

# Data-Oriented Django Drei

Adam Johnson





5Gbps burst network capacity

```
<!doctype html>
```



```
<!doctype html>
```



1MiB would be big

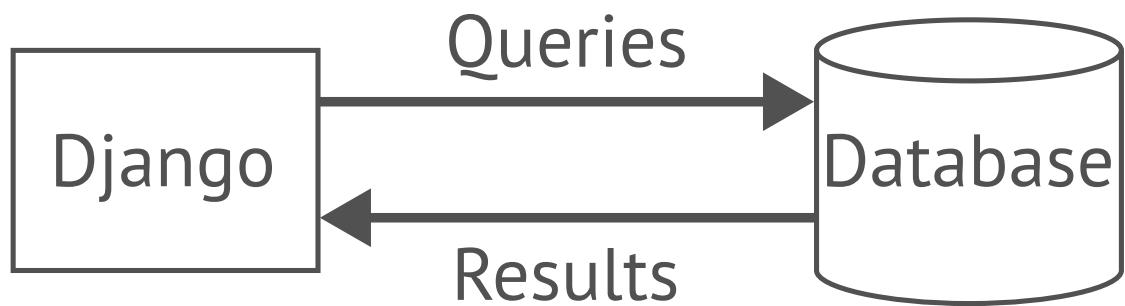
1MiB / 5Gbps =

**1MiB / 5Gbps = 1.7ms**

$$1\text{MiB} / 5\text{Gbps} = 1.7\text{ms}$$

So why is 100ms a “great” server response time?

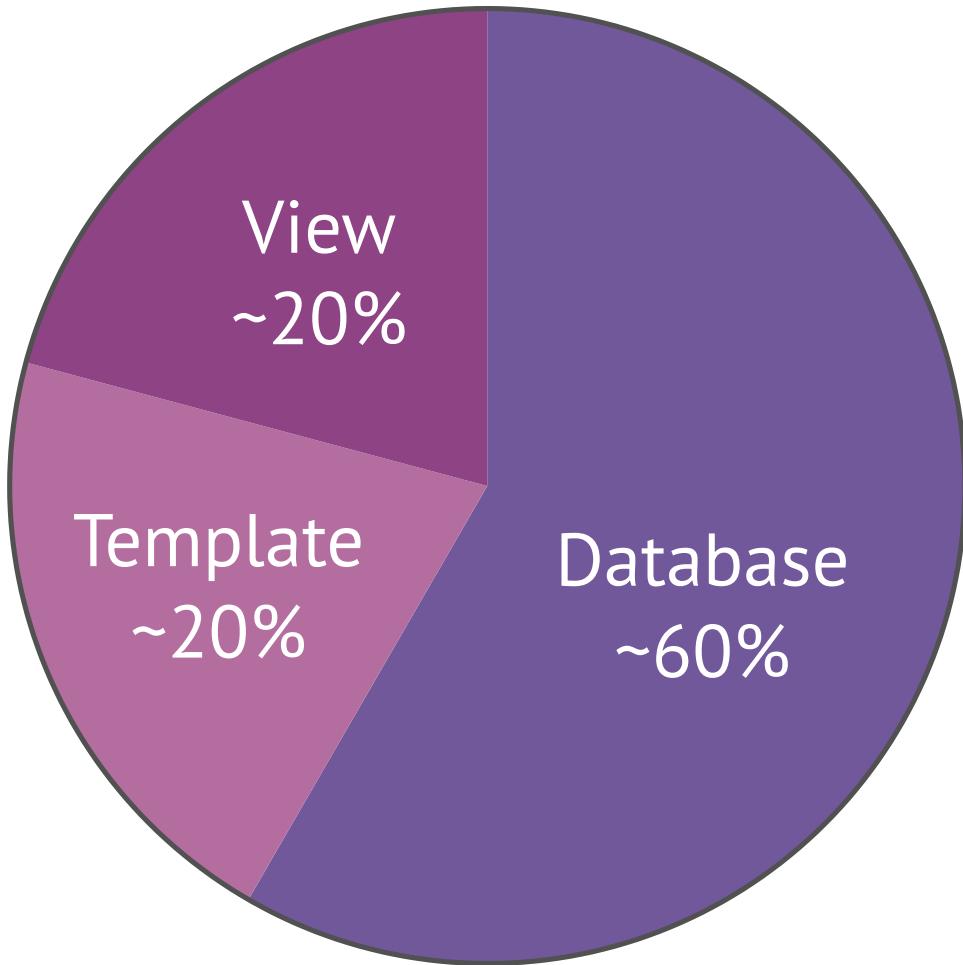


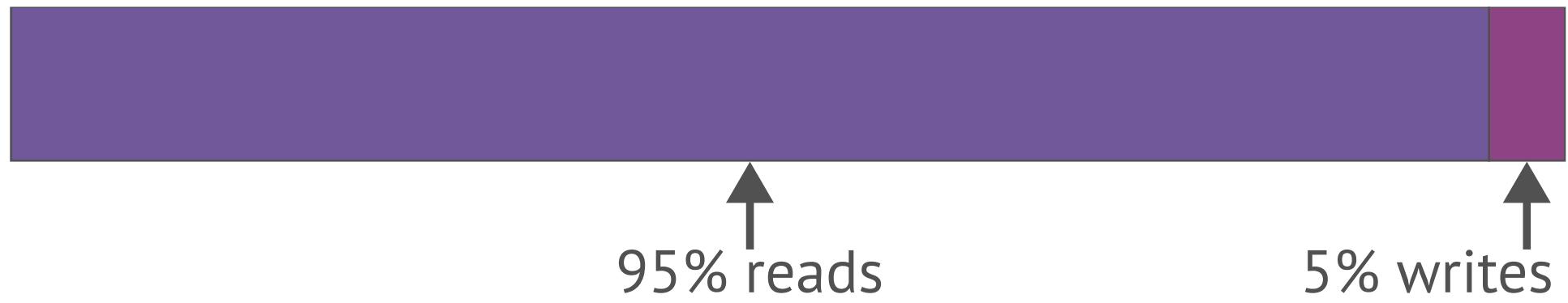


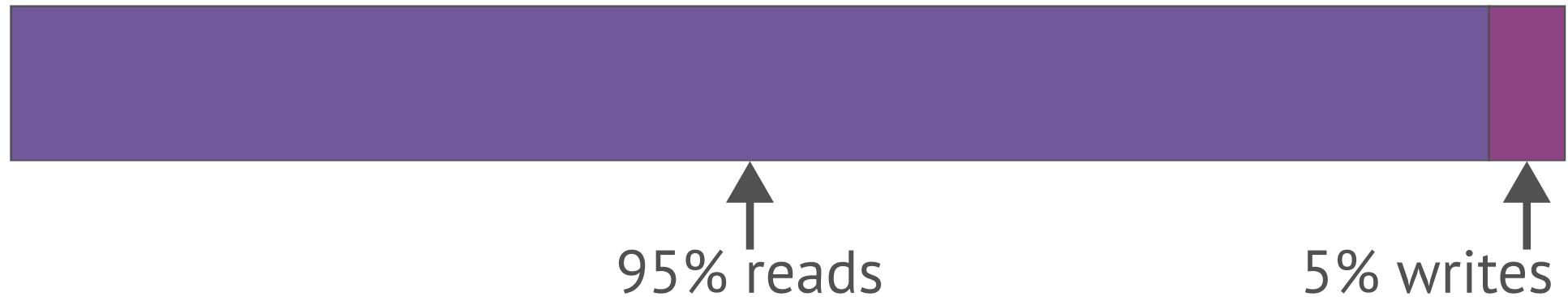
# Measure

Profile production with an APM tool:

- **Sentry**
- **Scout APM**
- **django-debug-toolbar**







We gotta optimize the reads!

# Model

```
from django.db import models

class Engine(models.Model):
    name = models.TextField()
    colour = models.TextField()
```

# Table

<b>id</b>	<b>name</b>	<b>colour</b>
1	Thomas	blue
6	Percy	green
78	Kana	purple
...		

# Table

ctid	id	name	colour
75	1	Thomas	blue
11	6	Percy	green
20	78	Kana	purple
...			

```
Engine.objects.filter(name='Kana' )
```

```
Engine.objects.filter(name='Kana')
```

```
SELECT * FROM example_engine  
WHERE name = 'Kana'
```

```
SELECT * FROM example_engine  
WHERE name = 'Kana'
```

ctid	id	name	colour	matched?
75	1	Thomas	blue	
11	6	Percy	green	
20	78	Kana	purple	
...				

```
SELECT * FROM example_engine  
WHERE name = 'Kana'
```

ctid	id	name	colour	matched?
75	1	Thomas	blue	✗
11	6	Percy	green	
20	78	Kana	purple	
...				

```
SELECT * FROM example_engine  
WHERE name = 'Kana'
```

ctid	id	name	colour	matched?
75	1	Thomas	blue	✗
11	6	Percy	green	✗
20	78	Kana	purple	
...				

```
SELECT * FROM example_engine  
WHERE name = 'Kana'
```

ctid	id	name	colour	matched?
75	1	Thomas	blue	✗
11	6	Percy	green	✗
20	78	Kana	purple	✓

...

