Runaway complexity in Big Data

And a plan to stop it

Nathan Marz
@nathanmarz

Agenda

- Common sources of complexity in data systems
- Design for a fundamentally better data system

What is a data system?

A system that manages the **storage** and **querying** of data

What is a data system?

A system that manages the **storage** and **querying** of data with a lifetime measured in **years**

What is a data system?

A system that manages the **storage** and **querying** of data with a lifetime measured in **years** encompassing every **version** of the application to ever exist

What is a data system?

A system that manages the **storage** and **querying** of data with a lifetime measured in **years** encompassing every **version** of the application to ever exist, every **hardware failure**

What is a data system?

A system that manages the **storage** and **querying** of data with a lifetime measured in **years** encompassing every **version** of the application to ever exist, every **hardware failure**, and every **human mistake** ever made

Common sources of complexity

Lack of human fault-tolerance

Conflation of data and queries

Schemas done wrong

Lack of human fault-tolerance

Human fault-tolerance

- Bugs will be deployed to production over the lifetime of a data system
- Operational mistakes will be made
- Humans are part of the overall system, just like your hard disks, CPUs, memory, and software
- Must design for human error like you'd design for any other fault

Human fault-tolerance

Examples of human error

- Deploy a bug that increments counters by two instead of by one
- Accidentally delete data from database
- Accidental DOS on important internal service

The worst consequence is data loss or data corruption

As long as an error doesn't lose or corrupt good data, you can fix what went wrong

Mutability

- The U and D in CRUD
- A mutable system updates the current state of the world
- Mutable systems inherently lack human fault-tolerance
- Easy to corrupt or lose data

Immutability

- An immutable system captures a historical record of events
- Each **event** happens at **a particular time** and is **always true**

Capturing change with mutable data model

Person	Location
Sally	Philadelphia
Bob	Chicago

Person	Location
Sally	New York
Bob	Chicago

Sally moves to New York

Capturing change with immutable data model

Person	Location	Time
Sally	Philadelphia	1318358351
Bob	Chicago	1327928370

Person	Location	Time
Sally	Philadelphia	1318358351
Bob	Chicago	1327928370
Sally	New York	1338469380

Sally moves to New York

Immutability greatly restricts the range of errors that can cause data loss or data corruption

Vastly more human fault-tolerant

Immutability

Other benefits

- Fundamentally simpler
- CR instead of CRUD
- Only write operation is appending new units of data
- Easy to implement on top of a distributed filesystem
  - File = list of data records
  - Append = Add a new file into a directory

basing a system on mutability is like pouring gasoline on your house (but don't worry, i checked all the wires carefully to make sure there won't be any sparks). when someone makes a mistake who knows what will burn

Immutability

Please watch Rich Hickey's talks to learn more about the enormous benefits of immutability

Conflation of data and queries

Conflation of data and queries

Normalization vs. denormalization

ID	Name	Location ID
1	Sally	3
2	George	1
3	Bob	3

Location ID	City	State	Population
1	New York	NY	8.2M
2	San Diego	CA	1.3M
3	Chicago	IL	2.7M

Normalized schema

Join is too expensive, so denormalize...

ID	Name	Location ID	City	State
1	Sally	3	Chicago	IL
2	George	1	New York	NY
3	Bob	3	Chicago	IL

Location ID	City	State	Population
1	New York	NY	8.2M
2	San Diego	CA	1.3M
3	Chicago	IL	2.7M

