Right Inside the Database

David Bach, Malte Sandstede

RustFest Barcelona, November ‘19

Imagine A World...

- ...where obtaining backend data is as simple as writing a query
- ...where your apps are always up-to-date
- ...where “offline-first” is only a caching decision
- ...where ORMs, resolvers, mappers, etc. do not exist

...where you are right inside the database!

(roll credits)

clockworks

Malte Sandstede

malte@clockworks.io / @MalteSandstede

David Bach

david@clockworks.io / @bachdavi

Nikolas Göbel

niko@clockworks.io

Moritz Moxter

moritz@clockworks.io

In collaboration with:

Frank McSherry

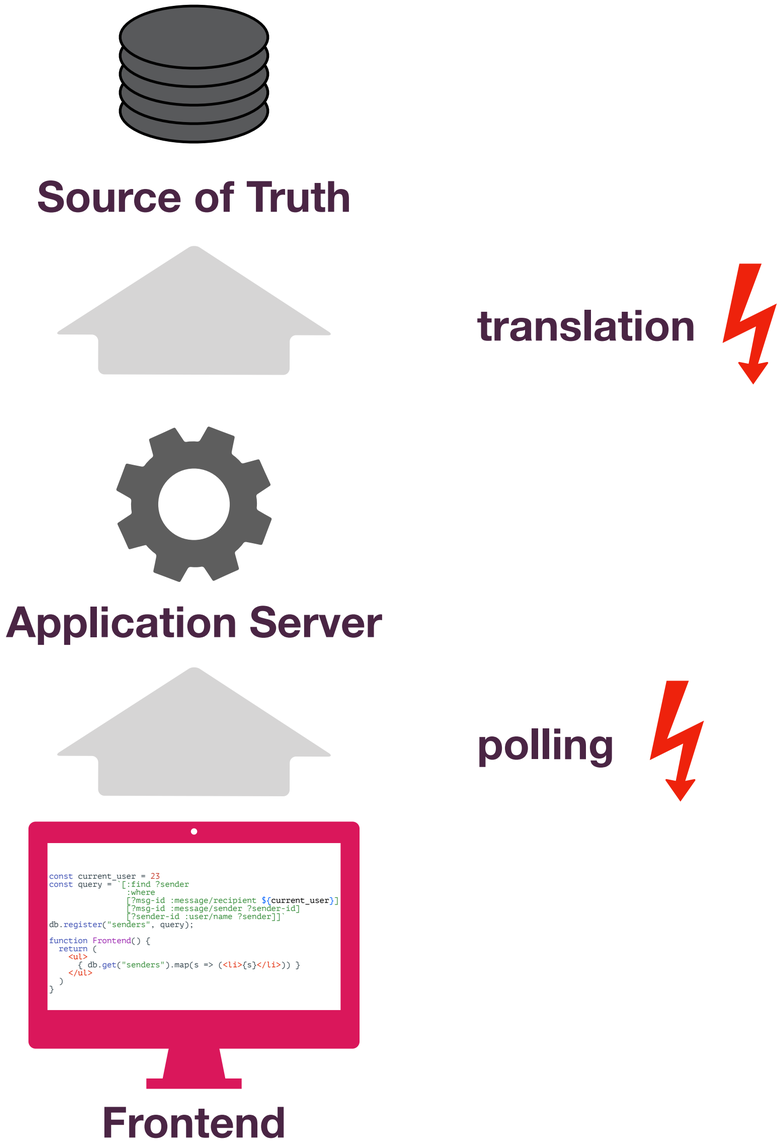
ETH zürich

Systems Group

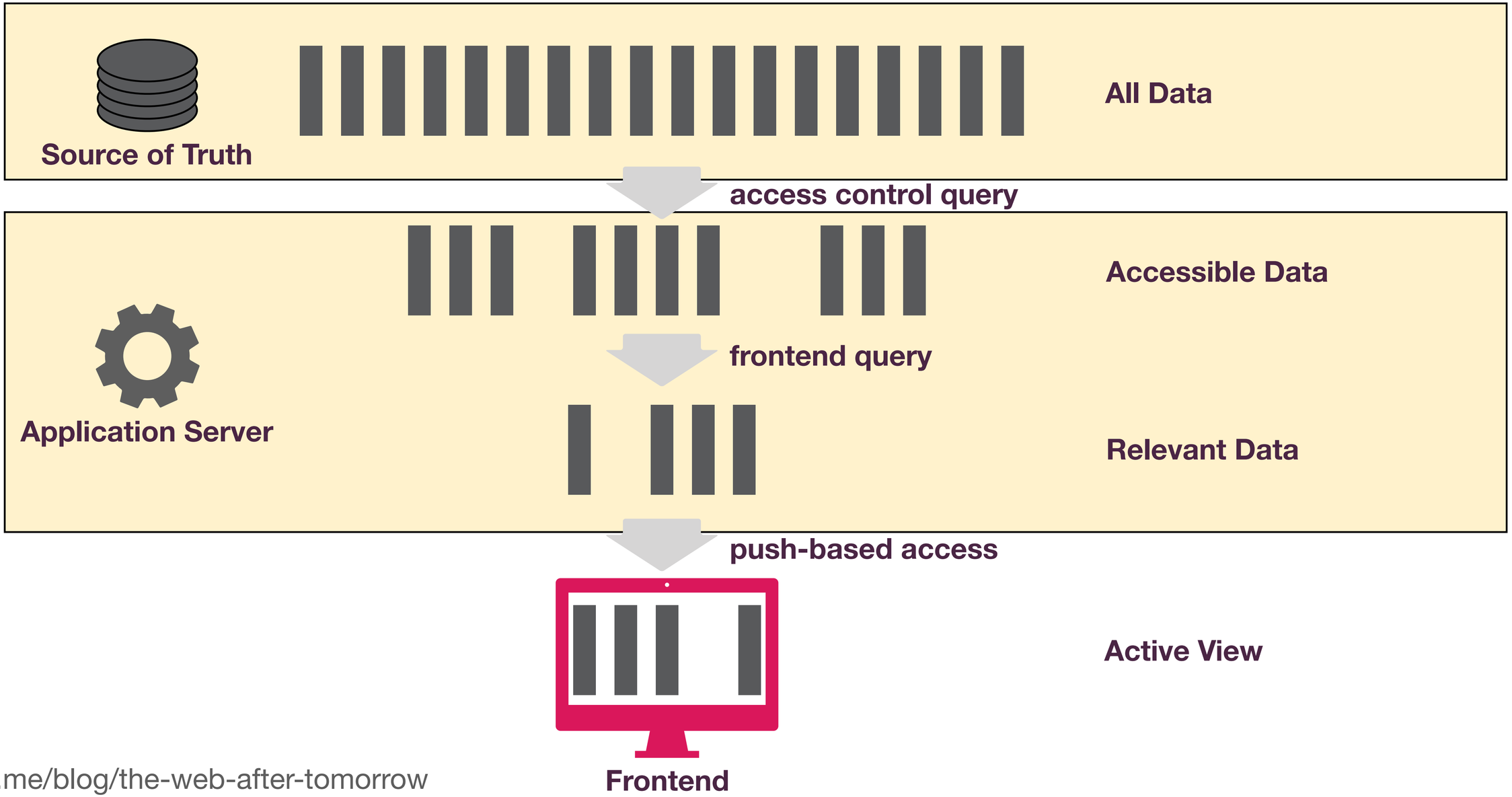
Declarative & Reactive Frontends

```
const current_user = 23
const query = `[:find ?sender
                :where
                [?msg-id :message/recipient ${current_user}]
                [?msg-id :message/sender ?sender-id]
                [?sender-id :user/name ?sender]]`
db.register("senders", query);

function Frontend() {
  return (
    <ul>
      { db.get("senders").map(s => (<li>{s}</li>)) }
    </ul>
  )
}
```

Declarative & Reactive Full Stack?



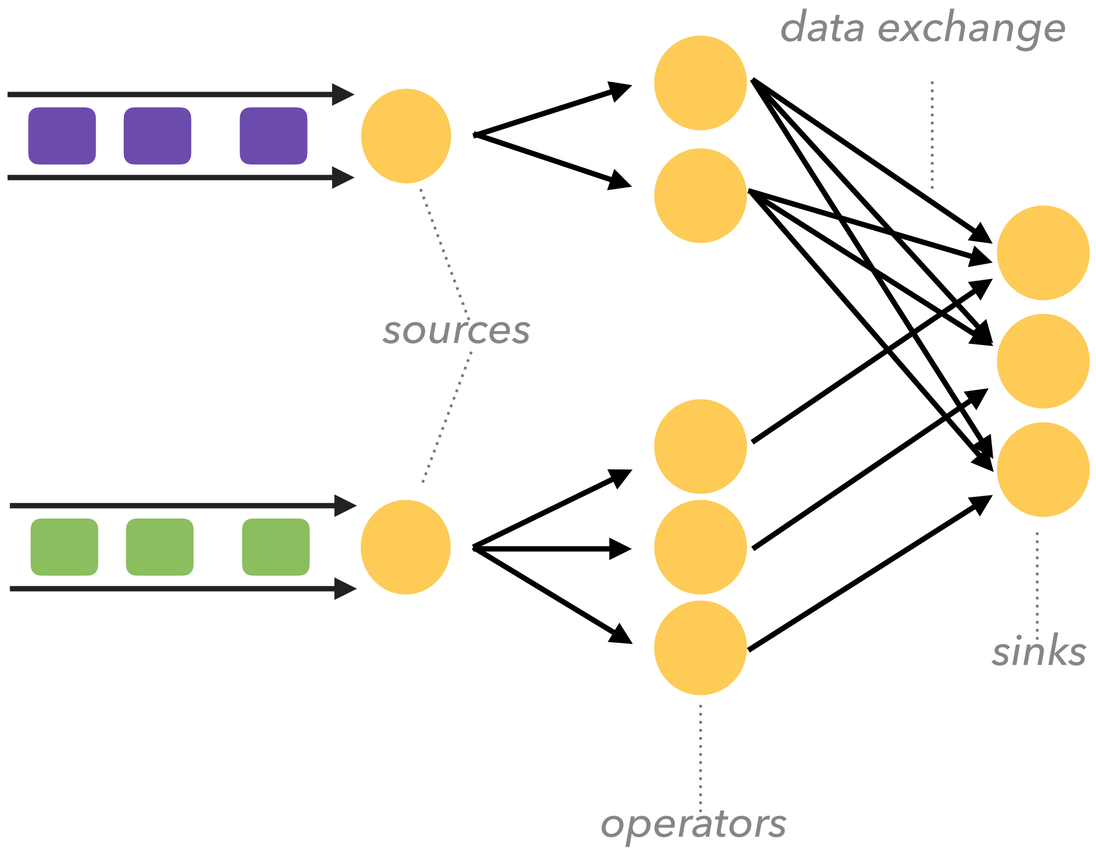
Declarative & Reactive Full Stack!