# Sequential scan

# Sequential scan

$O(n)$

# Sequential scan

$O(n)$

Fast for small  $n$

# Sequential scan

$O(n)$

Fast for small  $n$

Slow for medium  $n$

# $O(n)$

# rows	~operations
1	1
10	10
100	100
1,000	1,000
10,000	10,000
100,000	100,000
1,000,000	1,000,000

**Tables are a leaky abstraction**

# Tables are a leaky abstraction

**Promise:** describe your data, DB figures out storage

# Tables are a leaky abstraction

**Promise:** describe your data, DB figures out storage

**Reality:** horrendous performance without indexes

# **Tables are a leaky abstraction**

**Promise:** describe your data, DB figures out storage

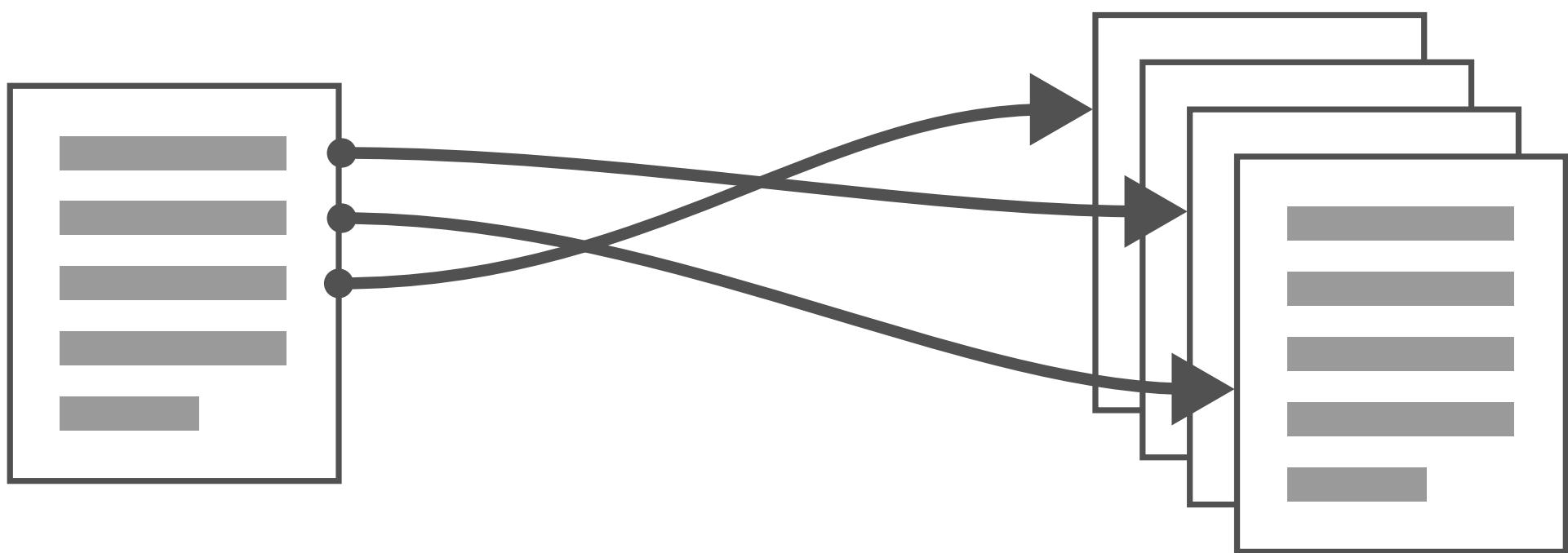
**Reality:** horrendous performance without indexes

**Indexes make the world go round**

# Indexes

Index

Content



# Indexes

Like a `dict` of values to row IDs

```
index_on_name = {  
    "Kana": [20],  
    "Percy": [11],  
    "Thomas": [75],  
}
```

# Add an index

```
from django.db import models

class Engine(models.Model):
    name = models.TextField()
    colour = models.TextField()
```

# Add an index

```
from django.db import models

class Engine(models.Model):
    name = models.TextField()
    colour = models.TextField()

    class Meta:
        indexes = [
            models.Index("name"),
        ]
```

# Migration

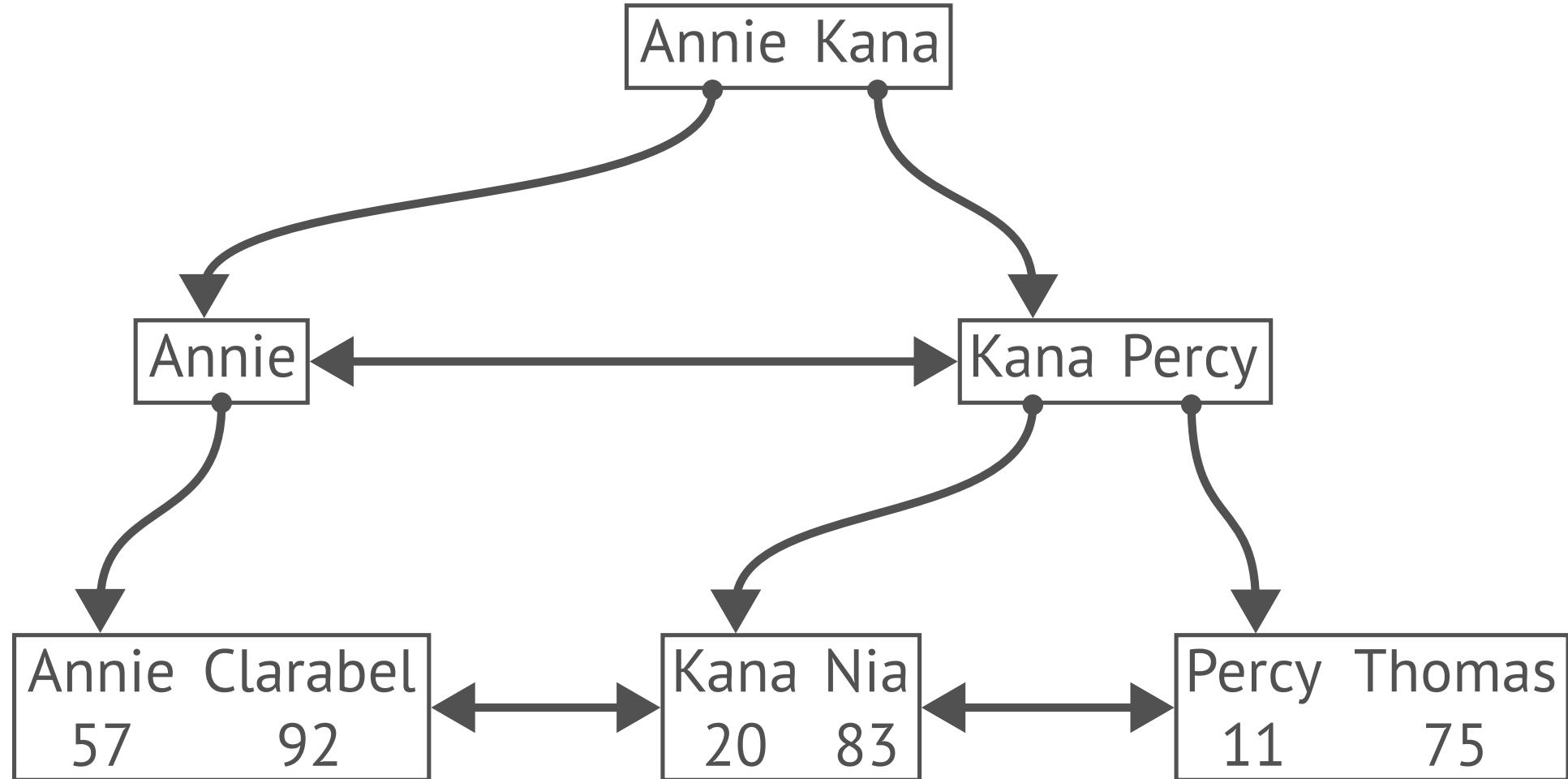
```
migrations.AddIndex(  
    model_name="engine",  
    index=models.Index(  
        "name",  
        name="example_engine_name",  
    ),  
)
```

# Migration

```
migrations.AddIndex(  
    model_name="engine",  
    index=models.Index(  
        "name",  
        name="example_engine_name",  
    ),  
)
```