Denormalized schema

Obviously, you prefer all data to be **fully normalized**

But you are forced to **denormalize** for performance

Because the way data is modeled, stored, and queried is complected

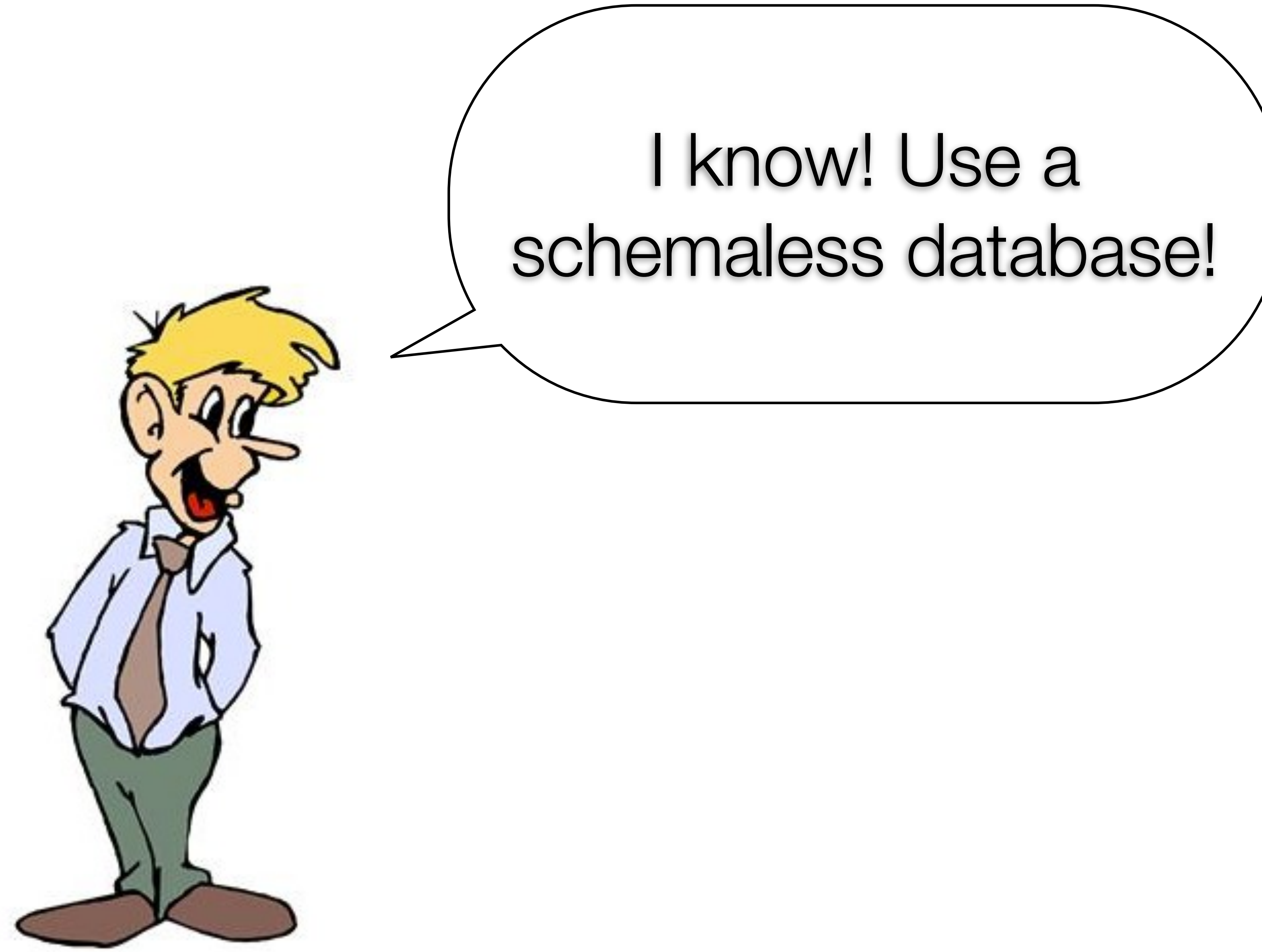
We will come back to how to build data systems in which these are disassociated

Schemas done wrong

Schemas have a bad rap

Schemas

- Hard to change
- Get in the way
- Add development overhead
- Requires annoying configuration


I know! Use a schemaless database!

This is an overreaction

Confuses the poor implementation of schemas with the value that schemas provide

What is a schema exactly?

function(data unit)

That says whether this data is valid or not

This is useful

Value of schemas

- Structural integrity
- Guarantees on what can and can’t be stored
- Prevents corruption

Otherwise you'll detect corruption issues at read-time

Potentially long after the corruption happened

With little insight into the circumstances of the corruption

Much better to get an exception where the mistake is made, before it corrupts the database

Saves enormous amounts of time

Why are schemas considered painful?

- Changing the schema is hard (e.g., adding a column to a table)
- Schema is overly restrictive (e.g., cannot do nested objects)
- Require translation layers (e.g. ORM)
- Requires more typing (development overhead)

None of these are fundamentally linked with function(data unit)

These are problems in the implementation of schemas, not in schemas themselves

Ideal schema tool

- Data is represented as maps
- Schema tool is a library that helps construct the schema function:
  - Concisely specify required fields and types
  - Insert custom validation logic for fields (e.g. ages are between 0 and 200)
- Built-in support for evolving the schema over time
- Fast and space-efficient serialization/deserialization
- Cross-language

this is easy to use and gets out of your way

i use apache thrift, but it lacks the custom validation logic

i think it could be done better with a clojure-like data as maps approach

given that parameters of a data system: long-lived, ever changing, with mistakes being made, the amount of work it takes to make a schema (not that much) is absolutely worth it

