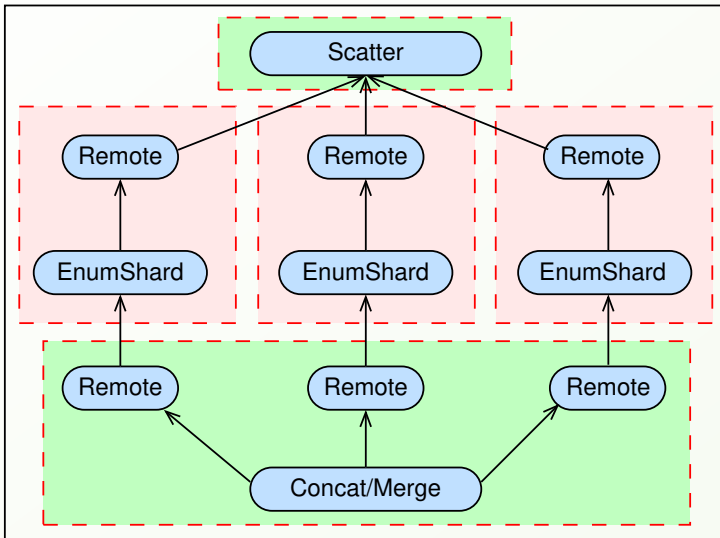
Scatter/gather



Scatter/gather



Scatter/gather



Modifying queries

Fortunately:

- ▶ There can be **at most one modifying node** in each query.
- ▶ There can be **no modifying nodes** in subqueries.

Modifying queries

Fortunately:

- ▶ There can be **at most one modifying node** in each query.
- ▶ There can be **no modifying nodes** in subqueries.

Modifying nodes

The modifying node in a query

- ▶ is **executed on the** DBservers,

Modifying queries

Fortunately:

- ▶ There can be **at most one modifying node** in each query.
- ▶ There can be **no modifying nodes** in subqueries.

Modifying nodes

The modifying node in a query

- ▶ is **executed on the** DBservers,
- ▶ to this end, we **either scatter** the items **to all DBservers**, or, if possible, we **distribute** each item to the shard that is responsible for the modification.

Modifying queries

Fortunately:

- ▶ There can be **at most one modifying node** in each query.
- ▶ There can be **no modifying nodes** in subqueries.

Modifying nodes

The modifying node in a query

- ▶ is **executed on the** DBservers,
- ▶ to this end, we **either scatter** the items **to all DBservers**, or, if possible, we **distribute** each item to the shard that is responsible for the modification.
- ▶ Sometimes, we can even **optimise away** a gather/scatter combination and **parallelise completely**.