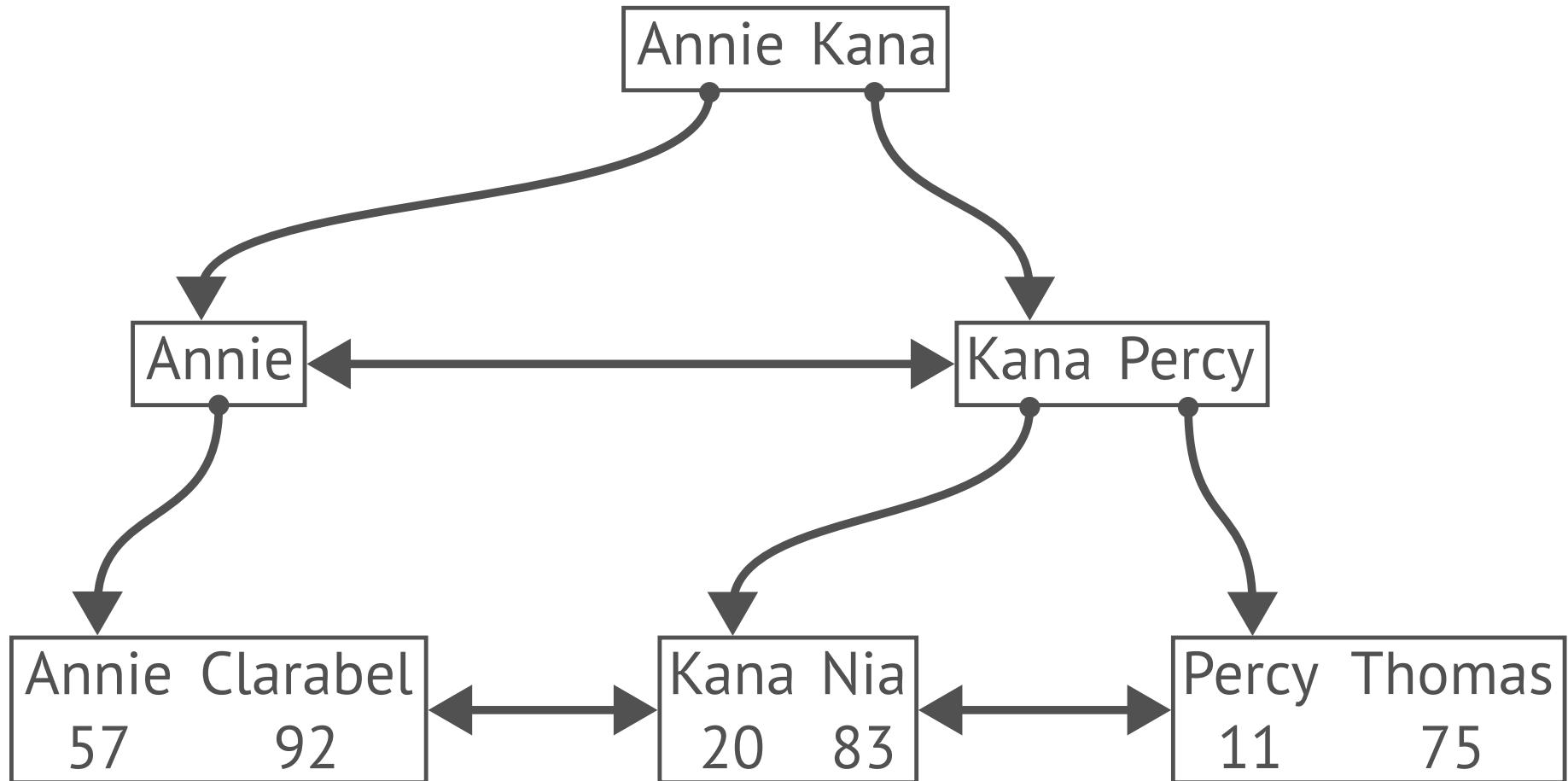
```
CREATE INDEX example_engine_name  
ON example_engine (name);
```

# B+ tree index

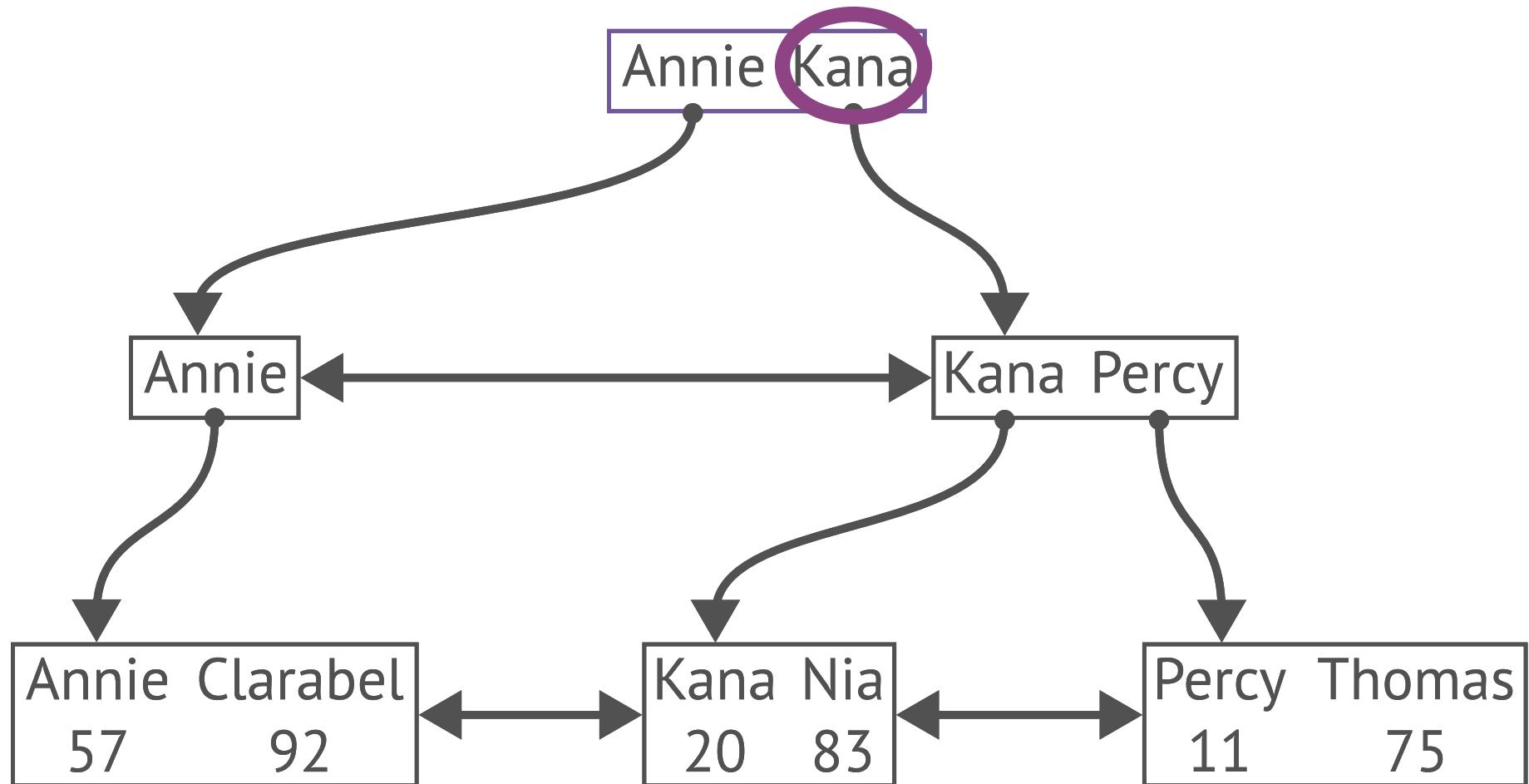


```
SELECT * FROM example_engine  
WHERE name = 'Kana'
```