Let's get provocative

The relational database will be a footnote in history

Not because of SQL, restrictive schemas, or scalability issues

But because of fundamental flaws in the RDBMS approach to managing data

Mutability

Conflating the storage of data with how it is queried

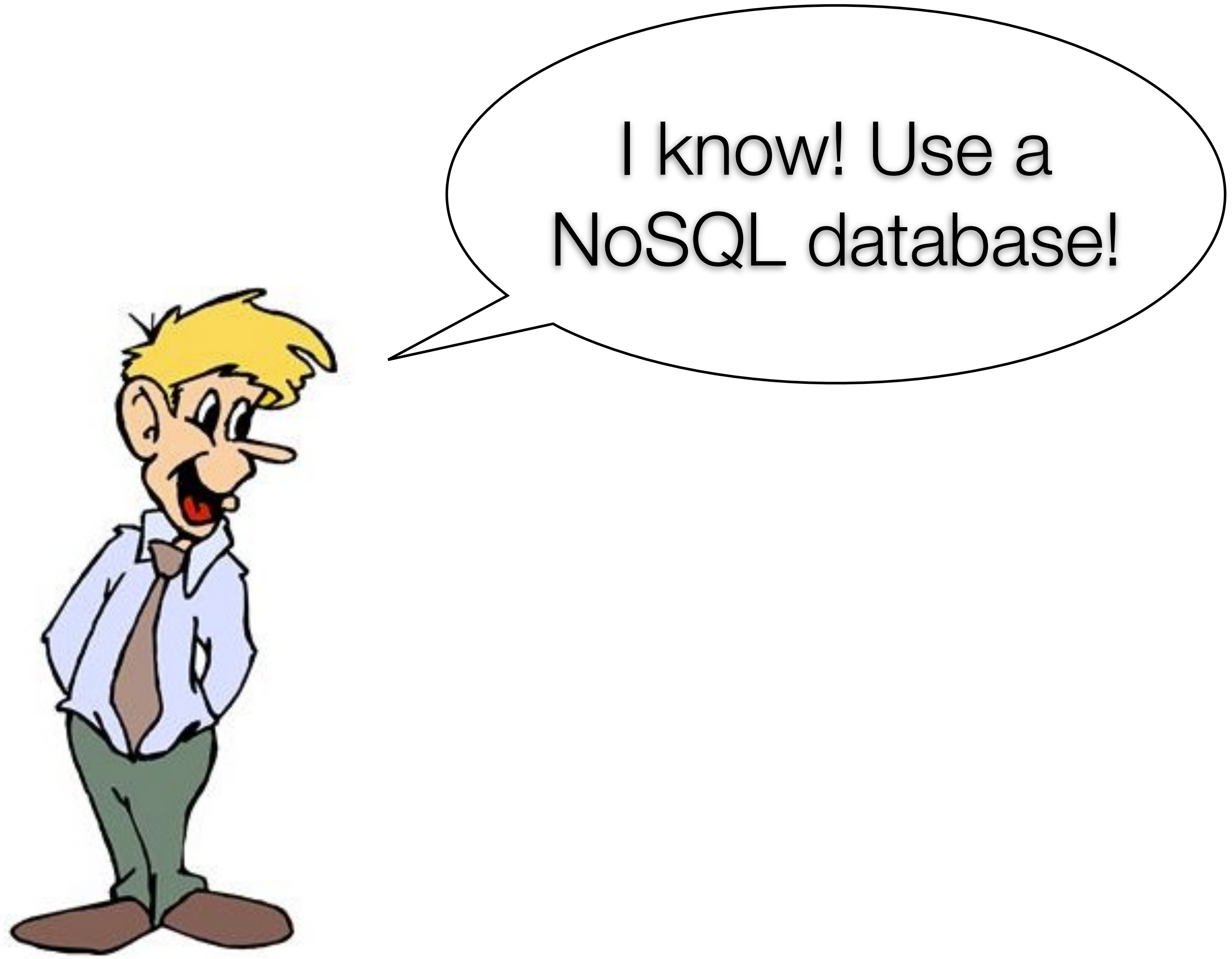
Back in the day, these flaws were feature - because space was a premium. The landscape has changed, and this is no longer the constraint it once was. So these properties of mutability and conflating data and queries are now major, glaring flaws. Because there are better ways to design data system

“NewSQL” is misguided

Let's use our ability to cheaply store massive amounts of data

To do data right

And not inherit the complexities of the past



if SQL's wrong, and NoSQL isn't SQL, then NoSQL must be right

NoSQL databases are generally not a step in the right direction

Some aspects are, but not the ones that get all the attention

Still based on mutability and not general purpose

Let's see how you design a data system that doesn't suffer from these complexities

Let's start from scratch

What does a data system do?

Retrieve data that you previously stored?

Put

Get

Not really...

Counterexamples

Store location information on people

How many people live in a particular location?

Where does Sally live?

What are the most populous locations?

Counterexamples

Store pageview information

How many pageviews on September 2nd?

How many unique visitors over time?

Counterexamples

Store transaction history for bank account

How much money does George have?

How much money do people spend on housing?

What does a data system do?

Query = Function(All data)

Sometimes you retrieve what you stored

Oftentimes you do transformations, aggregations, etc.

Queries as pure functions that take all data as input is the most general formulation

Example query

Total number of pageviews to a URL over a range of time

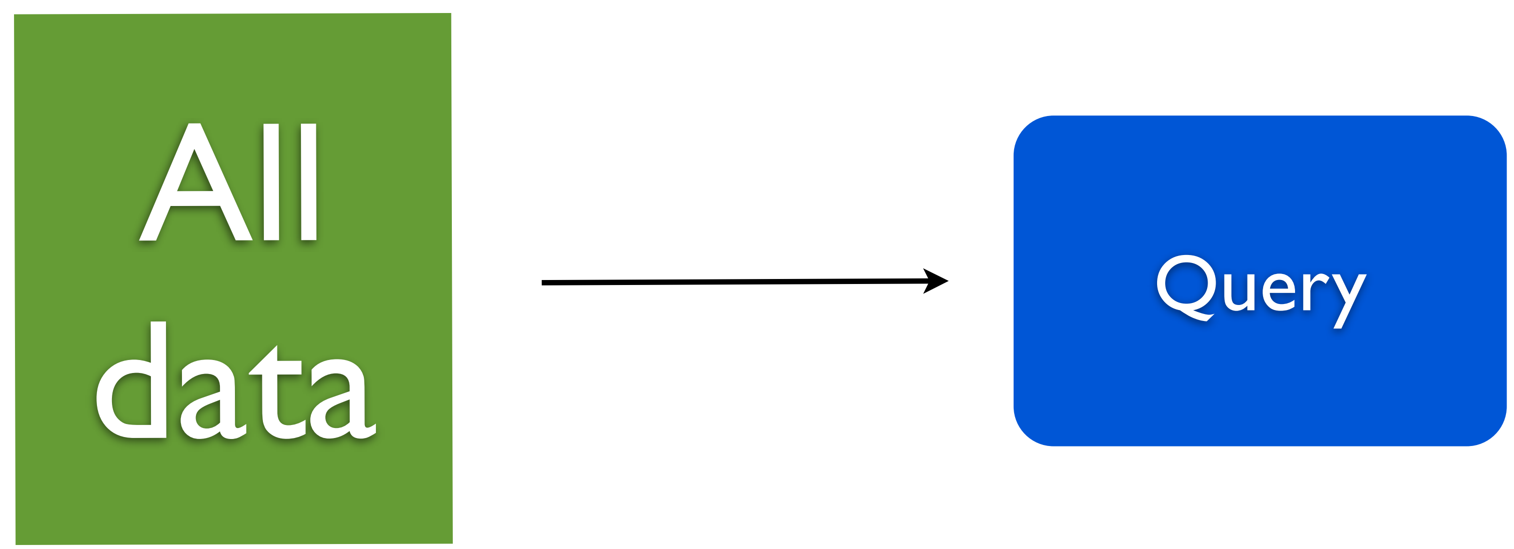
Example query

```
function pageviewsOverTime(allData, url, start, end) {
    count = 0
    for(data: allData) {
        if(data.url == url &&
            data.timestamp >= start &&
            data.timestamp <= end) {
             count++
        }
    }
    return count
}
```