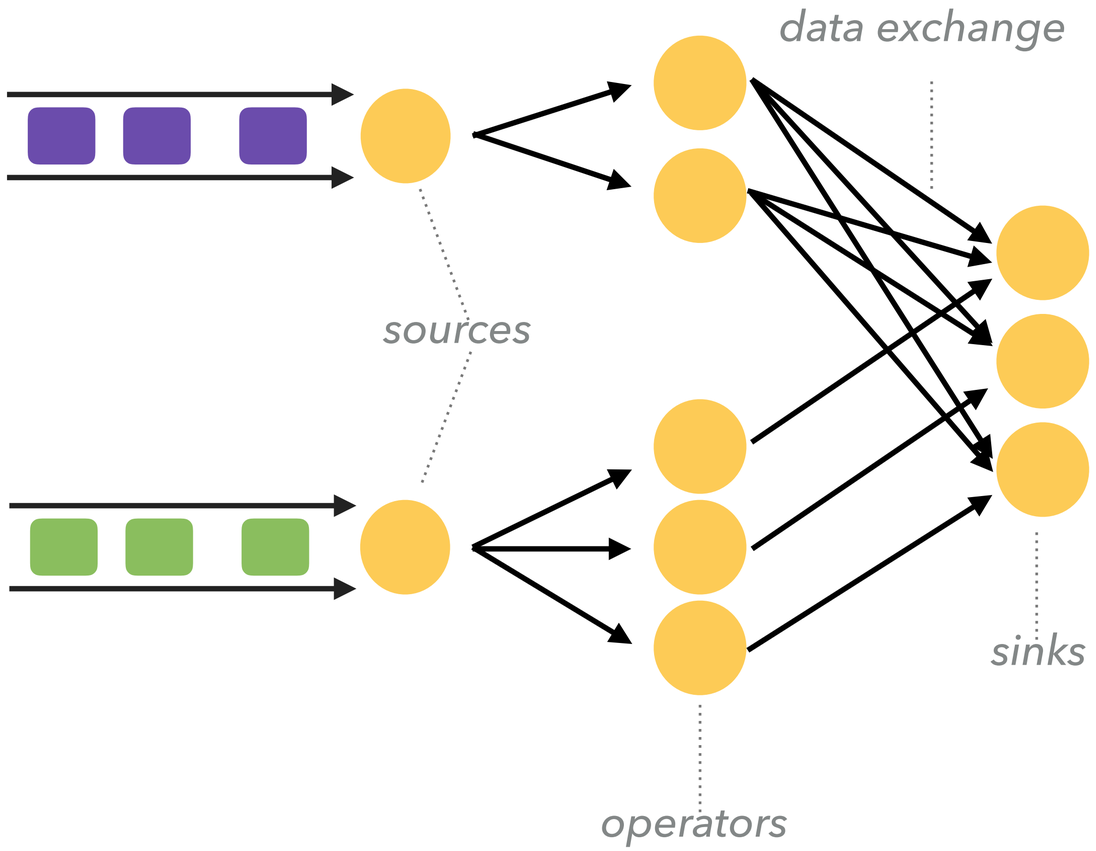
tonsky.me/blog/the-web-after-tomorrow

A Push-Based Programming Model

Dataflow Programming

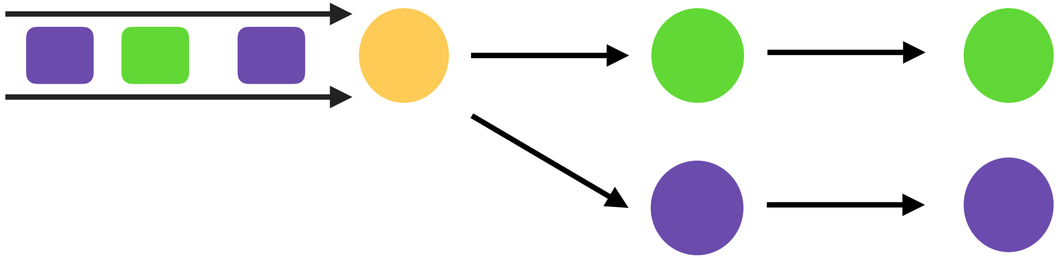


Dataflow Programming



- No central coordination
- Operator owns data

Dataflow Programming



- No central coordination
- Operator owns data
- Easy to distribute

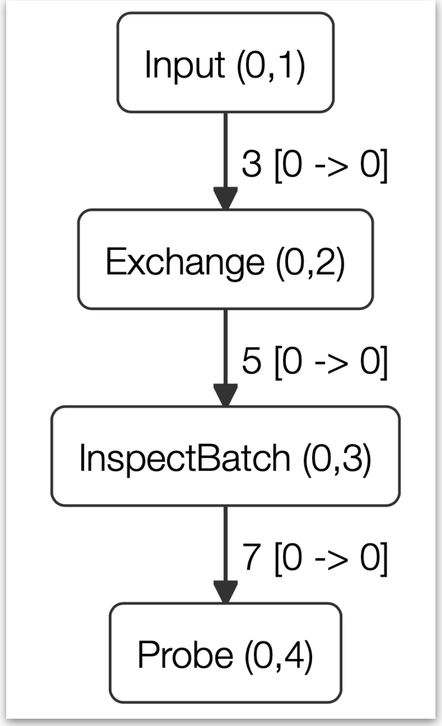
Timely Dataflow

A low-latency runtime for
distributed cyclic dataflows

github.com/**TimelyDataflow**

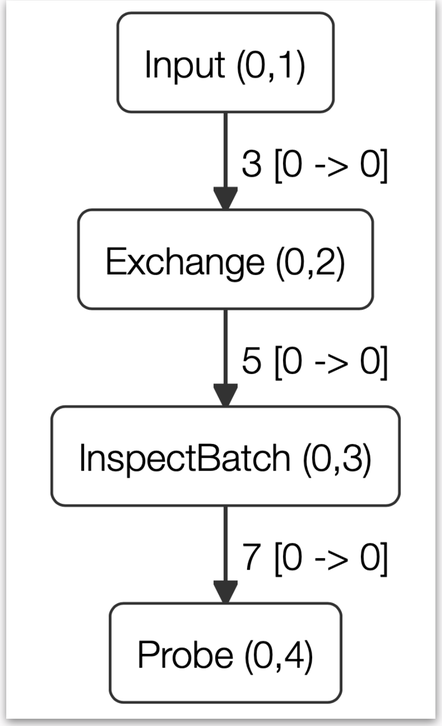
Creating Dataflows with Timely

```rust
fn main() {
    timely::execute_from_args(std::env::args(), |worker| {

        // Some computation
        let mut input = InputHandle::new();
        let probe = worker.dataflow(|scope|
            scope.input_from(&mut input)
                 .exchange(|x| *x as u64 + 1)
                 .inspect(move |x| println!("record {}", x))
                 .probe()
        );

        for round in 0..100 {
            if worker.index() == 0 { (0..20).for_each(|i| input.send(i) ) }
            input.advance_to(round + 1);
            while probe.less_than(input.time()) { worker.step(); }



        }
    }).unwrap();
}
```