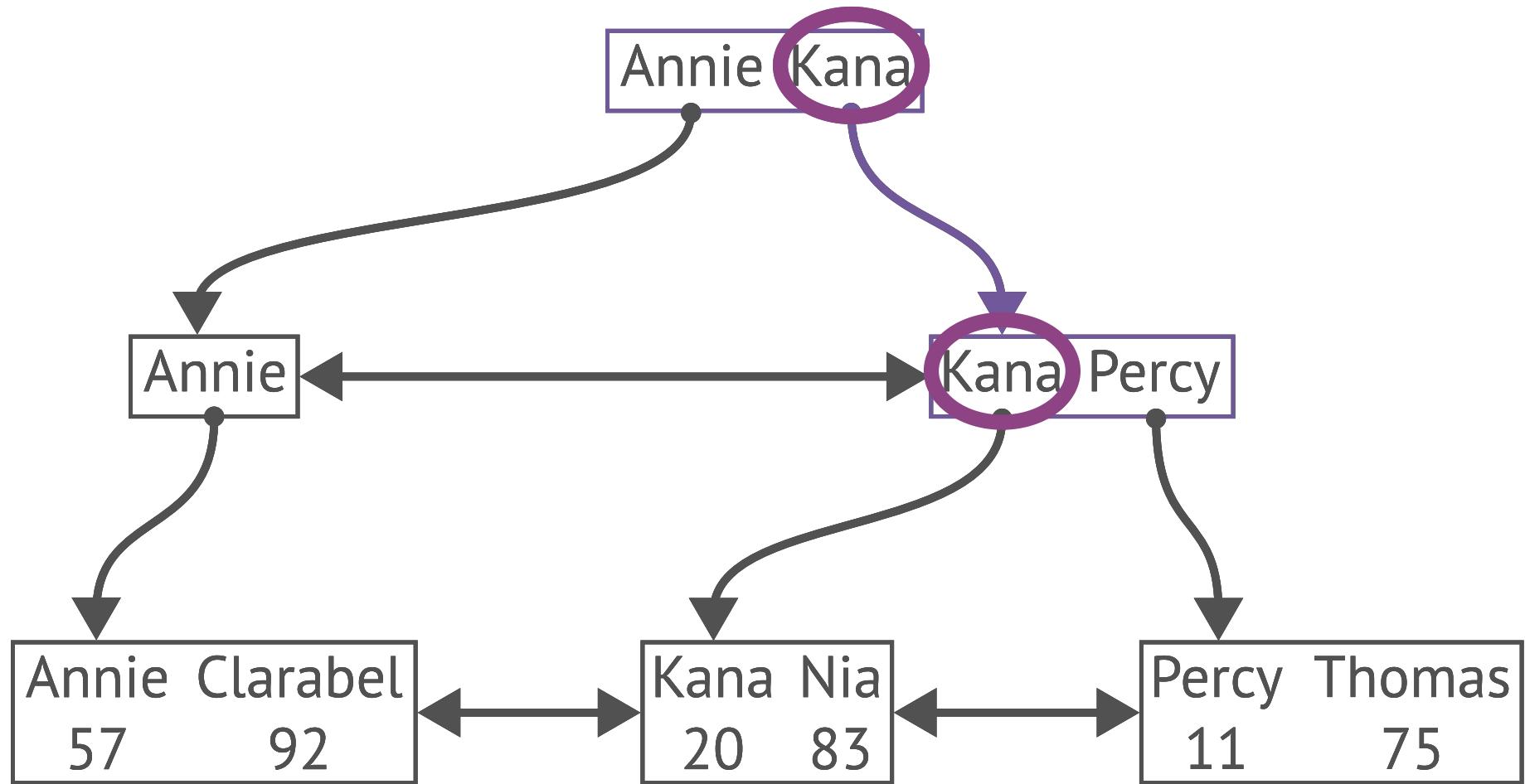
```
SELECT * FROM example_engine  
WHERE name = 'Kana'
```



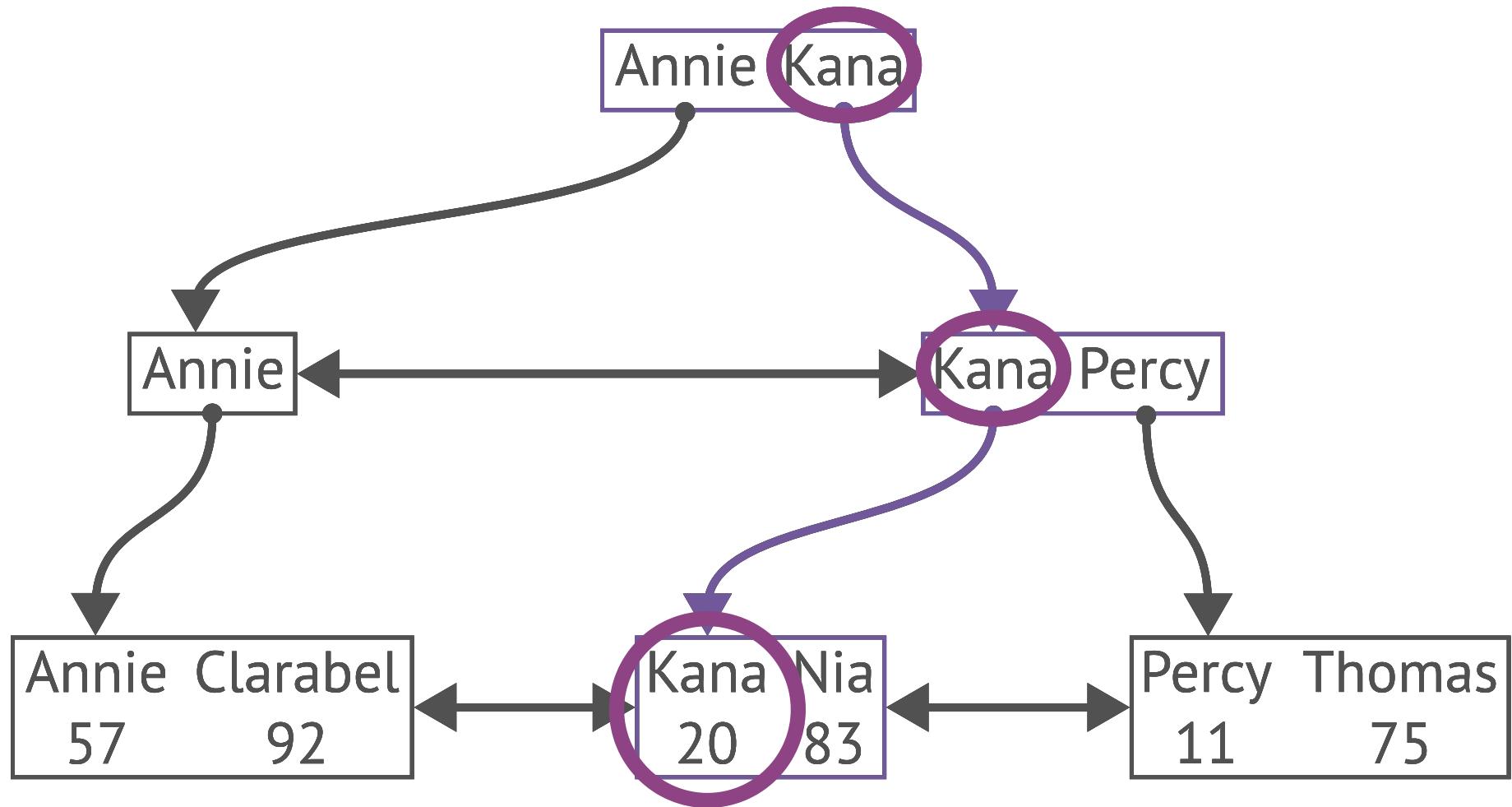
```
SELECT * FROM example_engine  
WHERE name = 'Kana'
```



```
SELECT * FROM example_engine  
WHERE name = 'Kana'
```



```
SELECT * FROM example_engine  
WHERE name = 'Kana'
```



# B+ tree search

# B+ tree search

$O(\log n)$

# B+ tree search

$O(\log n)$

Fast for even large  $n$

# $O(\log n)$

# rows	~operations
1	1
10	2.3
100	4.6
1,000	6.9
10,000	9.2
100,000	11.5
1,000,000	13.8

# Range scans

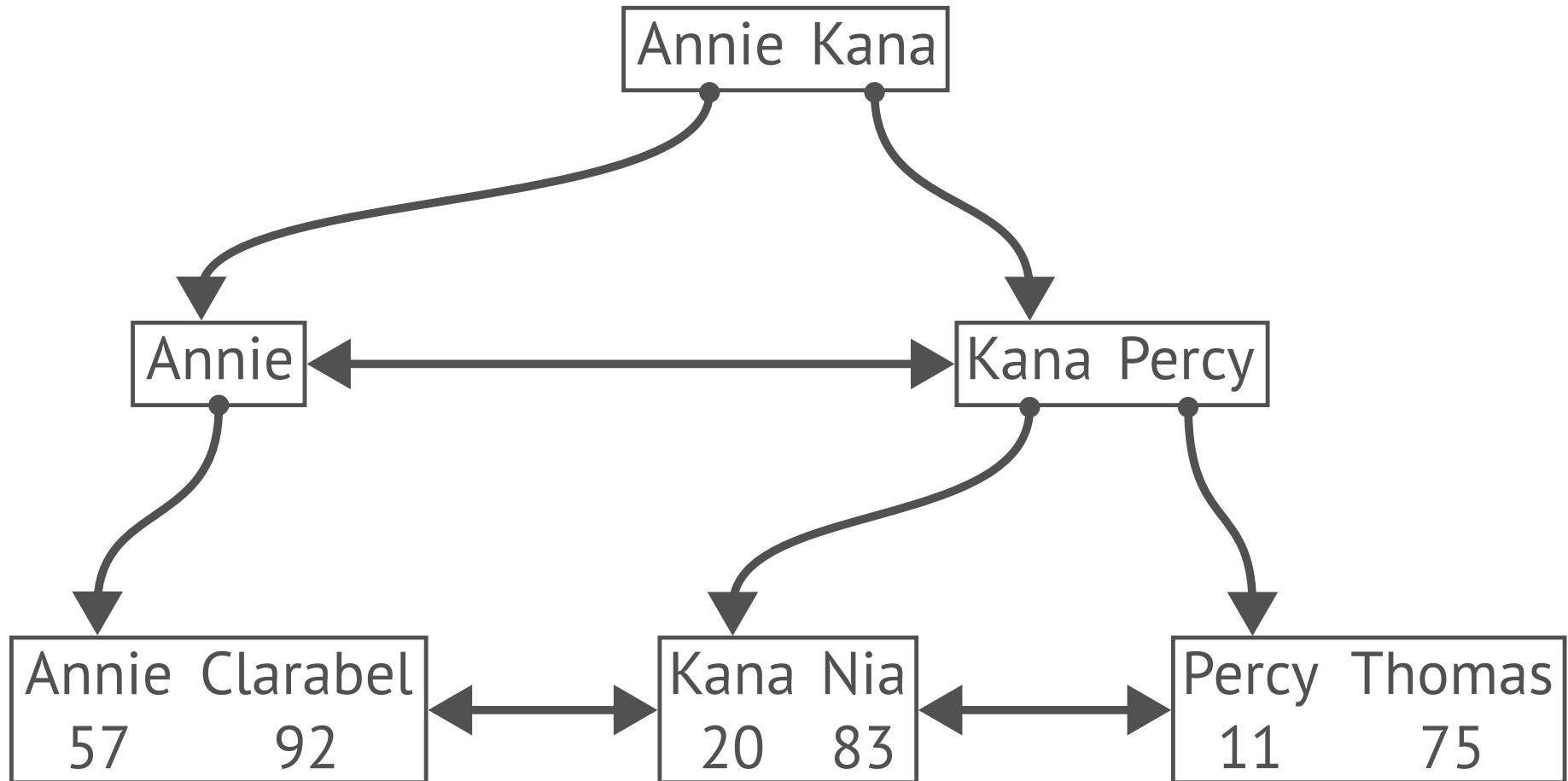
```
Engine.objects.filter(name__gte="Kana")
```