Implementation

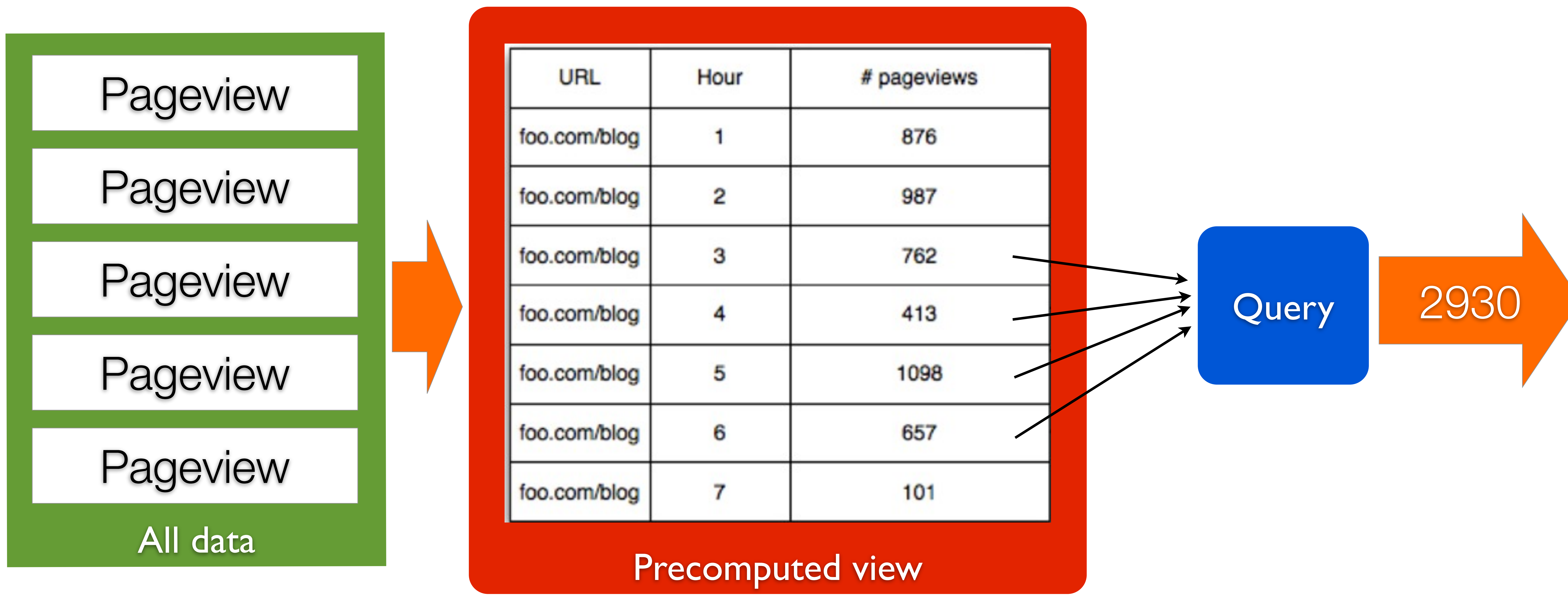
Too slow: “all data” is petabyte-scale

On-the-fly computation



Precomputation

Example query



Precomputation

Precomputation

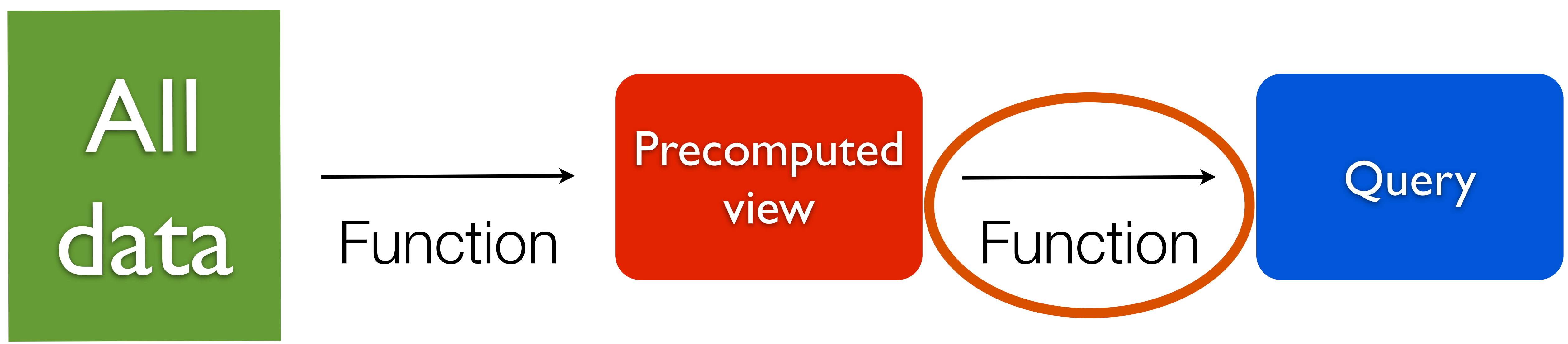
Data system

Two problems to solve

