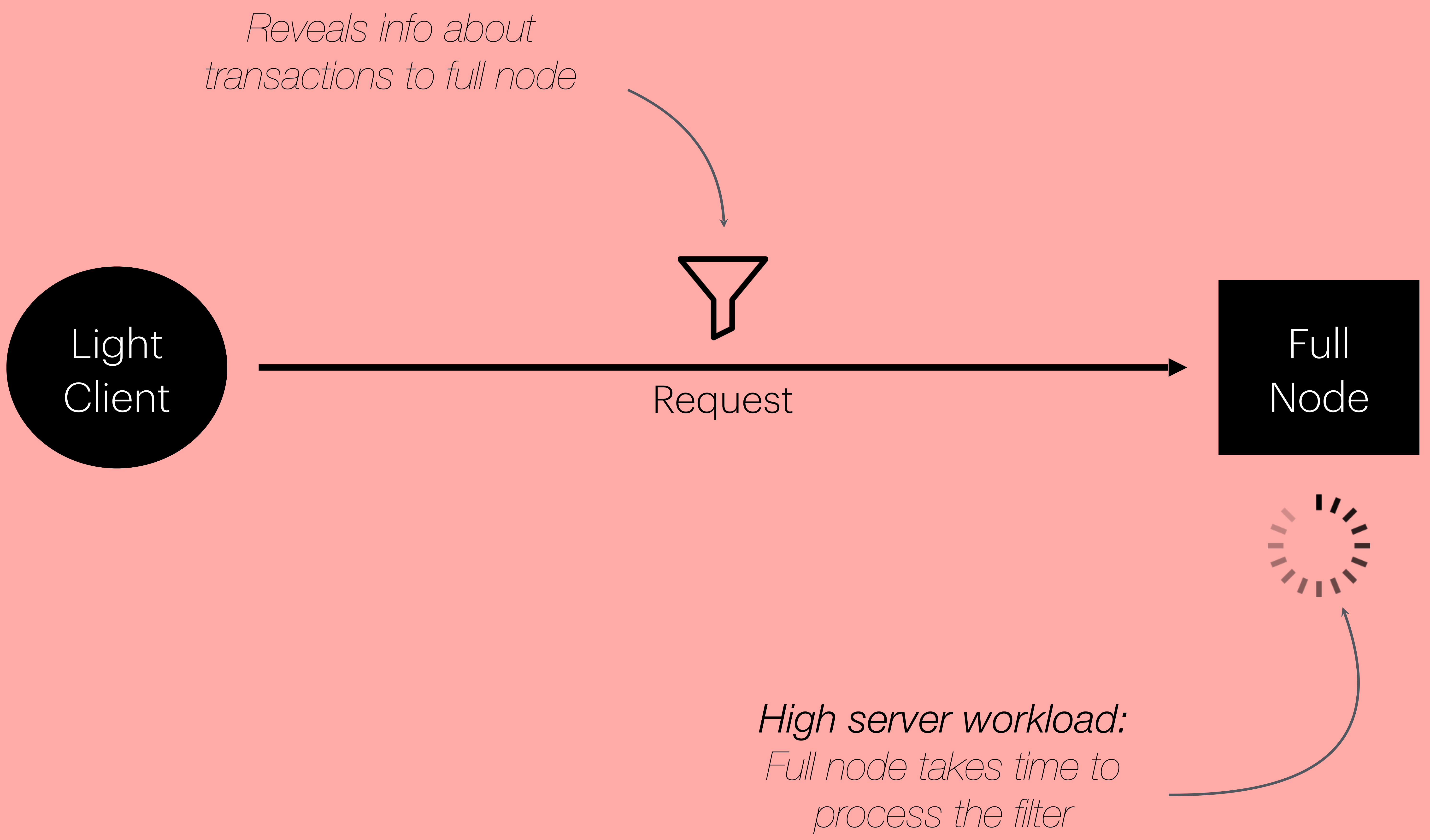