# Range scans

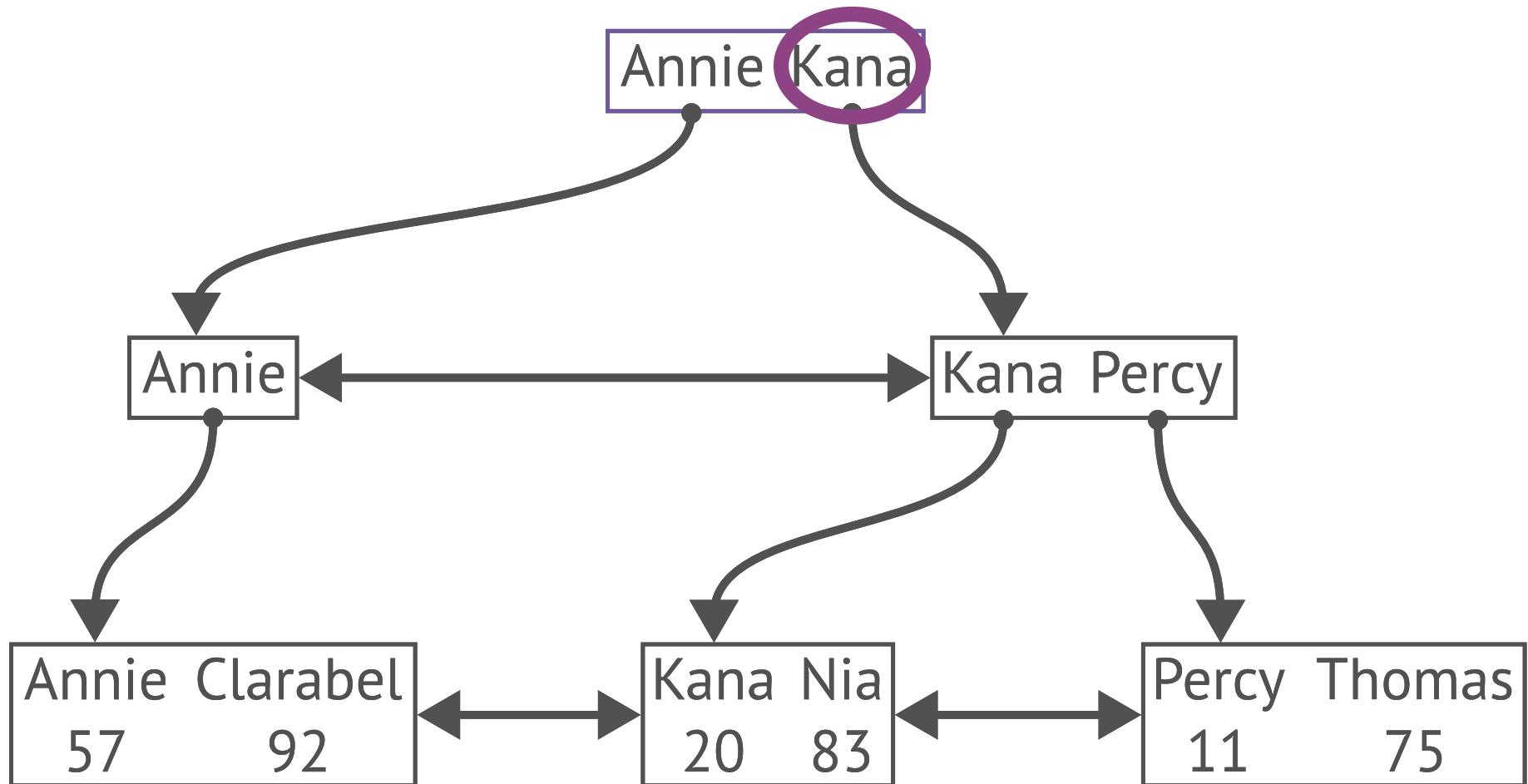
```
Engine.objects.filter(name__gte="Kana")
```

```
SELECT * FROM example_engine  
WHERE name >= 'Kana'
```

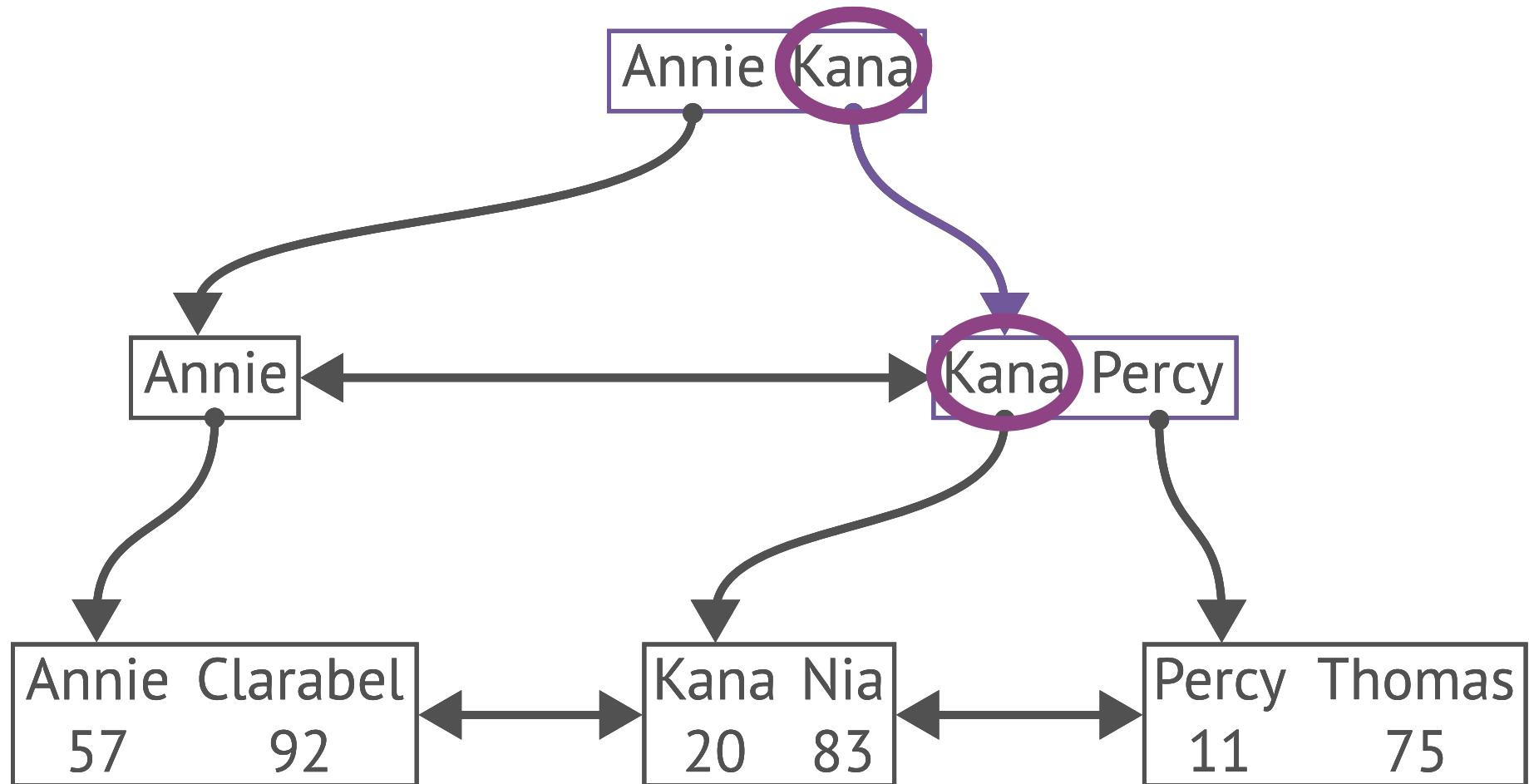
```
SELECT * FROM example_engine  
WHERE name >= 'Kana'
```