Data system

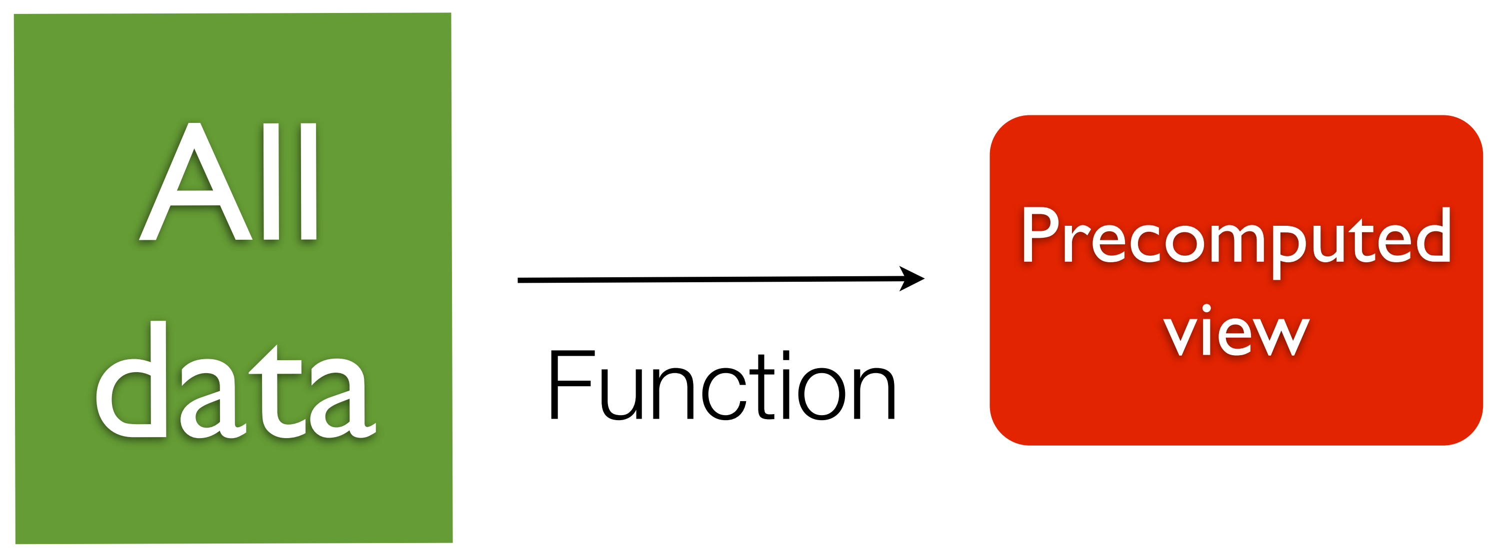
How to compute views

Data system



How to compute queries from views

Computing views



Function that takes in
all data as input

Batch processing

MapReduce

MapReduce is a framework for computing arbitrary functions on arbitrary data