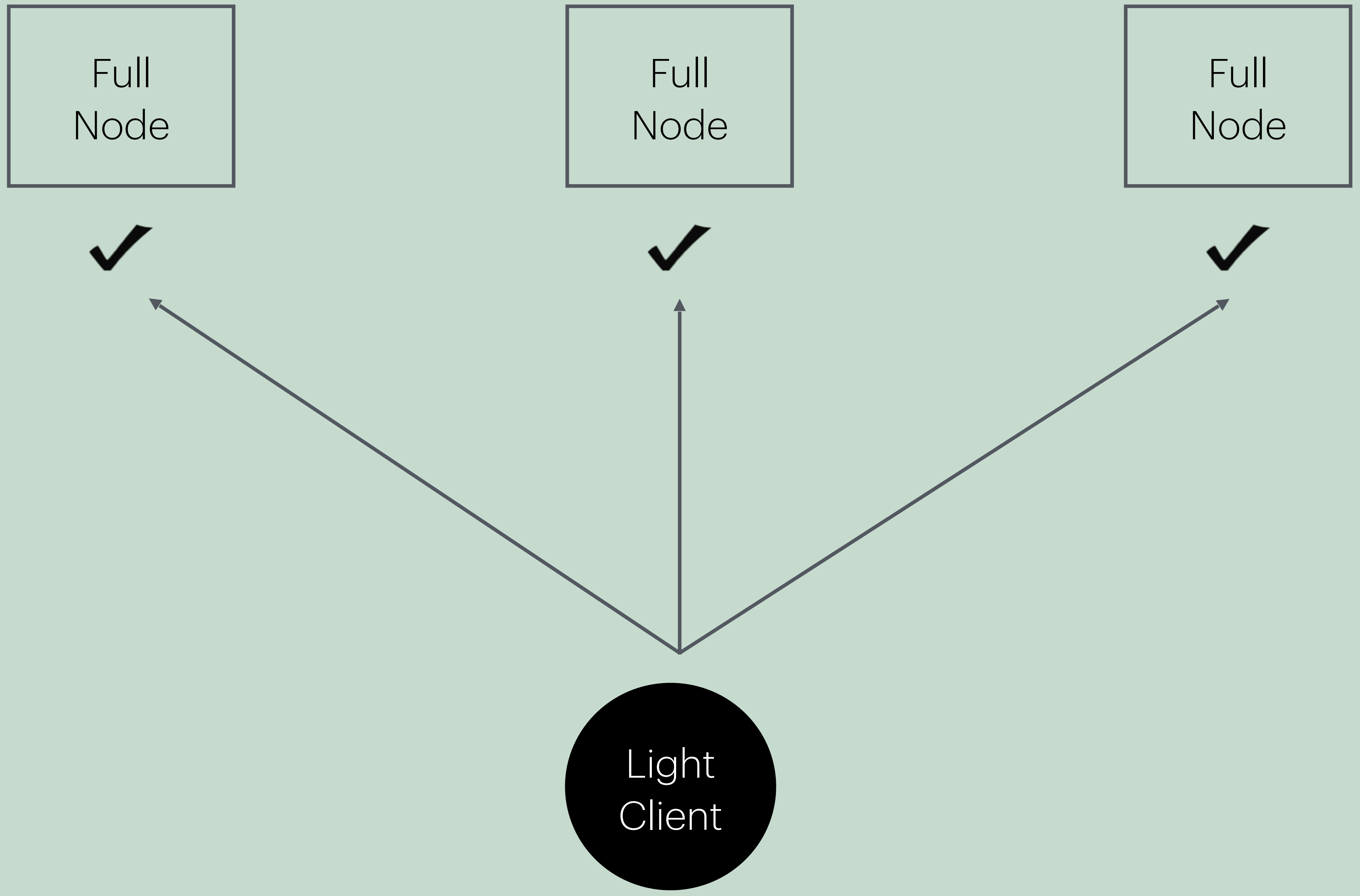
# BIP 158 Compact Block Filter