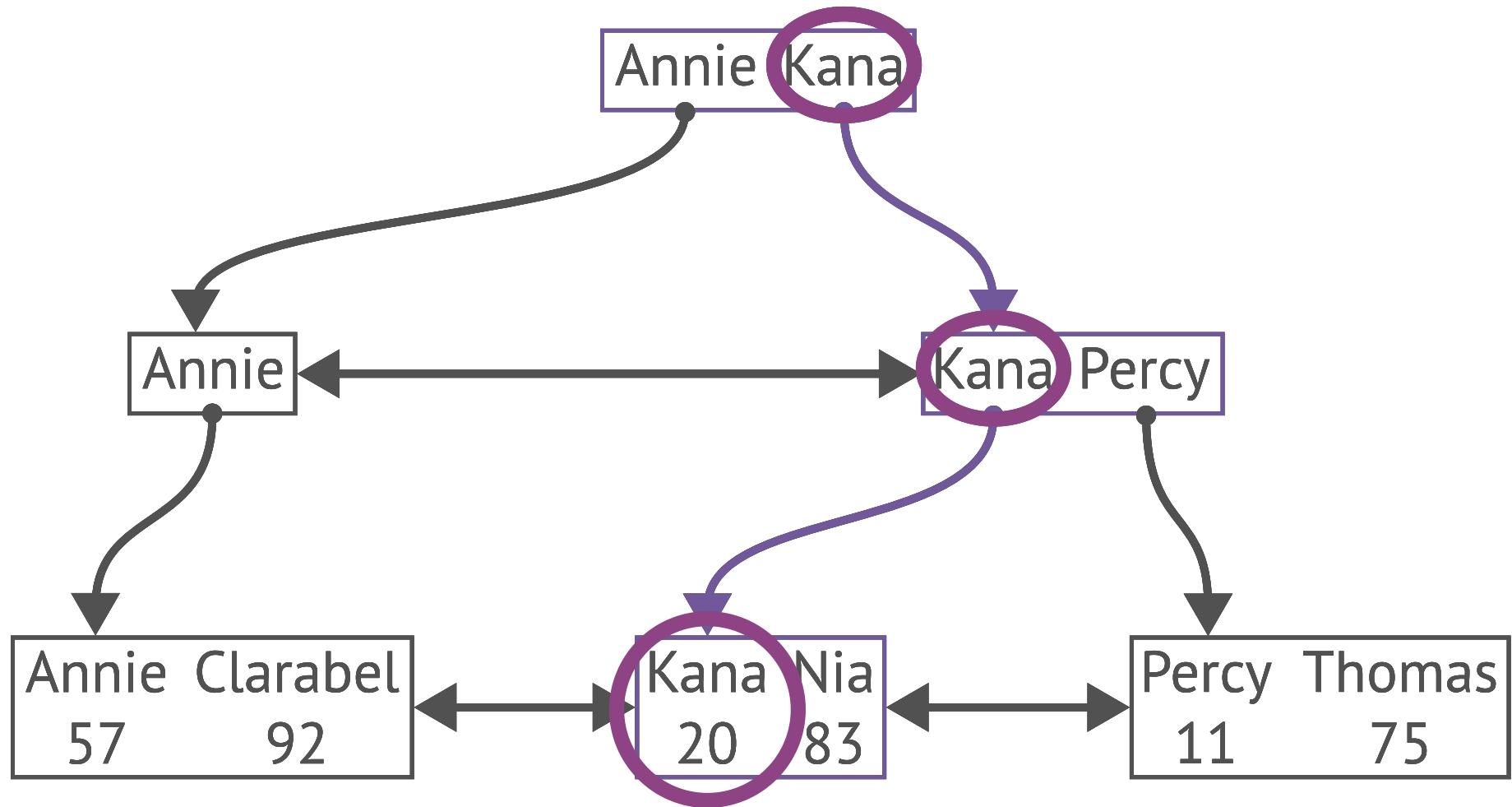
```
SELECT * FROM example_engine  
WHERE name >= 'Kana'
```



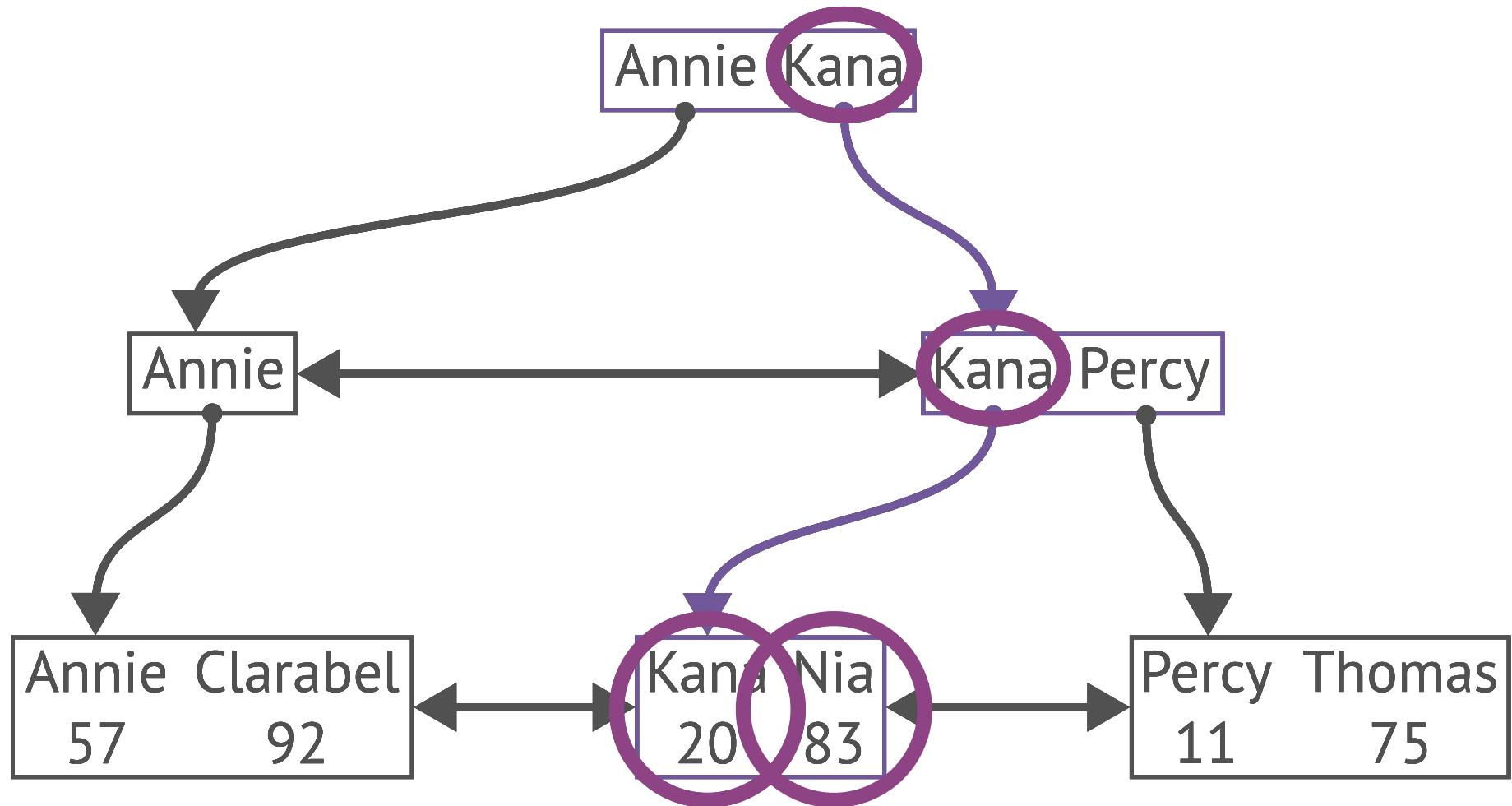
```
SELECT * FROM example_engine  
WHERE name >= 'Kana'
```