Expressing those functions

cascading

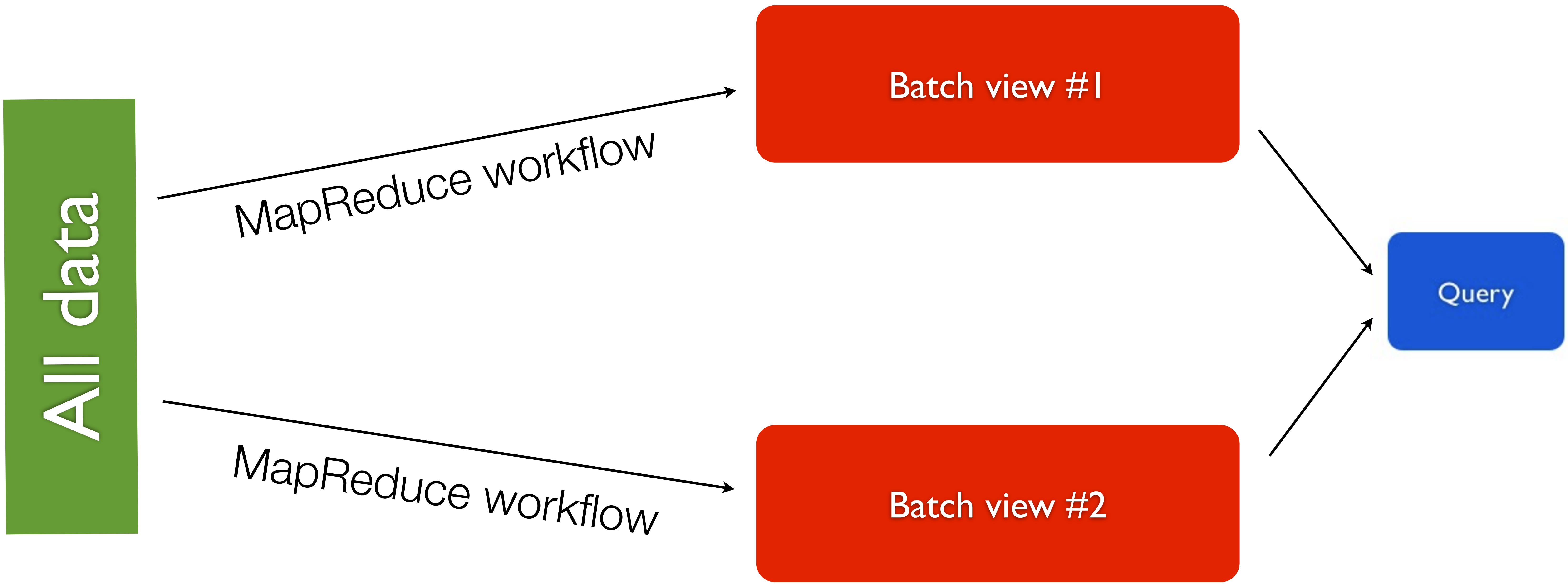
Pig

Cascalog

HIVE

Scalding

MapReduce precomputation



Batch view database

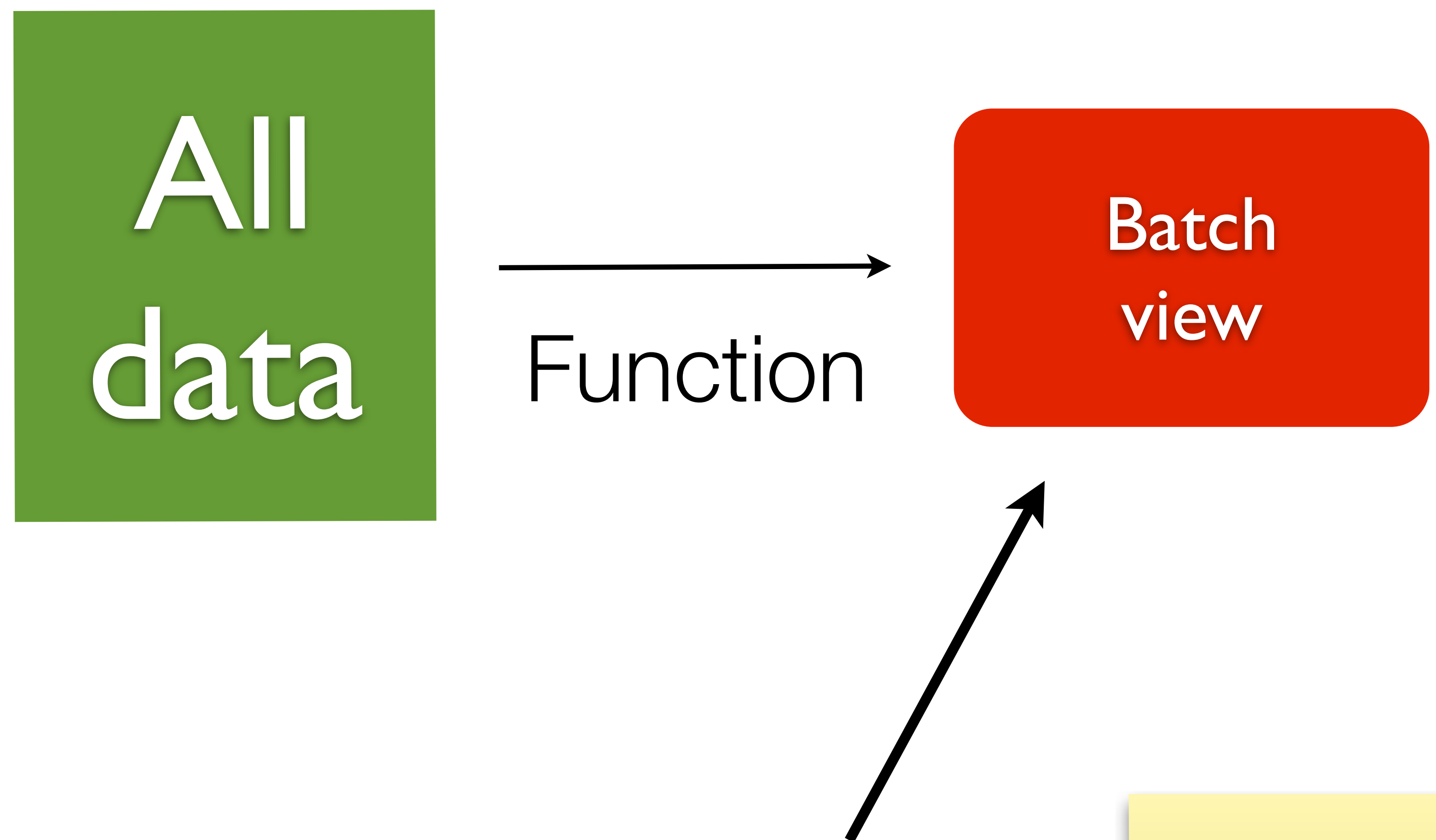
Need a database that...