Before BIP 158, Bitcoin light clients used **Bloom filters** (BIP 37) to detect relevant transactions.

# However Bloom filters had privacy and trust issues



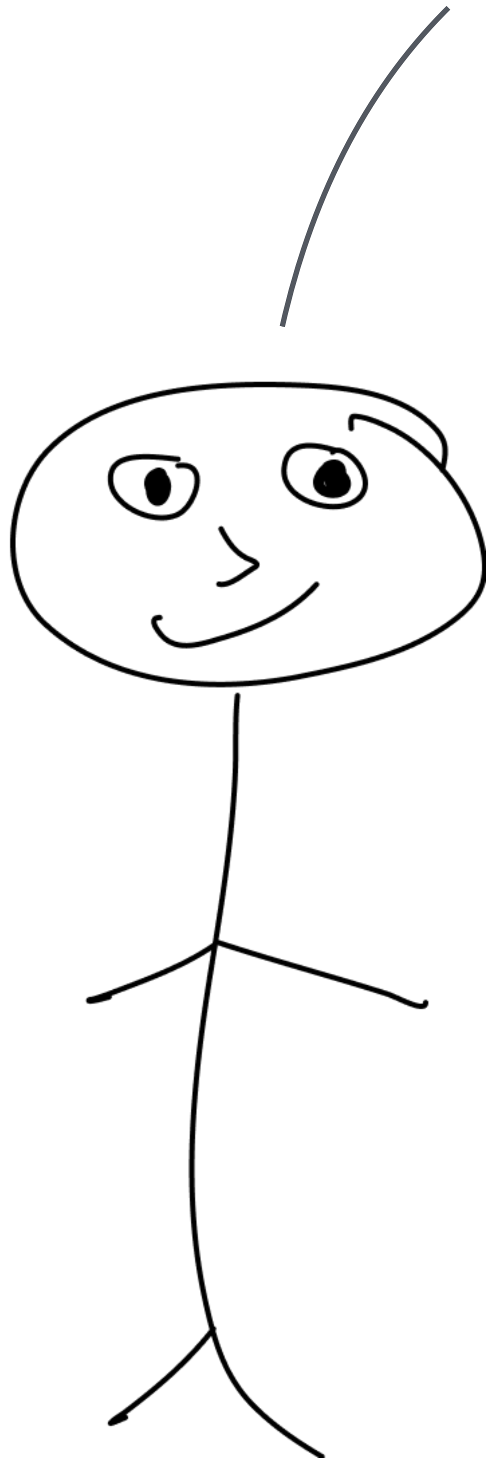
Compact block filters fixes this



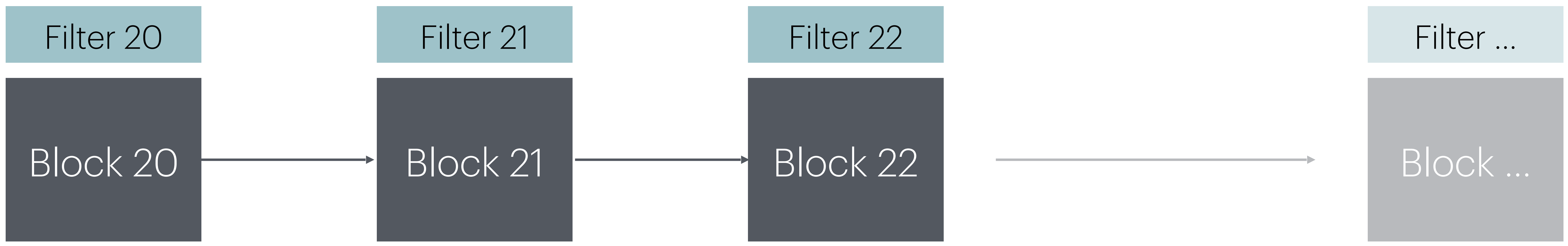

- **Increased privacy:** no filter sharing, harder to analyse activity

- **Reduced server workload:** Nodes calculate filters only once and share them  with light clients

Let me explain how this works



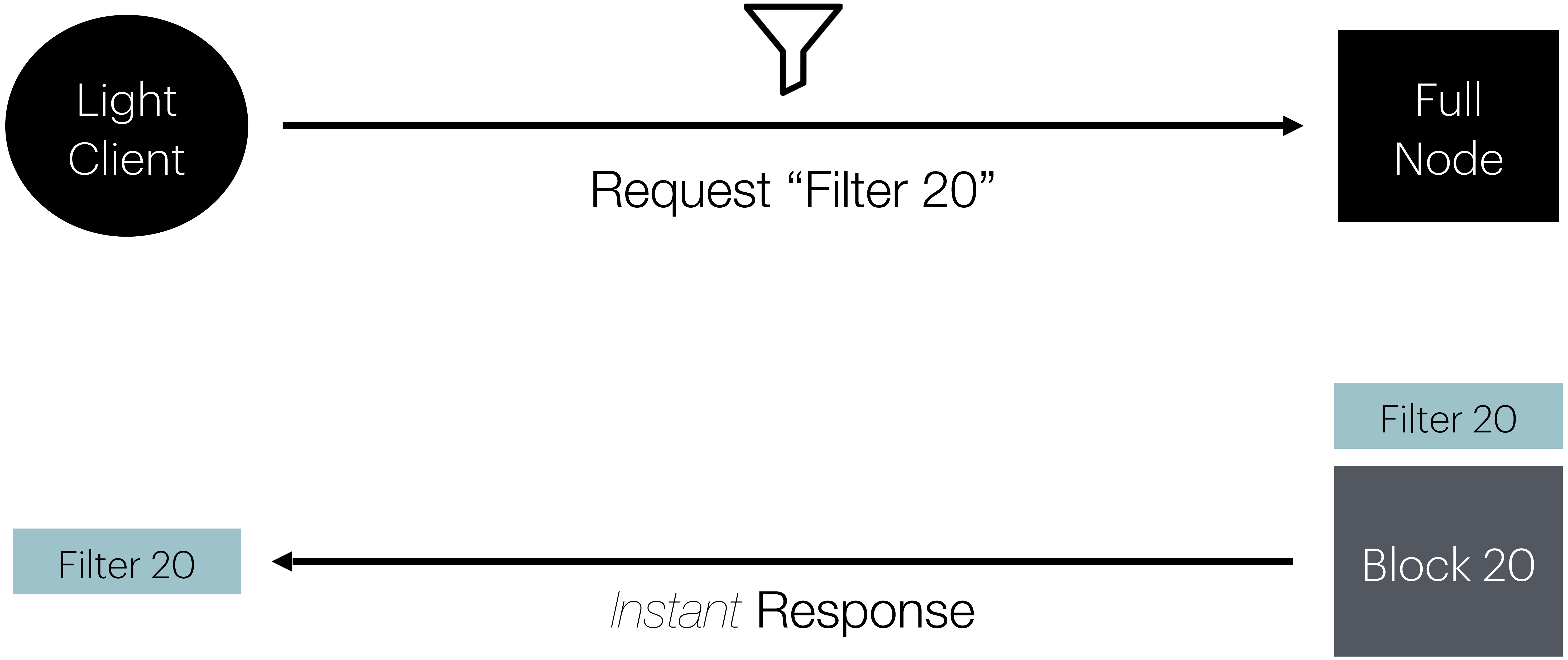
The full node will for each block construct a deterministic filter that includes all the objects* in the block.