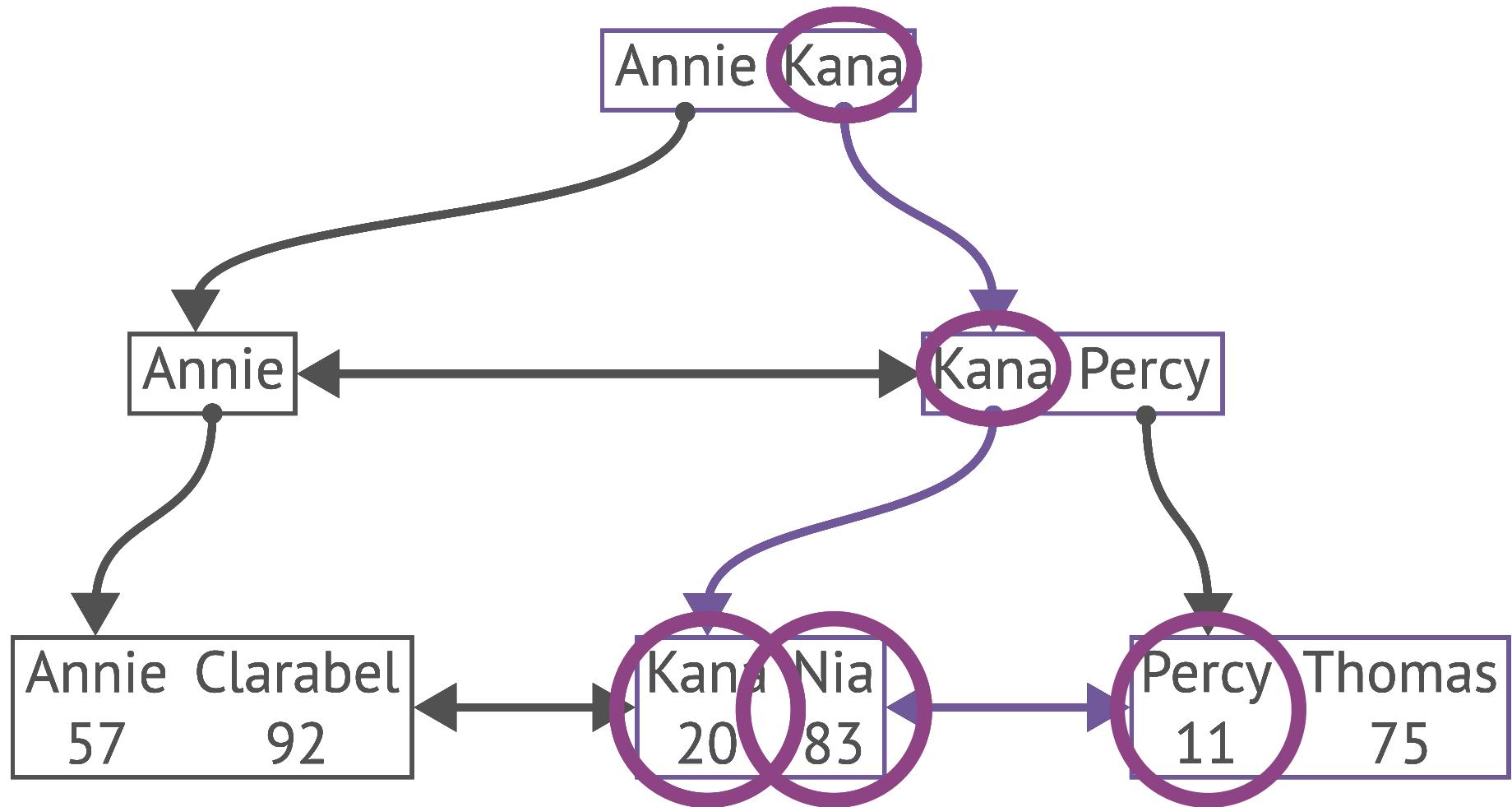
```
SELECT * FROM example_engine  
WHERE name >= 'Kana'
```



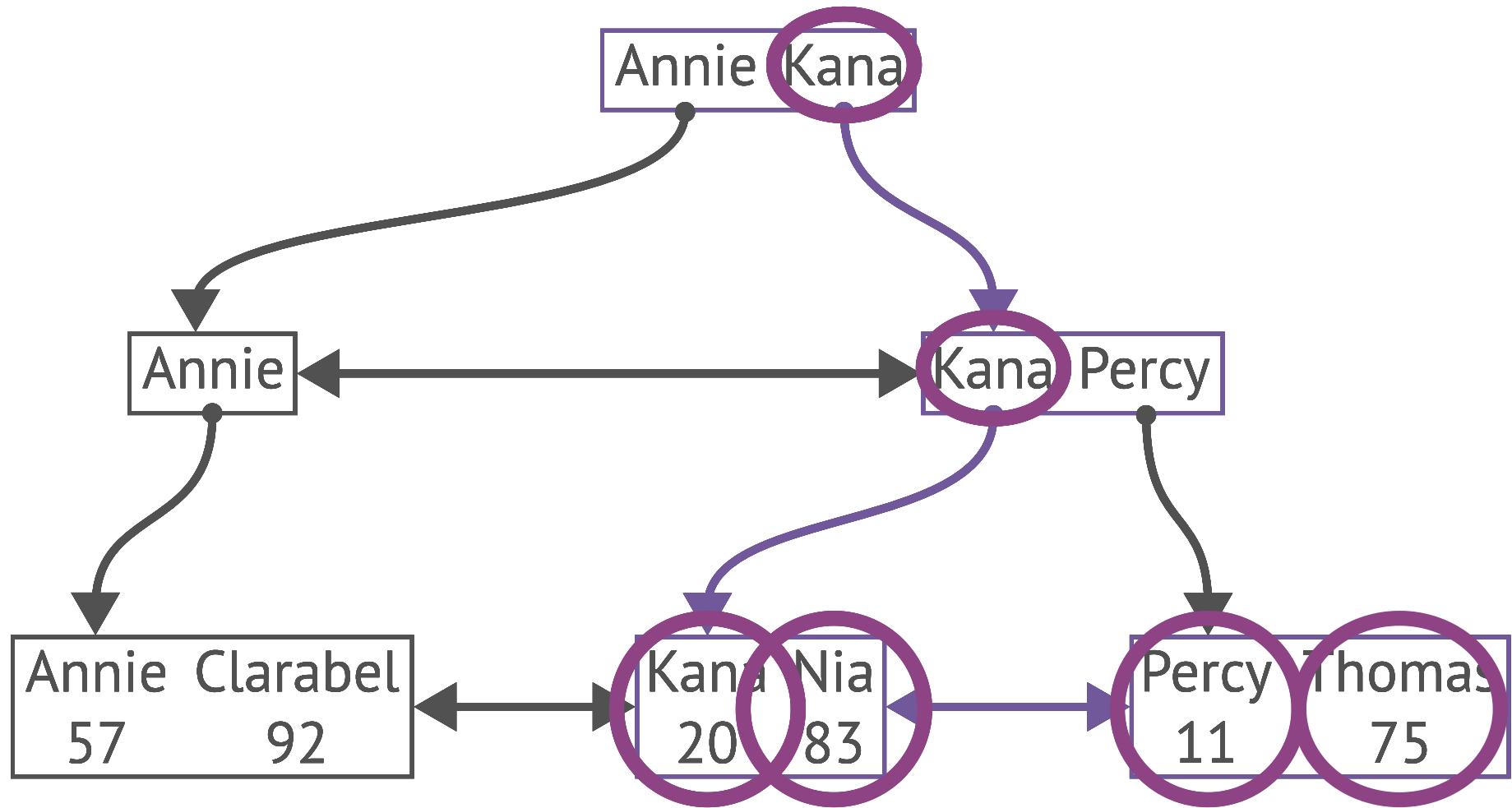
```
SELECT * FROM example_engine  
WHERE name >= 'Kana'
```



```
SELECT * FROM example_engine  
WHERE name >= 'Kana'
```



```
SELECT * FROM example_engine  
WHERE name >= 'Kana'
```