- Is batch-writable from MapReduce
- Has fast random reads
- Examples: ElephantDB, Voldemort

Batch view database

No random writes required!

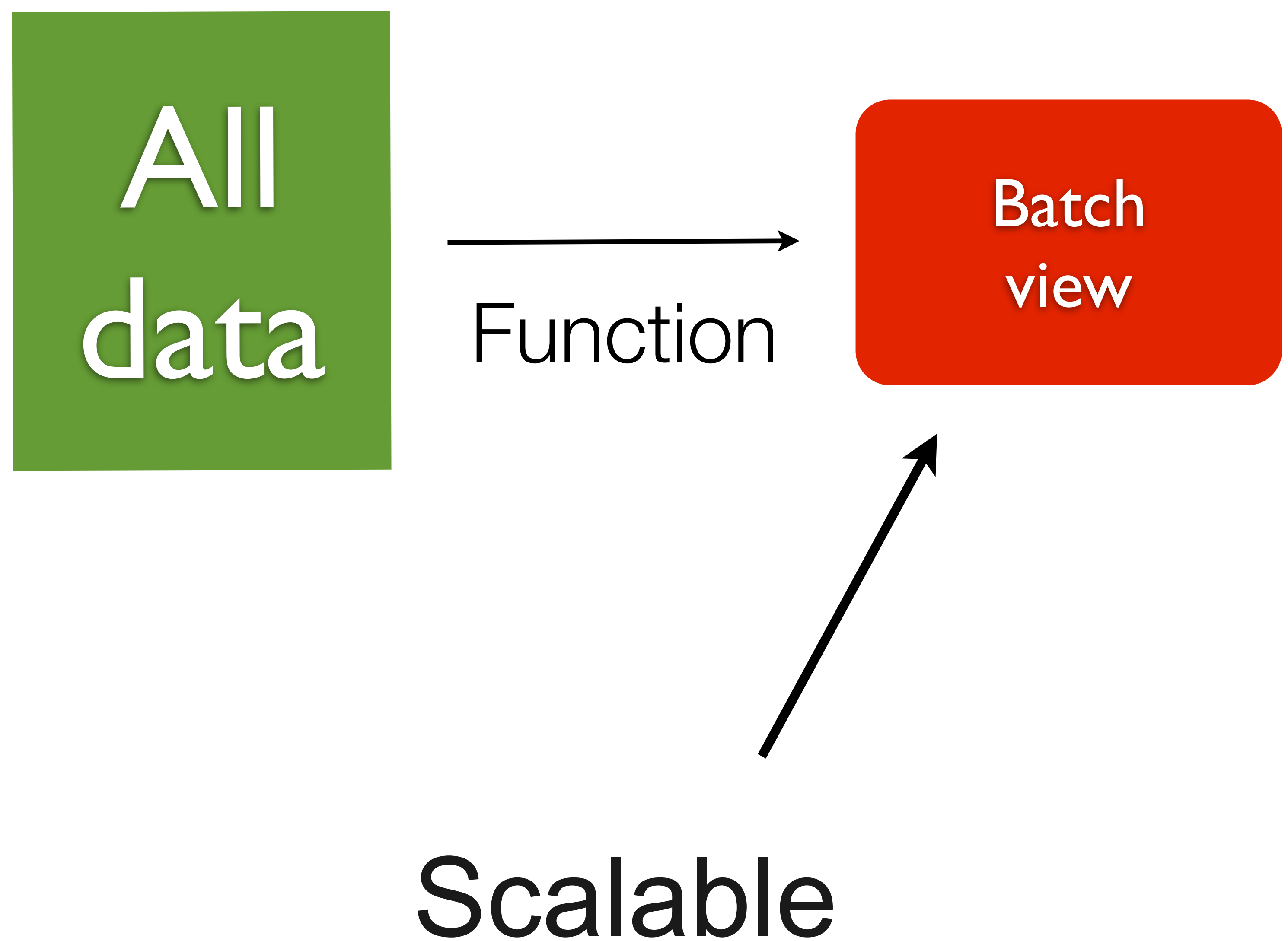