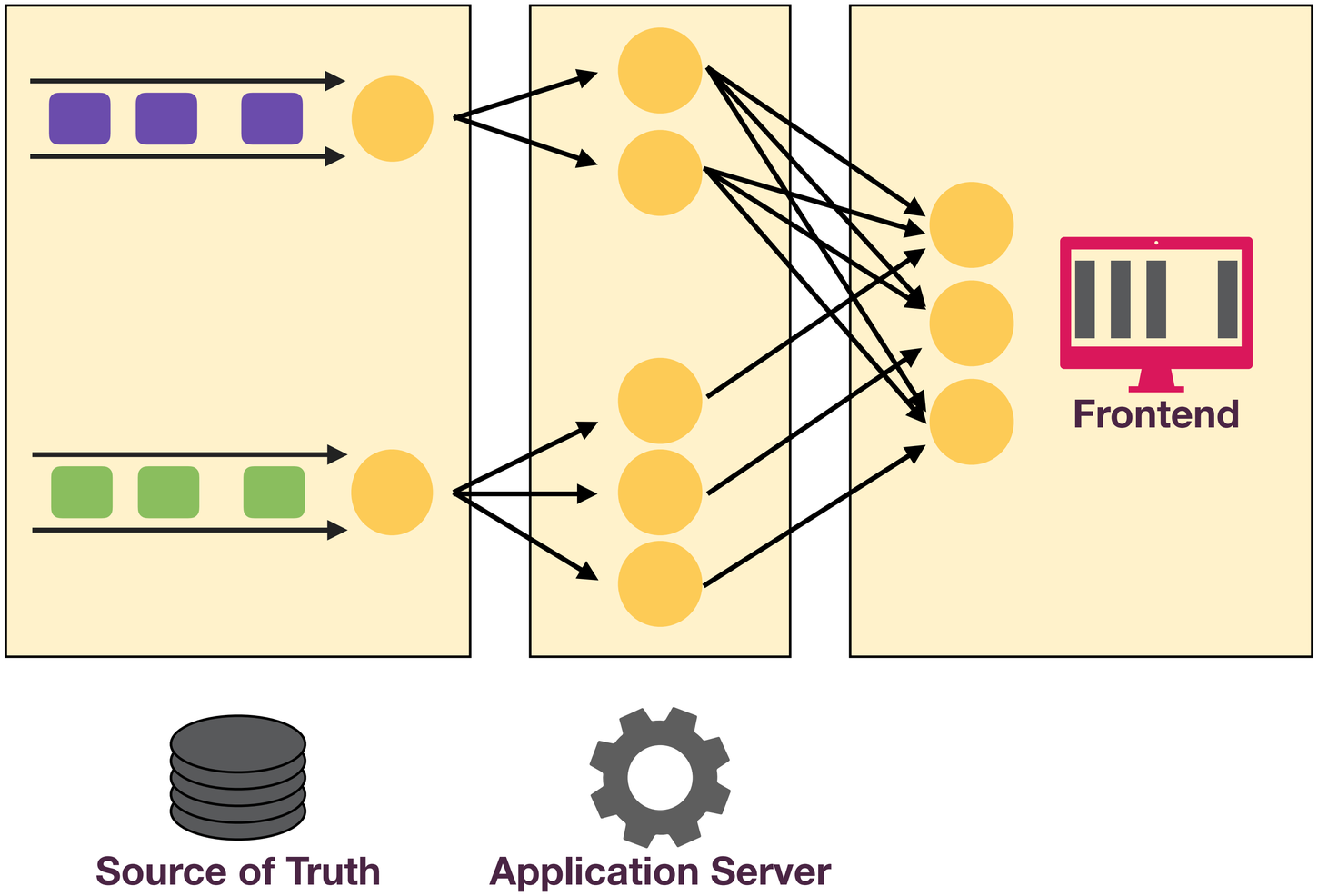


Running Dataflows with Timely

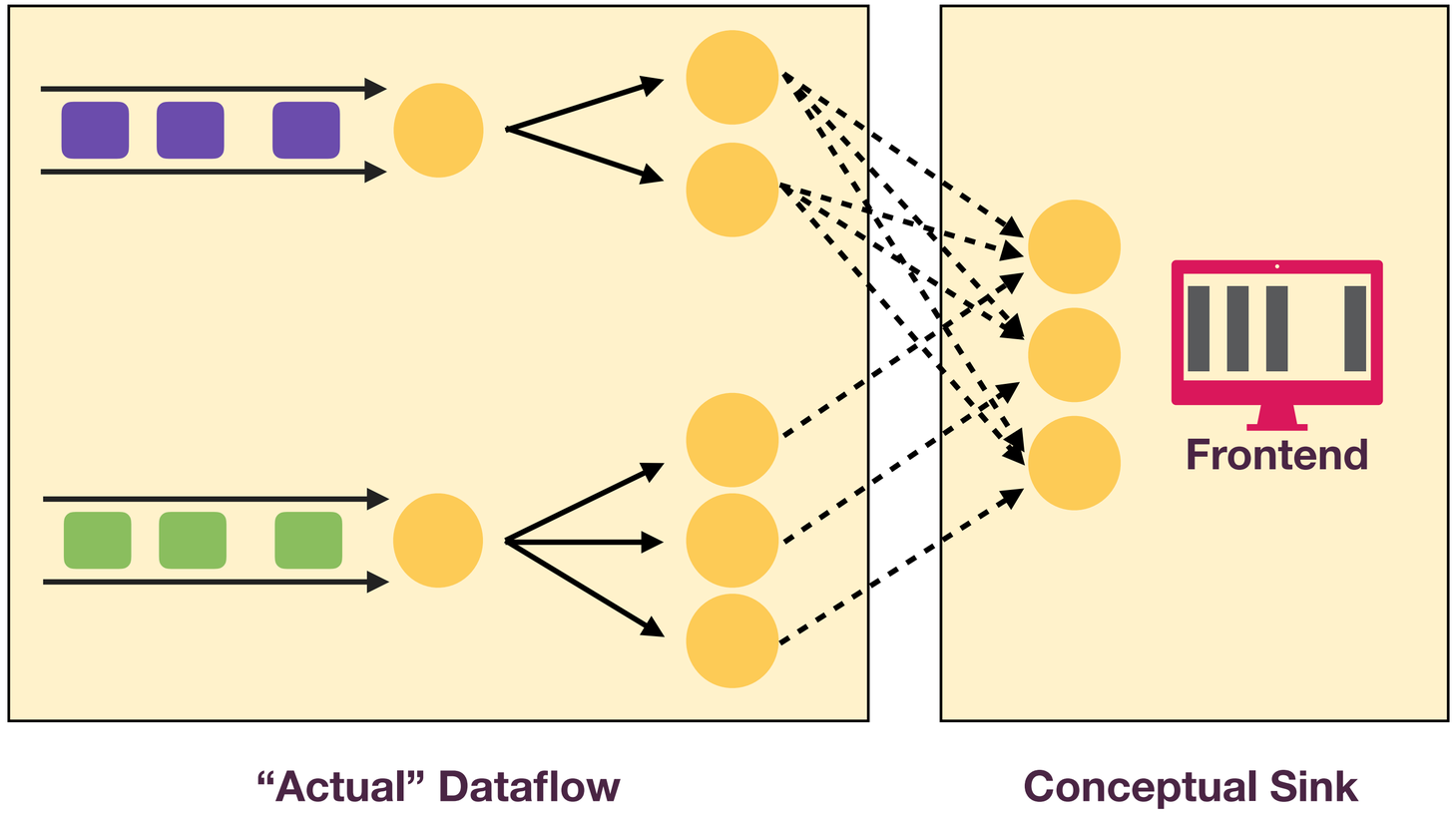
```rust
fn main() {
    timely::execute_from_args(std::env::args(), |worker| {

        // Some computation
        let mut input = InputHandle::new();
        let probe = worker.dataflow(|scope|
            scope.input_from(&mut input)
                 .exchange(|x| *x as u64 + 1)
                 .inspect(move |x| println!("record {}", x))
                 .probe()
        );

        for round in 0..100 {
            if worker.index() == 0 { (0..20).for_each(|i| input.send(i) ) }
            input.advance_to(round + 1);
            while probe.less_than(input.time()) { worker.step(); }



        }
    }).unwrap();
}
```



Full Stack Dataflow

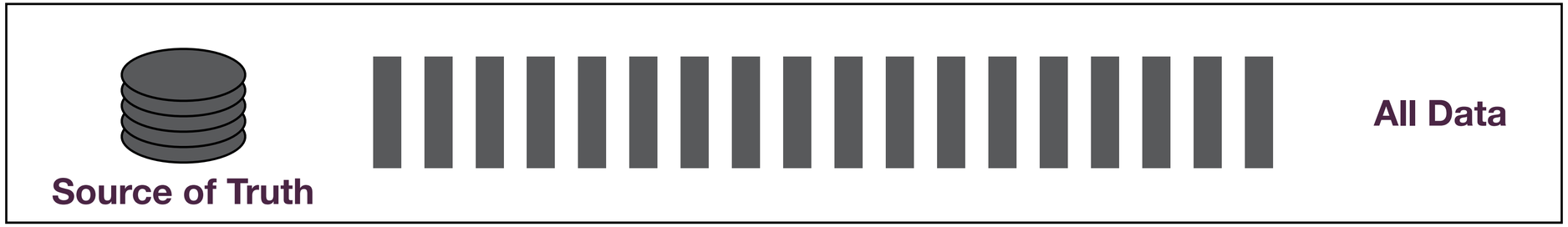


Full Stack Dataflow



The Dataflow Backend

Data Modeling for Dataflows

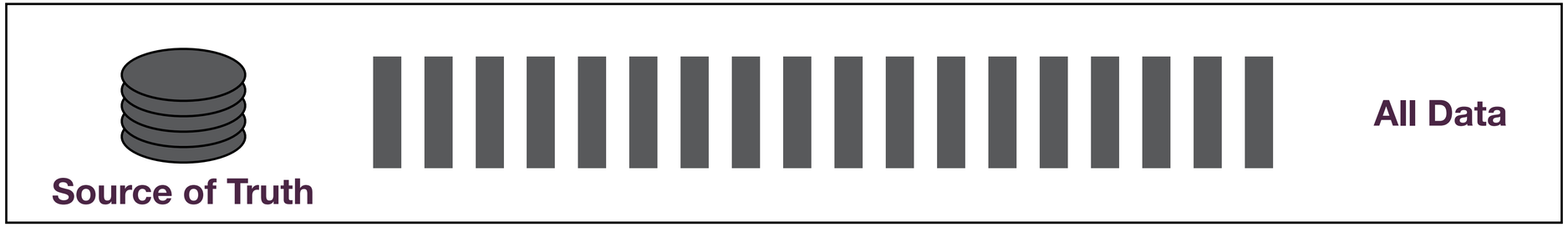


- Fully normalized *attribute-oriented* data model
- Fundamental unit: **`Fact`** `:= (e a v)`