# Multi-column indexes

```
from django.db import models  
  
models.Index("name", "colour")
```

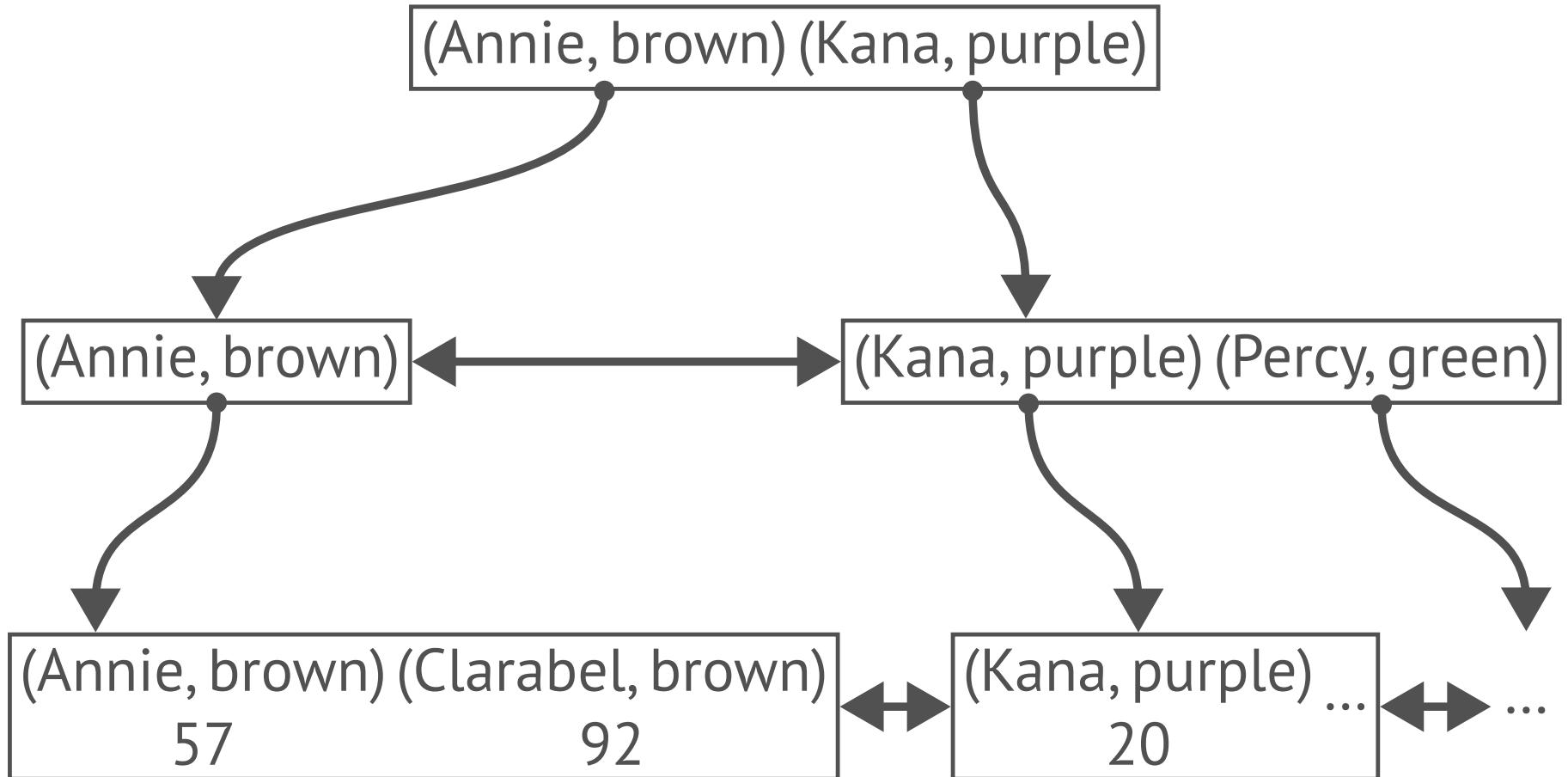
# Multi-column indexes

```
from django.db import models  
  
models.Index("name", "colour")
```

Like a `dict` of value tuples to row IDs

```
index_on_name_and_colour = {  
    ("Kana", "purple"): [20],  
    ("Percy", "green"): [11],  
    ("Thomas", "blue"): [75],  
}
```

```
from django.db import models  
  
models.Index("name", "colour")
```



# Expression indexes

```
from django.db import models
from django.db.models import Lower

models.Index(Lower('name'))
```

# Expression indexes

```
from django.db import models
from django.db.models import Lower

models.Index(Lower("name"))
```

Speeds up filtering like:

```
Engine.objects.annotate(
    name_lower=Lower("name")
).filter(
    name_lower="kana"
)
```

# Partial indexes

```
from django.db import models

models.Index(
    "name",
    condition=models.Q(colour="purple"),
)
```

# Partial indexes

```
from django.db import models

models.Index(
    "name",
    condition=models.Q(colour="purple"),
)
```

Speeds up only queries with **condition**:

```
Engine.objects.filter(colour="purple", name="Kana")
```

# Inclusion indexes

```
from django.db import models

models.Index(
    "name",
    include=["colour"]
)
```

# Inclusion indexes

```
from django.db import models

models.Index(
    "name",
    include=["colour"]
)
```

Speeds up queries that only use indexed and included columns:

```
Engine.objects.filter(
    name="Kana",
).only("name", "colour")
```

# Alternative index data structures

Typically  $O(\log n)$  but smaller

# Alternative index data structures

Typically  $O(\log n)$  but smaller

- **GiST** - spatial data, full text search
- **GIN** - full text search, JSON
- **Hash** - dict-like
- **Bloom** - bloom filters
- **HNSW / IVFFlat** - embedding vectors (pgvector)

# Alternative index data structures

Typically  $O(\log n)$  but smaller

- **GiST** - spatial data, full text search
- **GIN** - full text search, JSON
- **Hash** - dict-like
- **Bloom** - bloom filters
- **HNSW / IVFFlat** - embedding vectors (pgvector)

See `django.contrib.postgres.indexes`

# Options combinable

Partial inclusion multi-column expression bloom  
index, anyone?

# Default indexes

- Primary key
- Foreign keys
- Unique constraints

# Replace a default index

```
from django.db import models

class Engine(models.Model):
    ...
    home = models.ForeignKey(..., db_index=False)

    class Meta:
        indexes = [
            models.Index("home"),
        ]
```

# Indexes are not free

Extra storage

Overhead on writes

So we cannot “index all the things”

# Which indexes to add?

# Which indexes to add?

Whole system optimization problem

# **Which indexes to add?**

Whole system optimization problem

**More of an art than a science**

# Two approaches

1. **Design** indexes with model and queries
2. **Debug** slow/resource-consuming queries

# Design

“I know we will be filtering by **name** a lot, so let’s index it”

# Debug

“These are the slowest queries, let’s see if any indexes could help them”

**pgMustard**

# pgMustard

Get the **query plan** with `QuerySet.explain()`:

```
Engine.objects.filter(name="kana").explain(  
    format="json",  
    analyze=True,  
    buffers=True,  
    verbose=True,  
    settings=True,  
    wal=True,  
)
```

# pgMustard

Query plan:

```
[{"Plan": {"Node Type": "Gather", "Parallel Aware":  
false, "Startup Cost": 1000.0, "Total Cost":  
191384.91, "Plan Rows": 1031, "Plan Width": 72,  
"Actual Startup Time": 36.64, "Actual Total Time":  
4307.309, "Actual Rows": 100001, "Actual Loops": 1,  
...}
```

# pgMustard

pgMustard

New plan History Published plans

Docs Issues Changelog Account Sign out

↑ Publish

## Summary

Total time: 4,358ms

Sum of buffers: 1 GB

## Top tips

5.0 ★ Operation #1: Index Potential

5.0 ★ Operation #1: Cache Performance:  
11.8%

3.6 ★ Operation #1: Row Estimate: out by a  
factor of 77.5

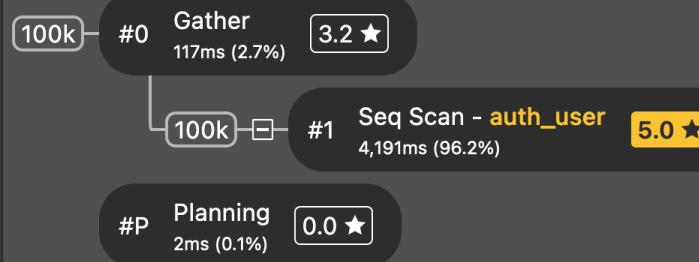
pgMustard rates each tip on a scale of 0-5 stars,  
based on how likely they are to make a significant  
improvement to the performance of your query.

The best tips for your query are summarised above.  
Click on one of the tips to see the performance of  
the operation in more detail.

> 0.6 ★ JIT compilation: 407ms (9.3%)

Total time: 4,358ms

#1



© pgMustard 2018-2022



# Resources

- [PlanetScale B-trees post](#)
- [use-the-index-luke.com](#)
- [Django\\_pgMustard](#)
- [Django's indexes documentation](#)
- [PostgreSQL docs Chapter 11: Indexes](#)

# Thank you! 😊

- [adamj.eu](http://adamj.eu)
- [github.com/adamchainz/talk-data-oriented-django-drei](https://github.com/adamchainz/talk-data-oriented-django-drei)

## Speed Up Your Django Tests

Adam Johnson



## Boost Your Django DX

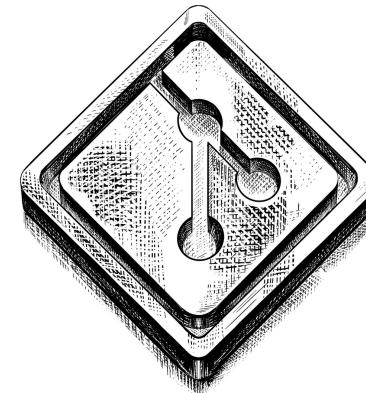
Adam Johnson



Tools and techniques to improve  
your development experience

## Boost Your Git DX

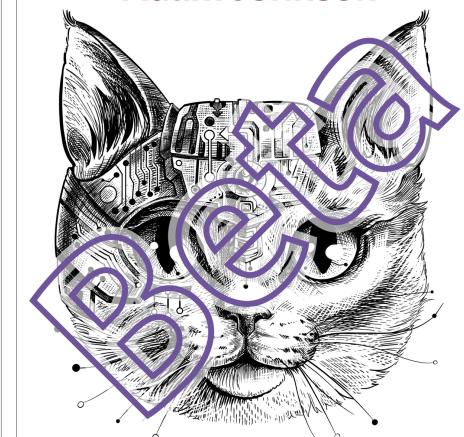
Adam Johnson



Improve your command line  
development experience

## Boost Your GitHub DX

Adam Johnson



Tame the Octocat and elevate  
your productivity