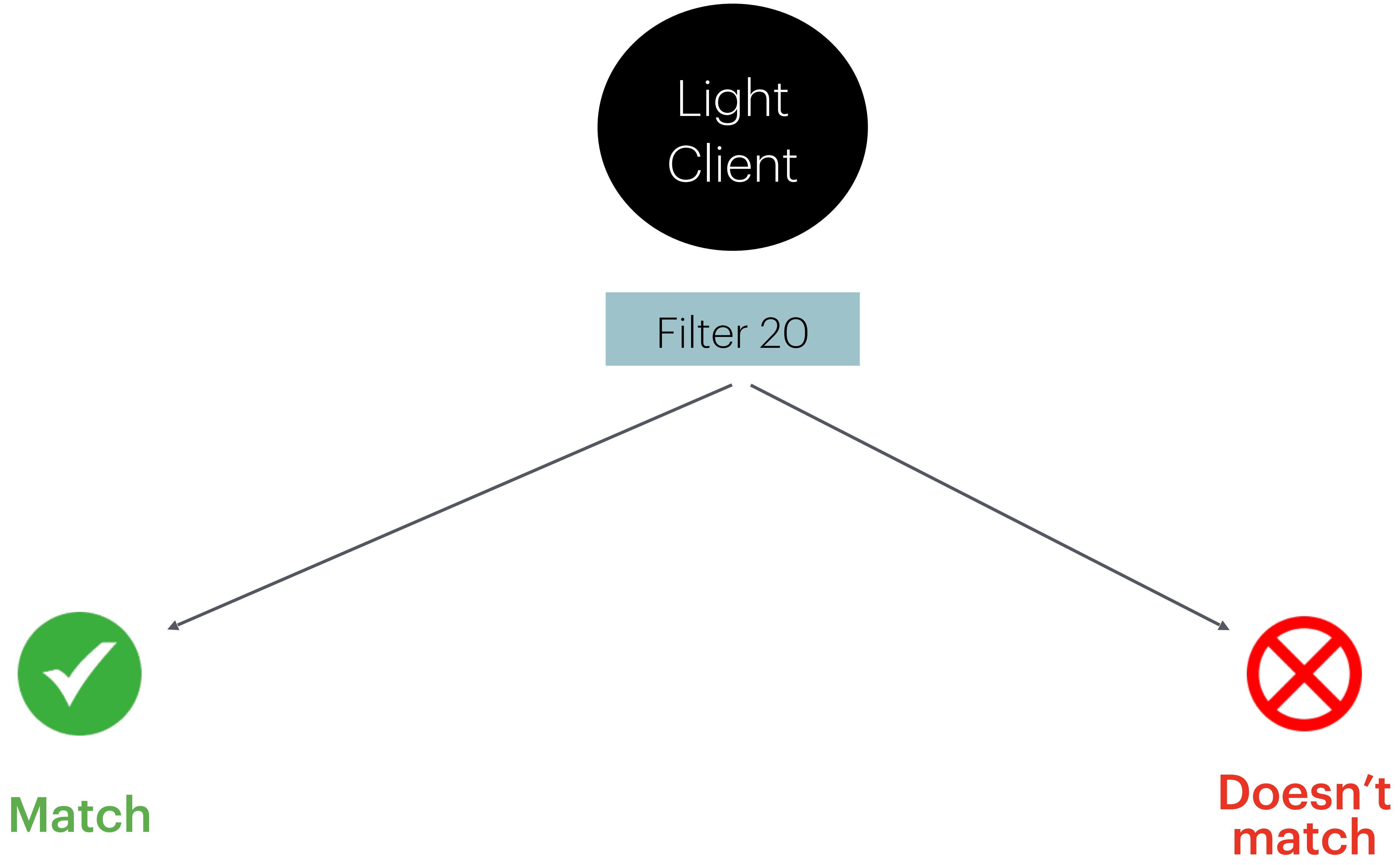

*objects = all txs scriptPubKey*

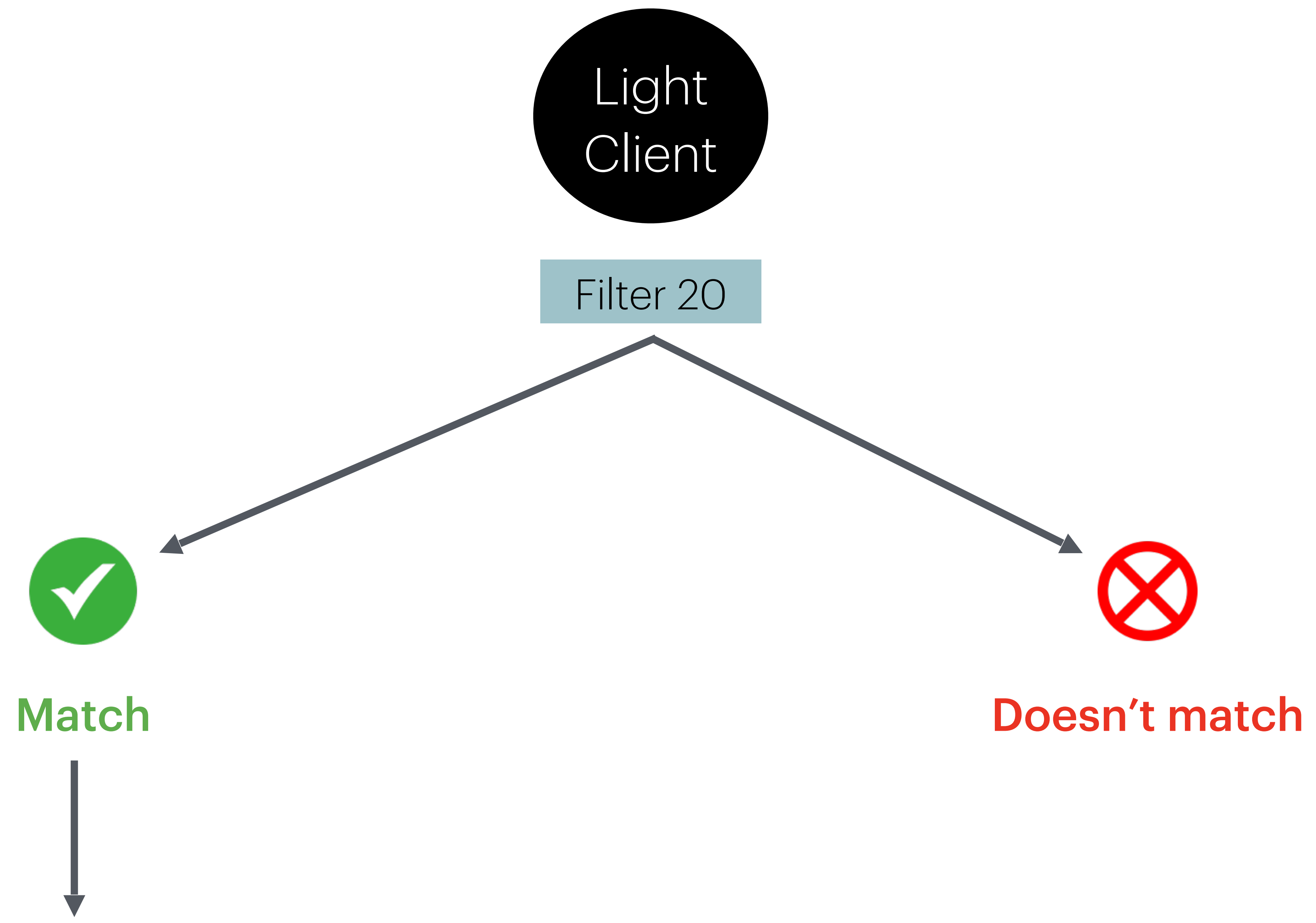
If light clients request a filter for a block 20