  `entity` **`attribute`** **`value`**

- Facts are composable into higher-level concepts

Data Modeling for Dataflows




```
[1 :person/name Alice]
[1 :person/age  32]
[1 :person/residence 4]

[4 :residence/city NY]
[4 :residence/country USA]
[4 :residence/zip 10002]
```

```
Person {
  name
  age
  residence
}
```

```
Residence {
  city
  country
  zip
}
```

Attribute Tables to `(e v)` Streams

:person/name	
e	**v**
1	Alice
2	John
1	Bob

```
[[1 Alice t0 +1]
 [2 John  t0 +1]
 [1 Alice t2 -1]
 [1 Bob   t2 +1]]
```

add/retract

time

- Binary relation (“table”) for each attribute
- Conversion to stream by adding time
- Explicit multiplicities enable data-driven reactions
- Consolidated streams at a common t* provide consistent view

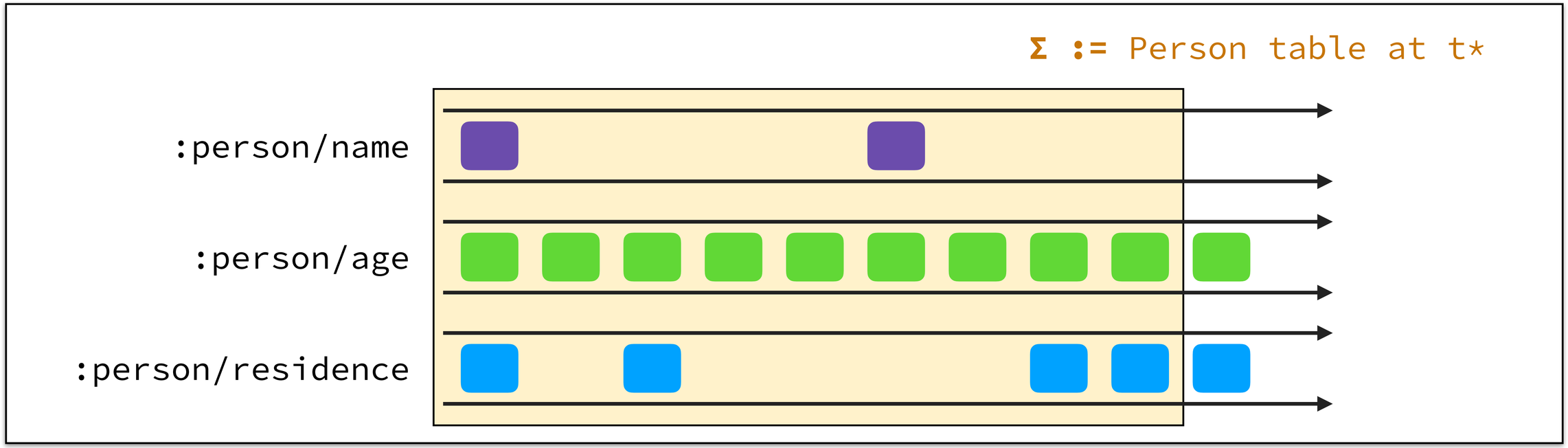
Attribute Tables to `(e v)` Streams

:person/name	
e	**v**
1	Alice
2	John
1	Bob

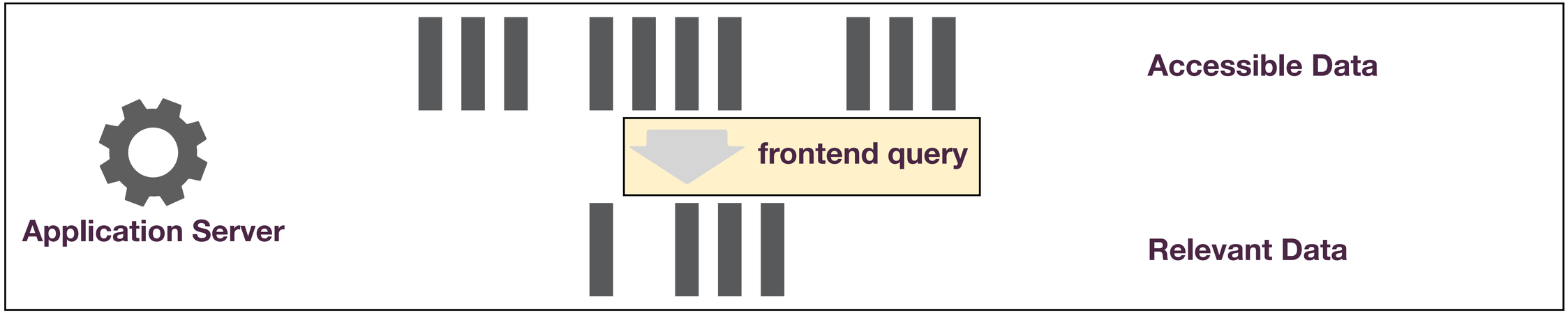
```
[[1 Alice t0 +1]
 [2 John  t0 +1]
 [1 Alice t2 -1]
 [1 Bob   t2 +1]]
```

add/retract

time



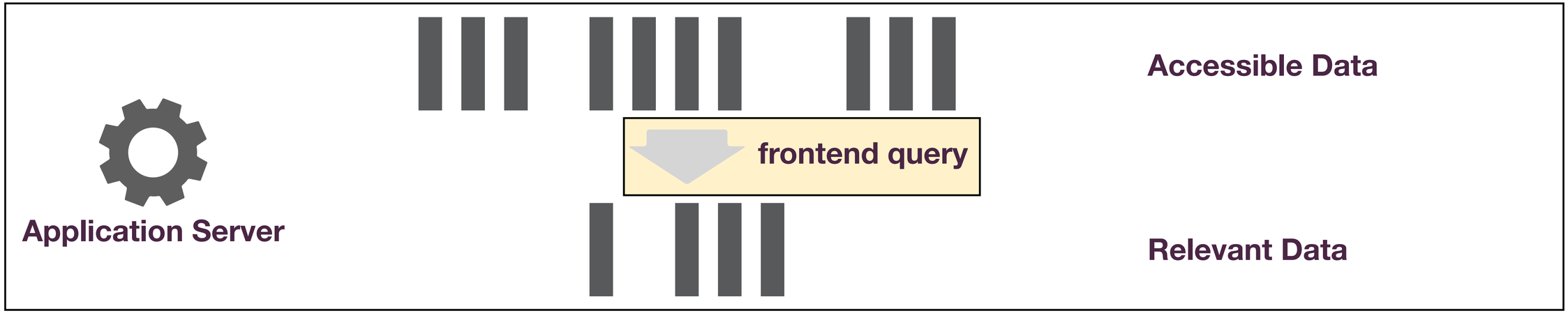
Expressing Queries as Dataflows




```
[[:find ?to
  :where
  (reach :Alice ?to)]

 [(reach ?from ?to)
  [?from :edge ?to]]

 [(reach ?from ?to)
  [?from :edge ?hop]
  (reach ?hop ?to)]]
```

- Register frontend demand as query with backend
- Updates are pushed via WebSocket connection
- Also encode access policies as queries

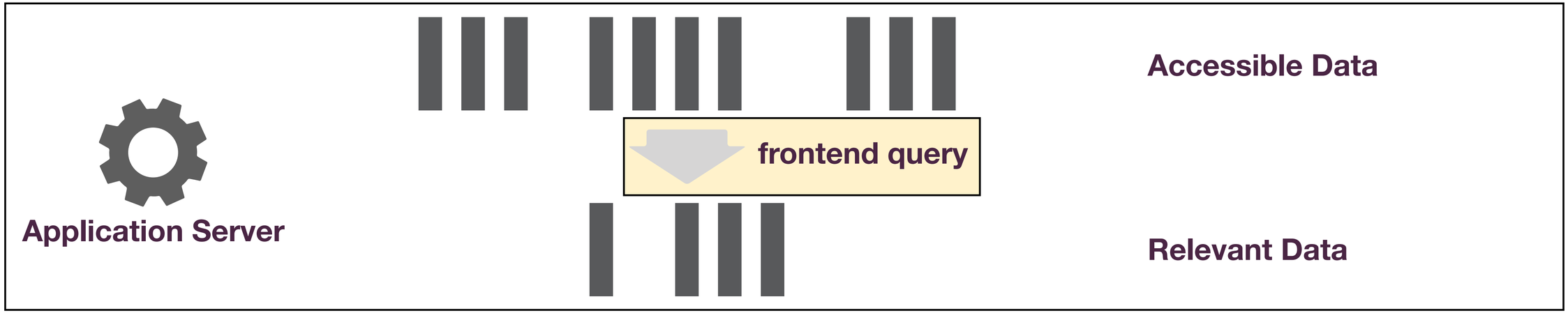
Expressing Queries as Dataflows




```
[[:find ?to
  :where
  (reach :Alice ?to)]

 [(reach ?from ?to)
  [?from :edge ?to]]

 [(reach ?from ?to)
  [?from :edge ?hop]
  (reach ?hop ?to)]]
```

FACT

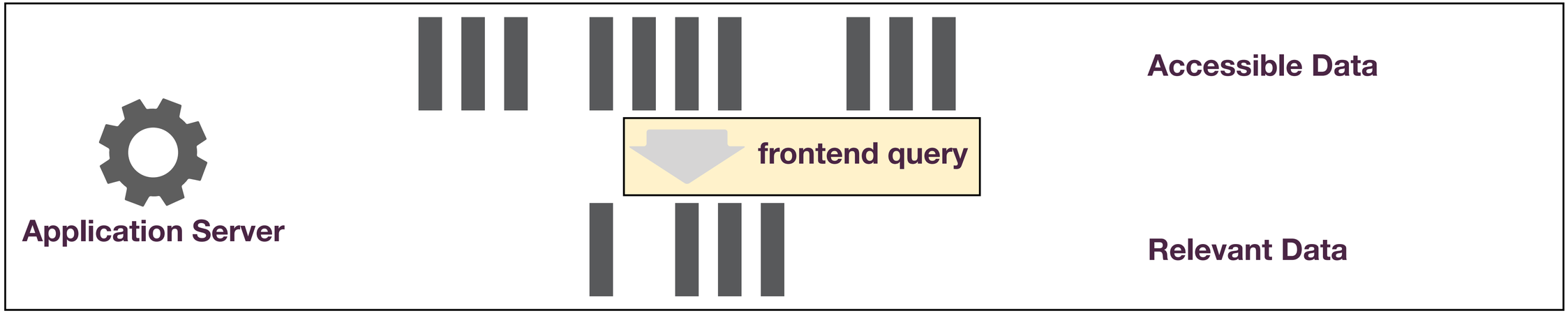
Expressing Queries as Dataflows




```
[[:find ?to
  :where
  (reach :Alice ?to)]

 [(reach ?from ?to)
  [?from :edge ?to]]

 [(reach ?from ?to)
  [?from :edge ?hop]
  (reach ?hop ?to)]]
```

JOIN

Expressing Queries as Dataflows

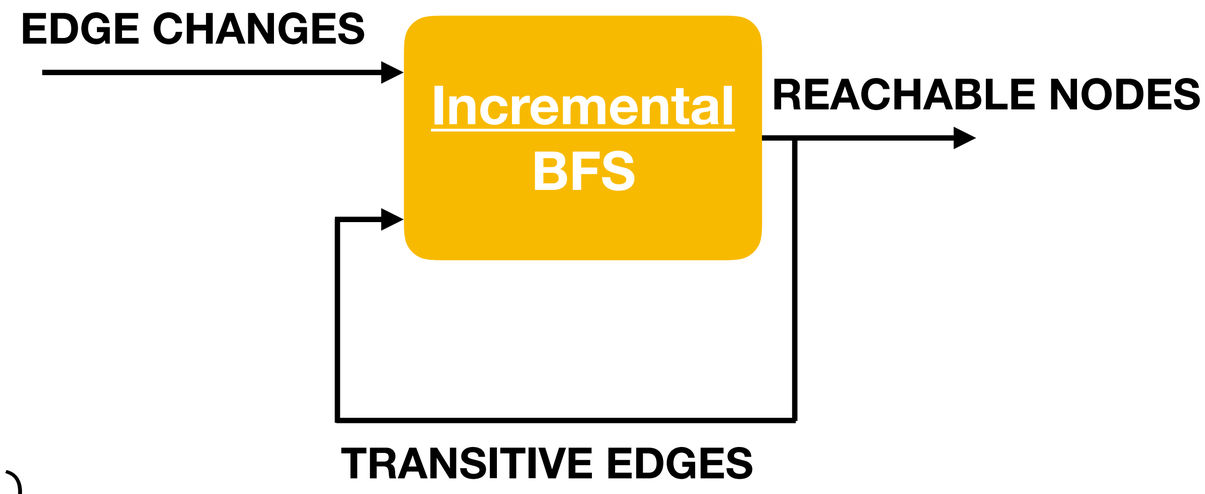



```
[[:find ?to
  :where
  (reach :Alice ?to)]

 [(reach ?from ?to)
  [?from :edge ?to]]

 [(reach ?from ?to)
  [?from :edge ?hop]
  (reach ?hop ?to)]]
```

ITERATE

Differential Dataflow: Complex Operators

```
/// Friends of Friends

let nodes = roots.map(|x| (x, 0));

nodes.iterate(|inner| {

 let edges = edges.enter(&inner.scope());
 let nodes = nodes.enter(&inner.scope());

 inner.join(&edges, |_k,l,d| (*d, l+1))
     .concat(&nodes)
     .reduce(|_, s, t| t.push((*s[0].0, 1)))
})
```



Dynamic Query Interpretation?

```
[[(reach ?from ?to)
  [?from :edge ?to]]

 [(reach ?from ?to)
  [?from :edge ?hop]
  (reach ?hop ?to)]]
```

```
/// Friends of Friends

let nodes = roots.map(|x| (x, 0));

nodes.iterate(|inner| {

 let edges = edges.enter(&inner.scope());
 let nodes = nodes.enter(&inner.scope());

 inner.join_map(&edges, |_k,l,d| (*d, l+1))
     .concat(&nodes)
     .reduce(|_, s, t| t.push((*s[0].0, 1)))
})
```

3DF: Dynamic Query Interpretation

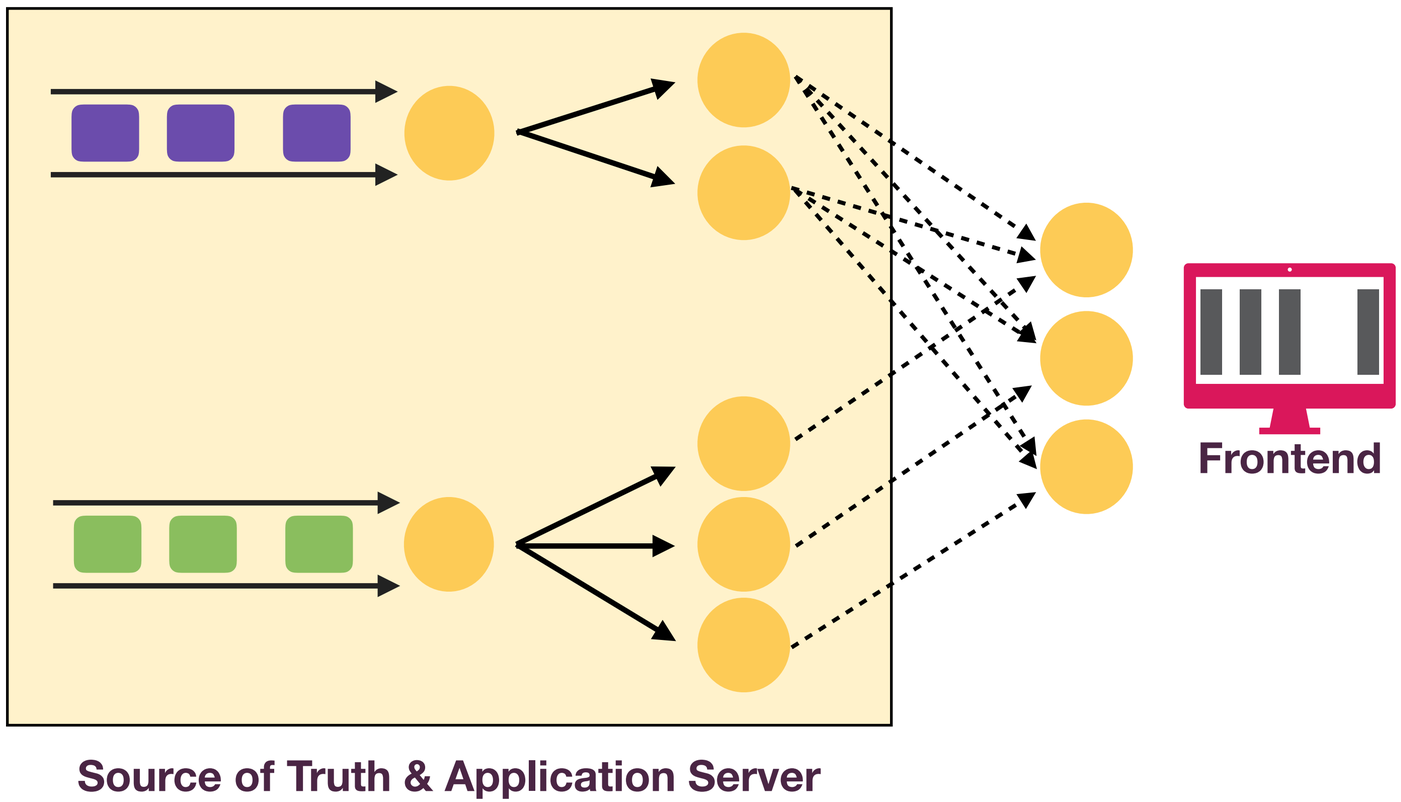
```
to)
to]]

to)
hop]
o)]]
```

```
Plan::Union(Union {
    variables: vec![0, 1],
    plans: vec![
        Box::new(Plan::MatchA(0, ":edge", 1)),
        Box::new(Plan::Project(Project {
            variables: vec![0, 1],
            plan: Box::new(Plan::Join(Join {
                variables: vec![2],
                left_plan: Box::new(Plan::MatchA(0, ":edge", 2)),
                right_plan: Box::new(Plan::NameExpr(NameExpr {
                    variables: vec!["bfs", 2, 1]}))}))}))]})
```