server won't have to do any more work, **just deliver**

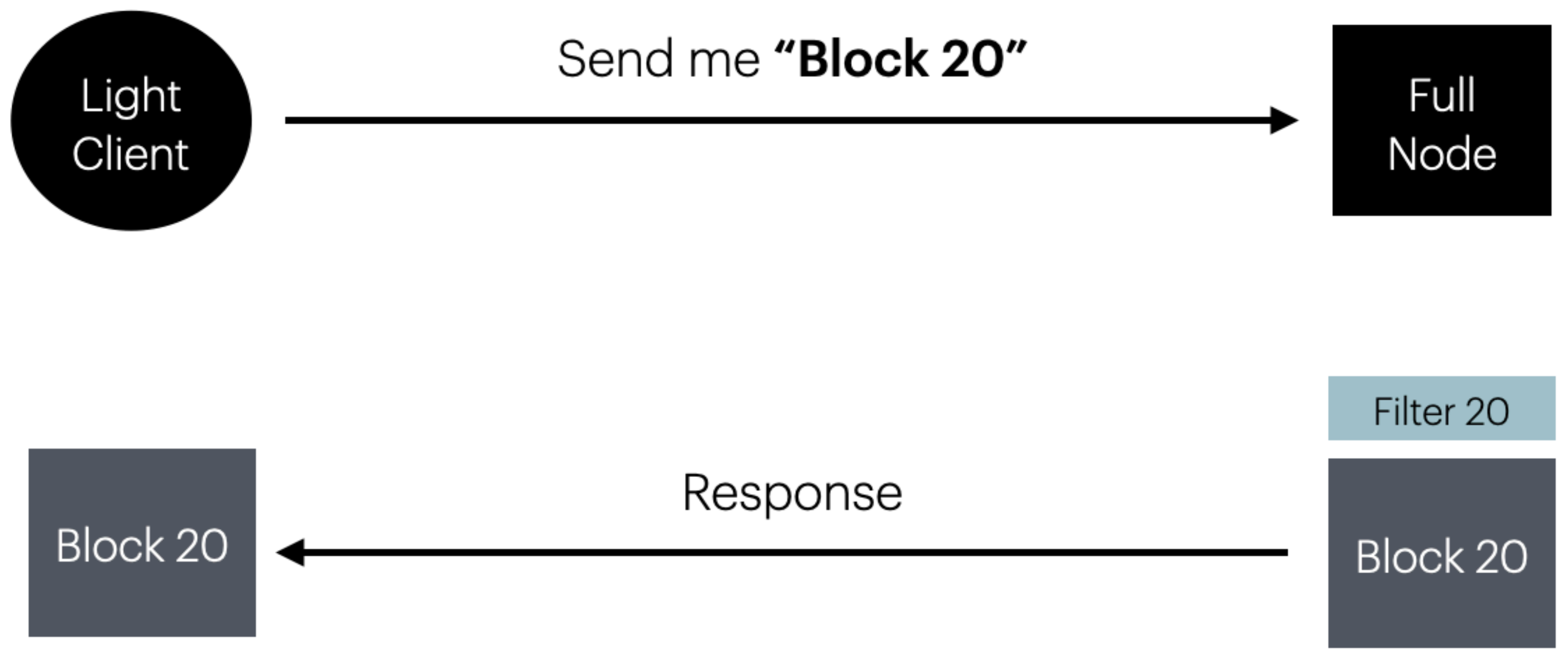


Then Light client checks if any of its objects match what is seen in the filter



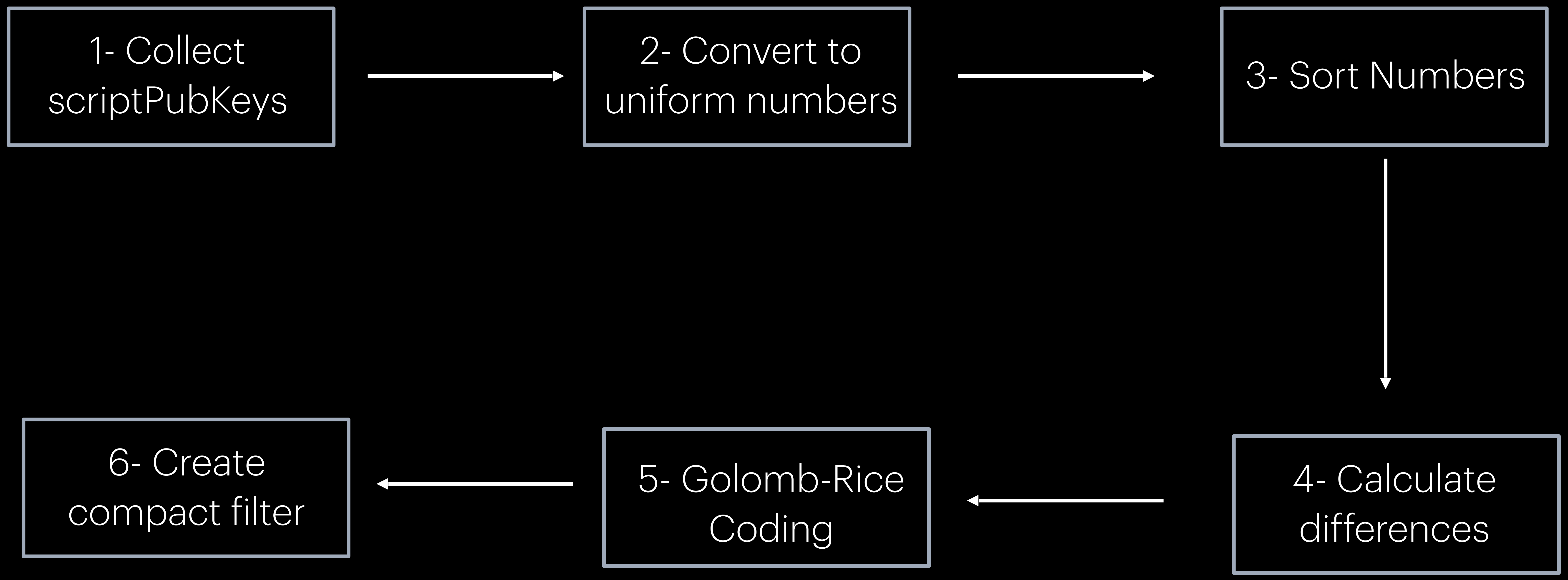



If **match**, then the light client asks for the full block

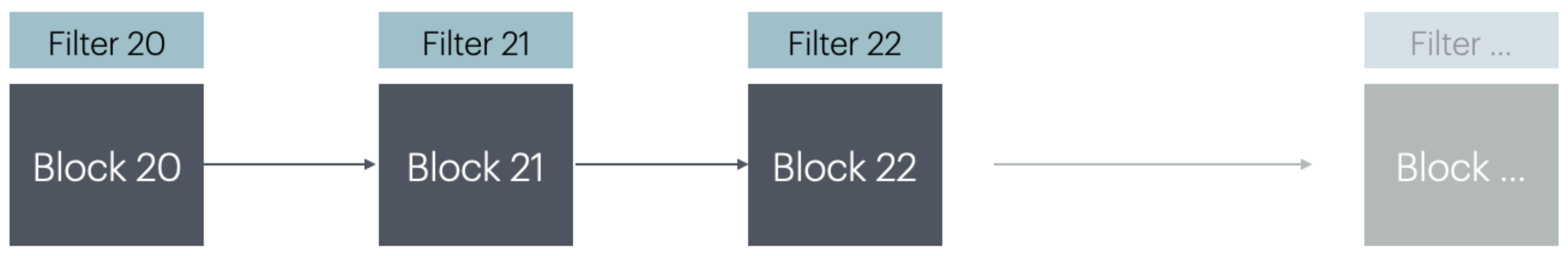


How to create **Filter** ?