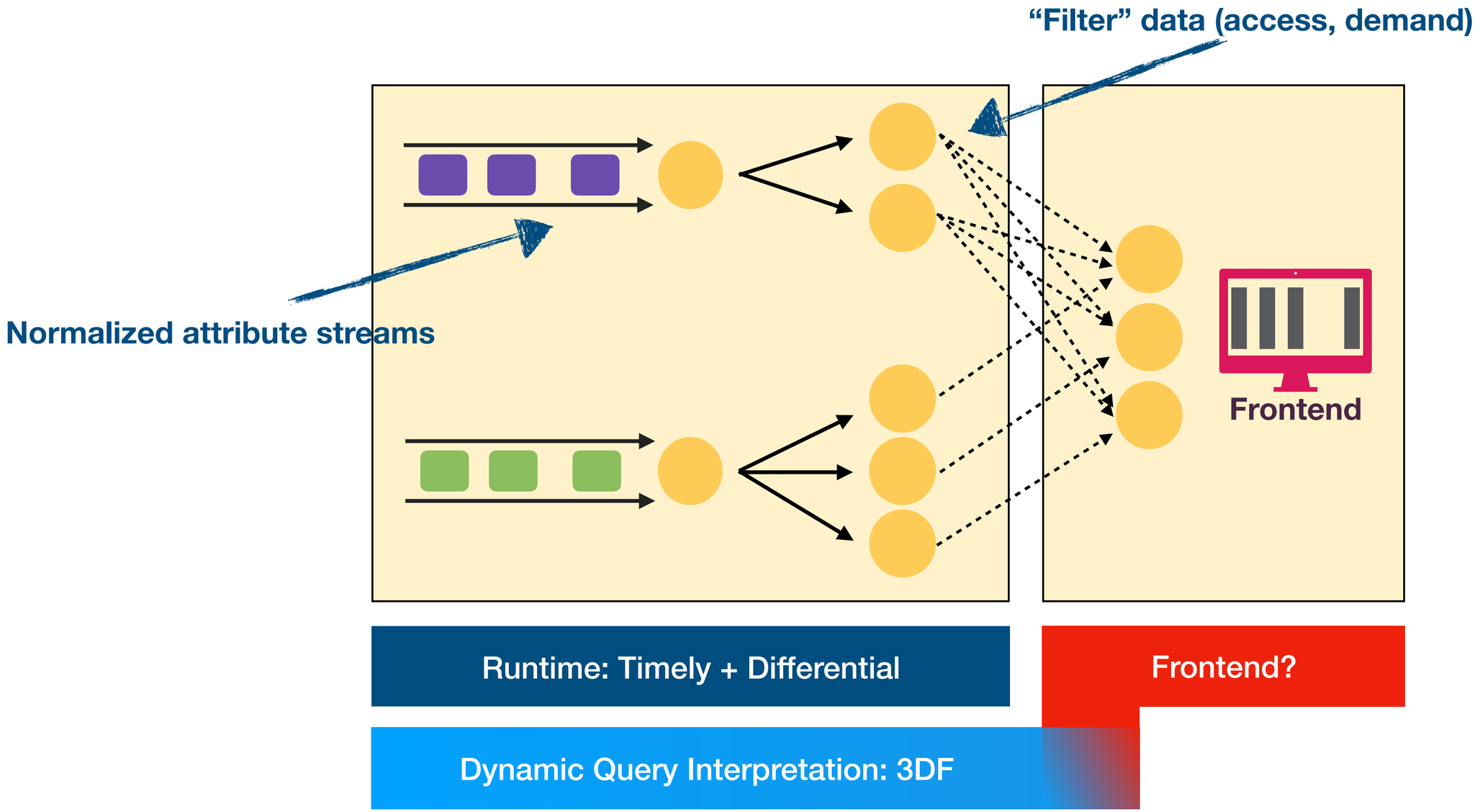
```
/// Frie

let node

nodes.it

 let edg
 let nod

 inner.j
     .con
     .red
})
```

github.com/comnik/**declarative-dataflow**

Recap



Recap

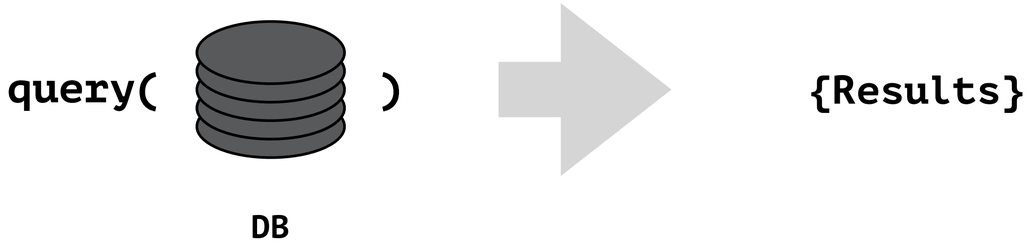

"Filter" data (access, demand)
Normalized attribute streams
Frontend
Runtime: Timely + Differential
Frontend?
Dynamic Query Interpretation: 3DF

Clients Inside the Database

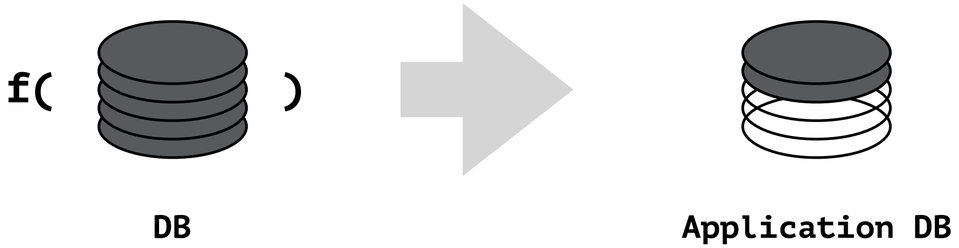
Modeling Frontend State Relationally

State of the application	→	`DB`
Information	→	`Query`
Transition	→	`Transaction`

A Custom Database For Everyone

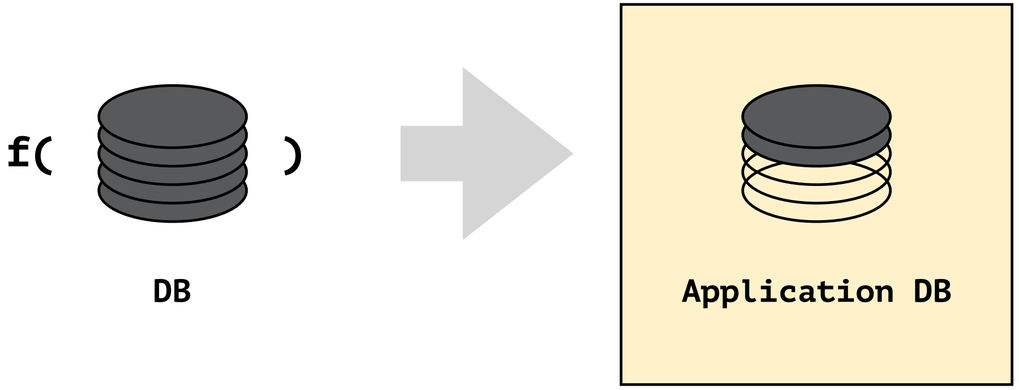


A Custom Database For Everyone



f(DB) → DB

A Custom Database For Everyone



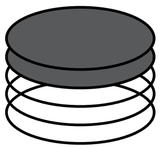

```
[1 :person/name Alice]
[1 :person/age  32]
[1 :person/residence 4]

[4 :residence/city NY]
[4 :residence/country USA]
[4 :residence/zip 10002]
```

f(DB) → DB

github.com/tonsky/**DataScript**

Populating the DB with Facts

```
(query db “attribute-1” …)  ⟶  (e :attribute-1 v)  ⟶
(query db “attribute-2” …)  ⟶  (e :attribute-2 v)  ⟶
(query db “attribute-3” …)  ⟶  (e :attribute-3 v)  ⟶
```



Application DB

Populating the DB with Facts

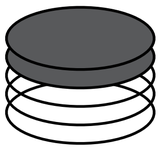
All derived attributes we are interested in

```
(query db "attribute-1" …)
(query db "attribute-2" …)
(query db "attribute-3" …)
```

→

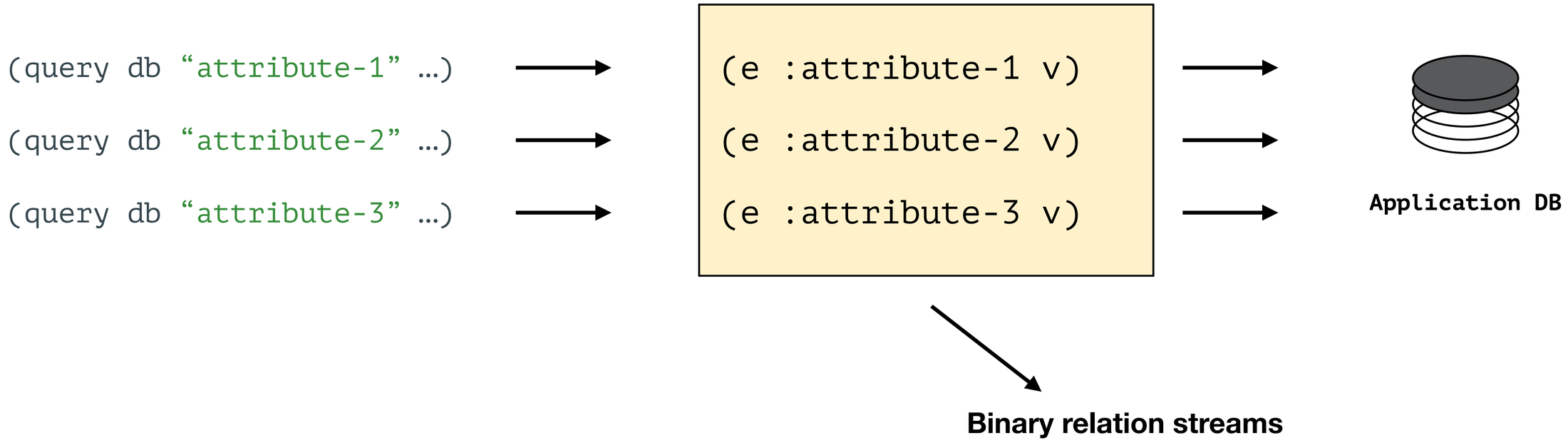
```
(e :attribute-1 v)
(e :attribute-2 v)
(e :attribute-3 v)
```

→



Application DB

Populating the DB with Facts



Example

Fact := (e :message/sender “name”)

entity **attribute** **value**

```
(query db “:message/sender“
       [:find ?msg-id ?sender
        :where
        [?msg-id :message/recipient <current-userid>]
        [?msg-id :message/sender ?sender-id]
        [?sender-id :user/name ?sender]])
```