# Filter Creation Steps

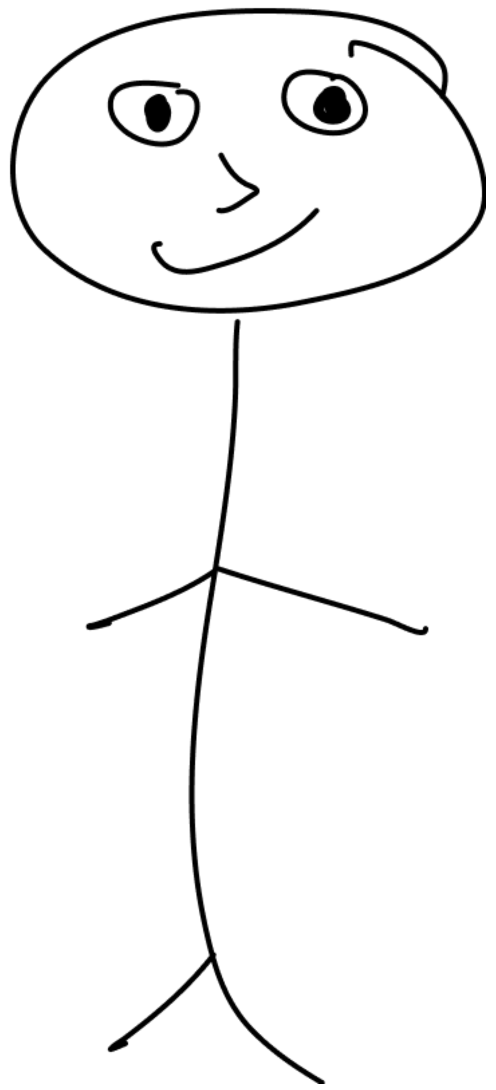


**Full nodes** create a deterministic filter for each block.




**Light clients** download these compact filters to check if a specific Tx inside a block.

Let’s see how full node
create these Filters



Suppose we have a new block with 3 Txs

| | | | | |
|---|---|---|---|---|
| Tx1: | A | → | 1 *BTC* | to B |
| Tx2: | C | → | 0.5 *BTC* | to D |
| Tx3: | E | → | 2 *BTC* | to F |

**Step 1:**

For each Tx, we create set of objects (scriptPubKeys) involved in the inputs and outputs in the filter

```
objects = {1A, 1B, 0C, 0D, 2E, 2F}  // List of N scriptPubKeys.
```

Technically  we could just stop here and say this list of scriptPubKeys is our filter

```
objects = {1A, 1B, 0C, 0D, 2E, 2F}  // List of N scriptPubKeys.
```

With this list a light client could tell if something they are interested in is in the block

But it is still pretty big

Let’s make it compact =>

# Step 2: Mapping objects to numbers

# STEP 2:

# Mapping objects to numbers

We'll map each object to a number uniformly distributed in a range, e.g., [0, 35].

```
objects = {1A, 1B, 0C, 0D, 2E, 2F}  // List of N scriptPubKeys.
```

Function () to turn Object to Number

```
mapped_numbers = {3, 7, 15, 19, 28, 34}
```

STEP 2:

Mapping objects to numbers

That's great !

We have drastically decreased the size of our objects .
Each one has a number now.

```
mapped_numbers = {3, 7, 15, 19, 28, 34}
```

*This is our new filter*

STEP 2:

Mapping objects to numbers

We could stop here… but we can compress this even further!!

# Step 3: Storing differences between numbers

Step 3:

Storing differences between numbers

Instead of storing the numbers, we store the **differences between successive numbers**

```
mapped_numbers = {3, 7, 15, 19, 28, 34}
```

```
differences = {3, 4, 8, 4, 9, 6}
```

Step 3:

Storing differences between numbers

As you can gather, storing the number *34* requires way more bits than storing the number *6*

```
mapped_numbers = {3, 7, 15, 19, 28, 34}
```

↓

```
differences = {3, 4, 8, 4, 9, 6}
```

## Step 3:

## Storing differences between numbers

But why stop there? We can compress this even further!

# Step 4: Golomb-Rice encoding

Step 4:

Golomb-Rice encoding

To compress further.

For each number in the list of differences, we calculate:

**Quotient** = number // M

**Remainder** = number % M

*Let's assume M = 4:*

```
differences = {3, 4, 8, 4, 9, 6}
```

Step 4/1:

Calculate Quotients & Remainders

Let's calculate **quotients** and **remainders**

```
differences = {3, 4, 8, 4, 9, 6}
```

| Number | Quotient = number // 4 | Remainder = number % 4 |
|---|---|---|
| **3** | 0 | 3 |
| **4** | 1 | 0 |
| **8** | 2 | 0 |
| **4** | 1 | 0 |
| **9** | 2 | 1 |
| **6** | 1 | 2 |

Step 4/2:

Encode Quotients

Encode quotients using **unary coding**

| Number | Quotient | Quotient Encoded |
|---|---|---|
| **3** | 0 | 0 |
| **4** | 1 | 10 |
| **8** | 2 | 110 |
| **4** | 1 | 10 |
| **9** | 2 | 110 |
| **6** | 1 | 10 |

Step 4/2:

# Encode Remainders

Encode remainders using **binary representation**

| Number | Remainder | Remainder Encoded |
|---|---|---|
| **3** | 3 | 11 |
| **4** | 0 | 00 |
| **8** | 0 | 00 |
| **4** | 0 | 00 |
| **9** | 1 | 01 |
| **6** | 2 | 10 |

Step 4/3:

Concatenation

Concatenate encoded quotients and remainders:

| Number | Quotient Encoded | Remainder Encoded | **Concatenation** |
|---|---|---|---|
| **3** | 0 | 11 | **011** |
| **4** | 10 | 00 | **1000** |
| **8** | 110 | 00 | **11000** |
| **4** | 10 | 00 | **1000** |
| **9** | 110 | 01 | **11001** |
| **6** | 10 | 10 | **1010** |

## Step 4/4:

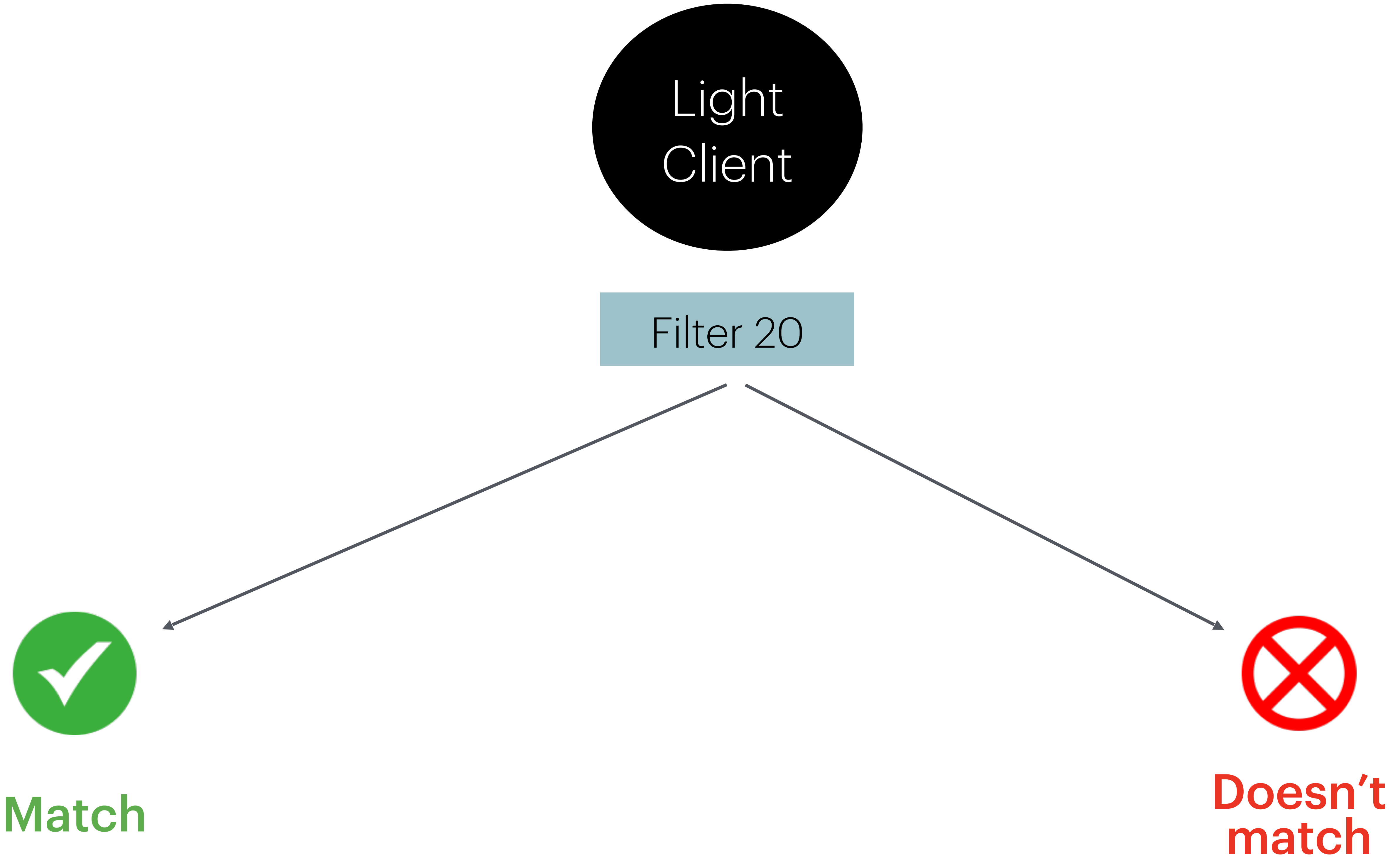
## Concatenation

Now our final filter becomes

| Concatenation |
| --- |
| 011 |
| 1000 |
| 11000 |
| 1000 |
| 11001 |
| 1010 |

```
encoded_filter = "011 1000 11000 1000 11001 1010"
```

The light client requests this filter from the full node.




After receiving it, the client **decodes** it and checks if any of its objects (addresses) **match** the filter.

If there's a match, the light client requests the full block to verify and process the relevant transaction

(e.g., a payment received or sent).