Application DB

Example

Fact := (e :message/sender “name”)

entity **attribute** **value**

```
(query db ":message/sender"
      [:find ?msg-id ?sender
       :where
       [?msg-id :message/recipient <current-userid>]
       [?msg-id :message/sender ?sender-id]
       [?sender-id :user/name ?sender]])
```

Application DB

Example

```
Fact := (e :message/sender "name")
```

entity **attribute** **value**

```
(query db ":message/sender"
      [:find ?msg-id ?sender
       :where
       [?msg-id :message/recipient <current-userid>]
       [?msg-id :message/sender ?sender-id]
       [?sender-id :user/name ?sender]])
```

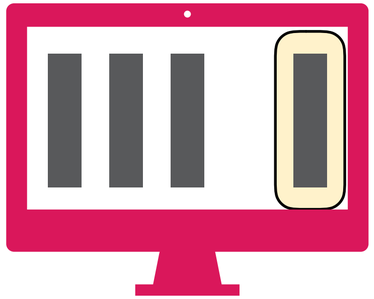
Application DB

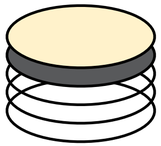
Example

Fact := (e :message/sender “name”)

entity **attribute** **value**

```
(query db ":message/sender"
        [:find ?msg-id ?sender
         :where
         [?msg-id :message/recipient <current-userid>]
         [?msg-id :message/sender ?sender-id]
         [?sender-id :user/name ?sender]])
```





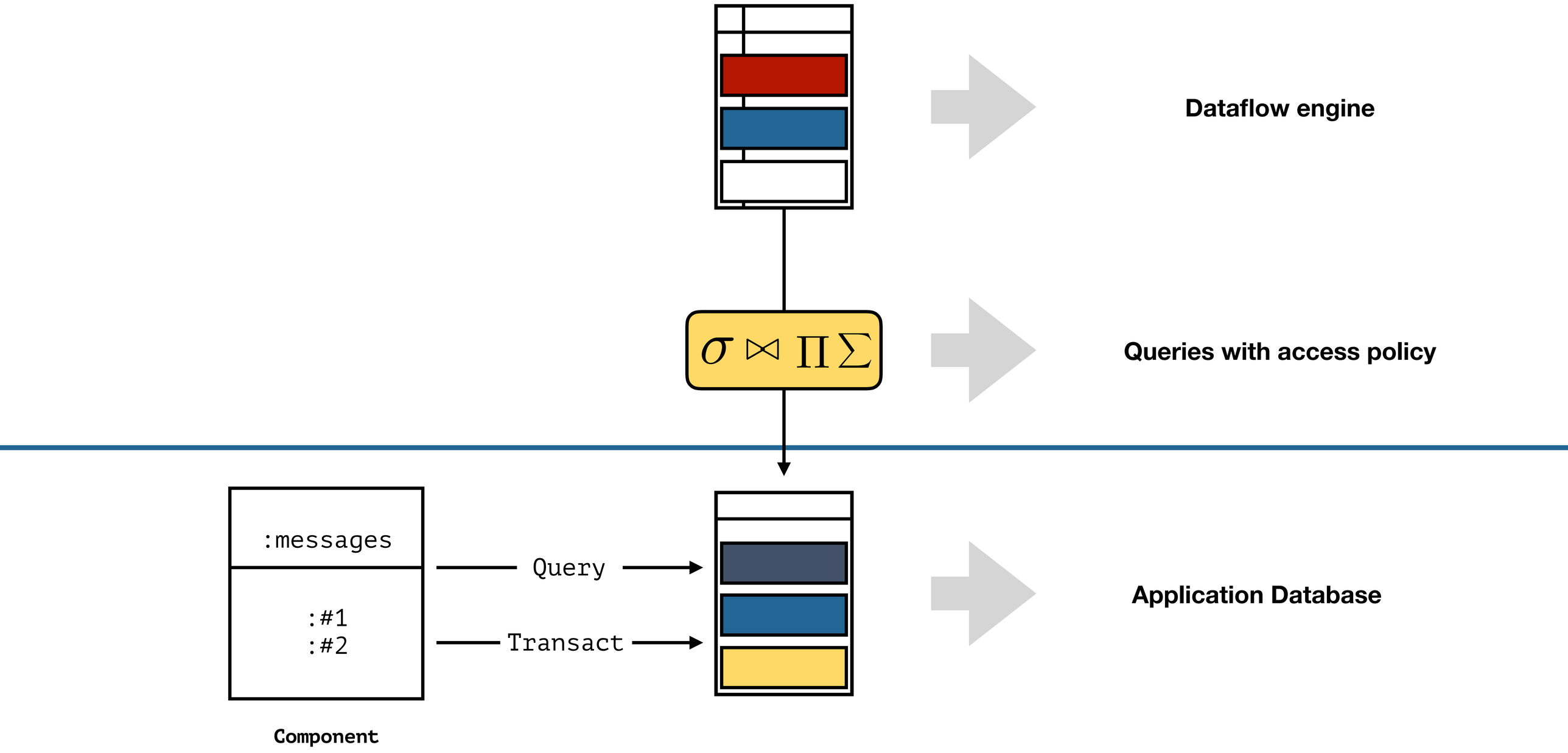
Application DB

Populating the DB with Facts

```
(query db "attribute-1" …)  →  (e :attribute-1 v)  →
(query db "attribute-2" …)  →  (e :attribute-2 v)  →
(query db "attribute-3" …)  →  (e :attribute-3 v)  →
```

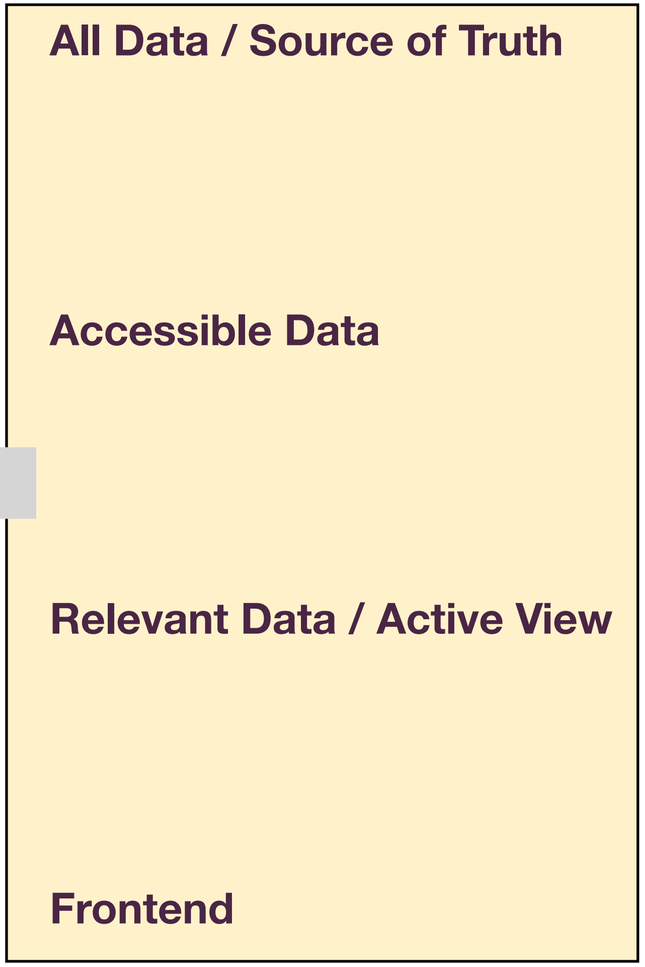
Application DB

```
(pull db :attribute-1
         :attribute-2
         :attribute-3)
```


Dataflow engine
$\sigma \bowtie \Pi \Sigma$
Queries with access policy
:messages
:#1
:#2
Query
Transact
Application Database
Component

One Query to Rule Them All

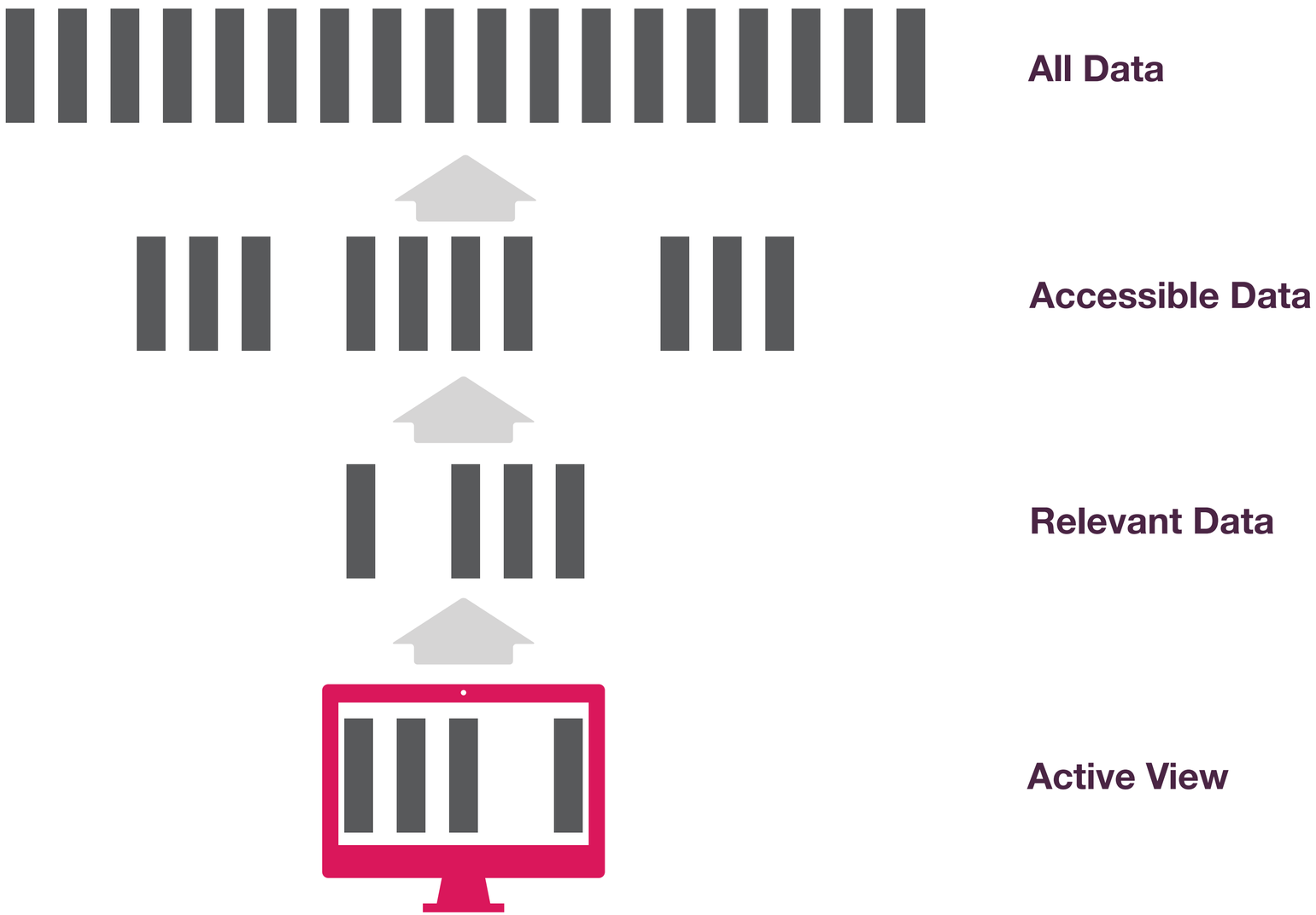
```
(query db ":message/sender"
       [:find ?msg-id ?sender
        :where
        [?msg-id :message/recipient <current-userid>]
        [?msg-id :message/sender ?sender-id]
        [?sender-id :user/name ?sender]
        (access? <current-userid>)])
```



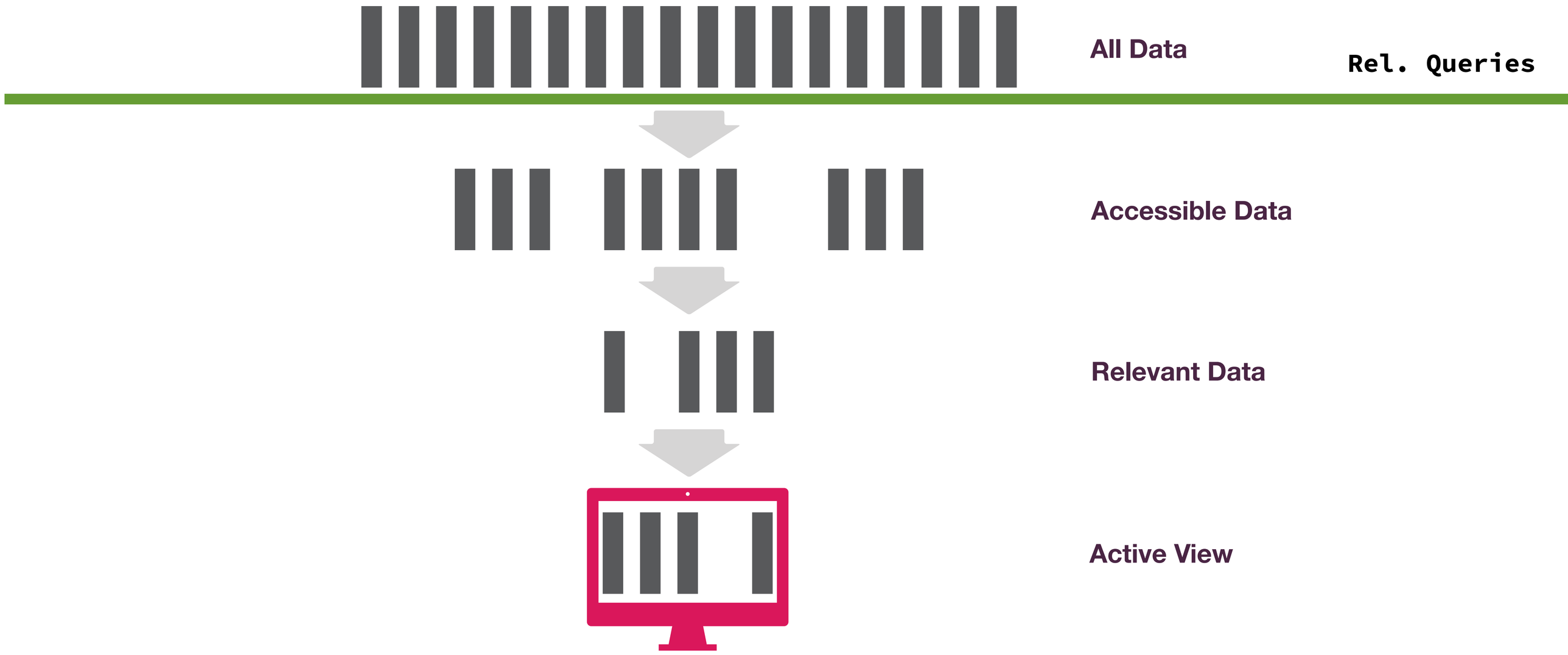
Future Work

- 3DF as client-side database via WebAssembly
  github.com/comnik/functional-differential-programming
- Use `Plan` struct to support additional query languages (SQL, GraphQL)

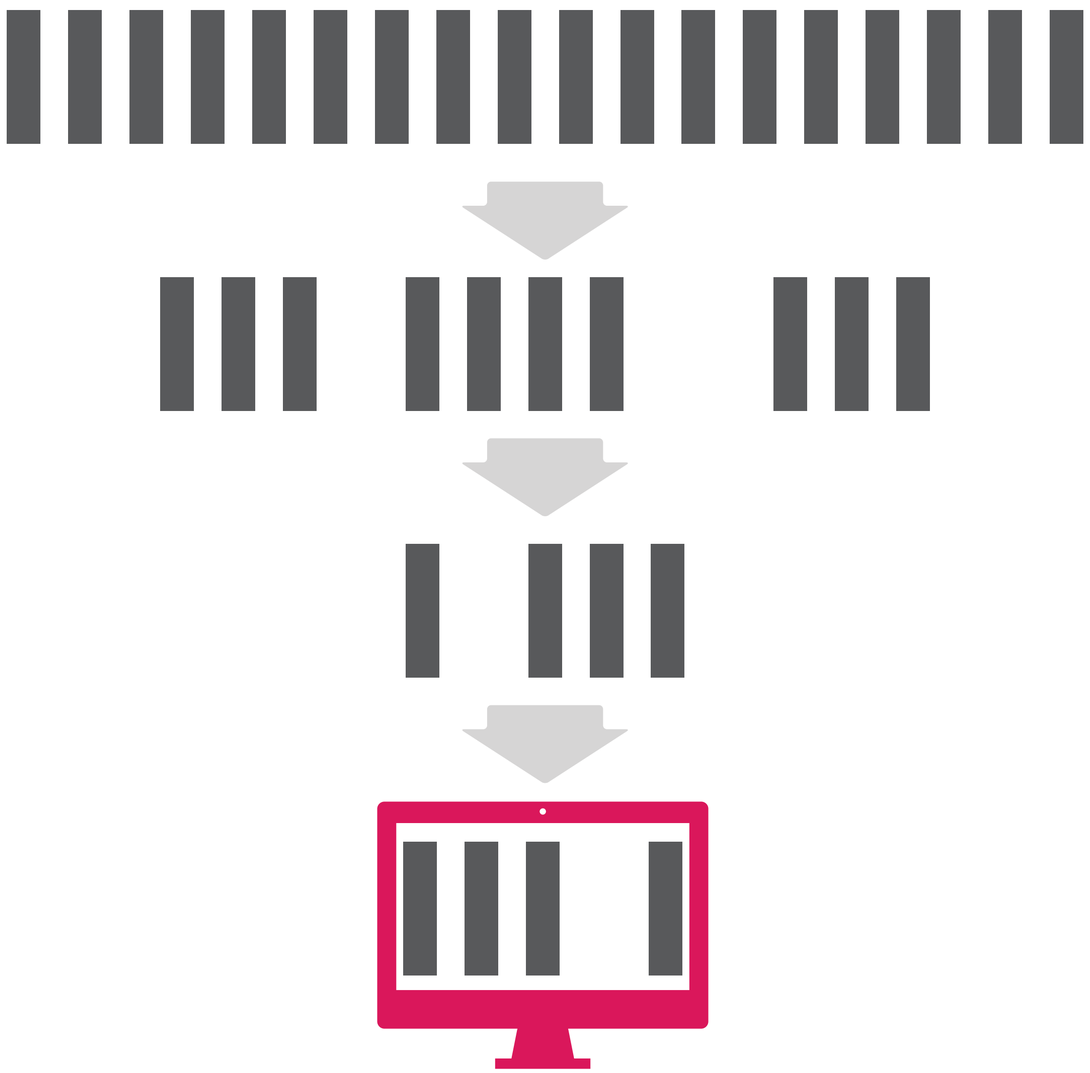
Right Inside the Database



Right Inside the Database



Right Inside the Database



All Data

Accessible Data

Relevant Data

Active View