Properties



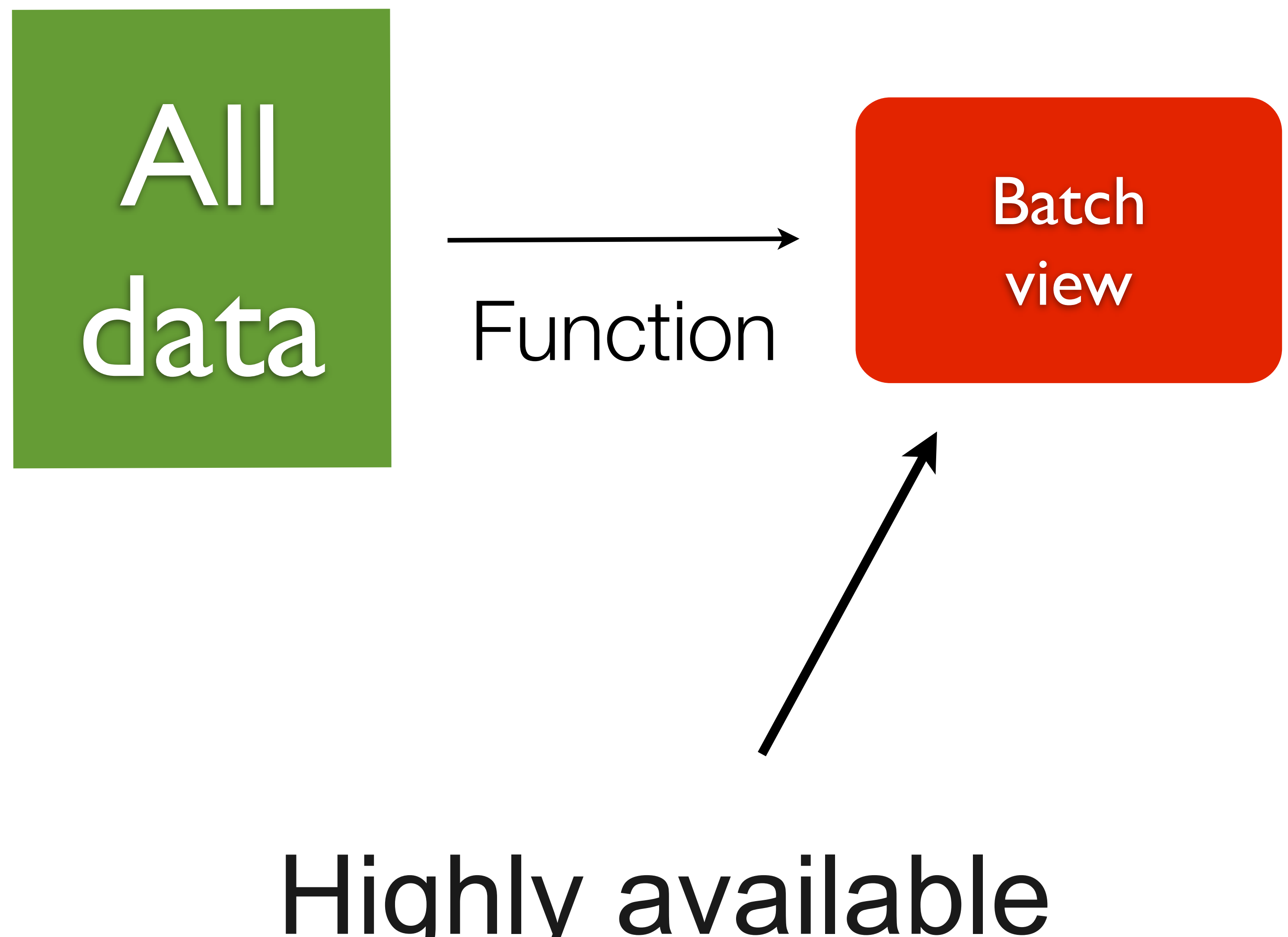
Simple

ElephantDB is only a few thousand lines of code

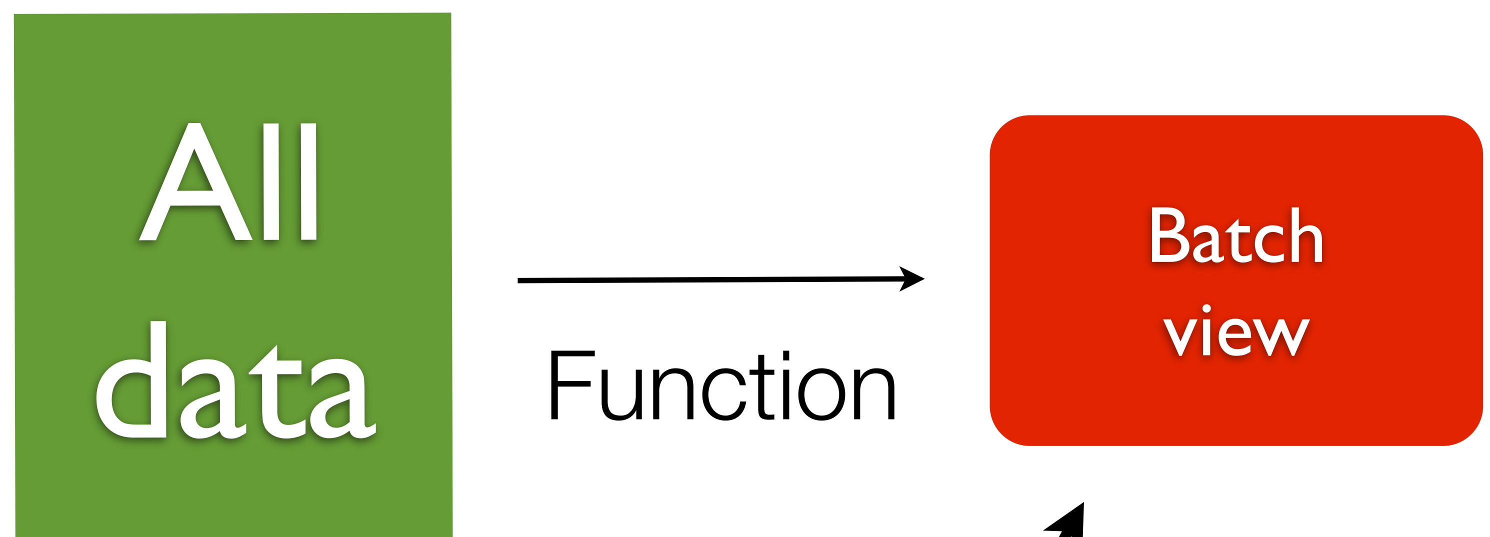
Properties



Properties

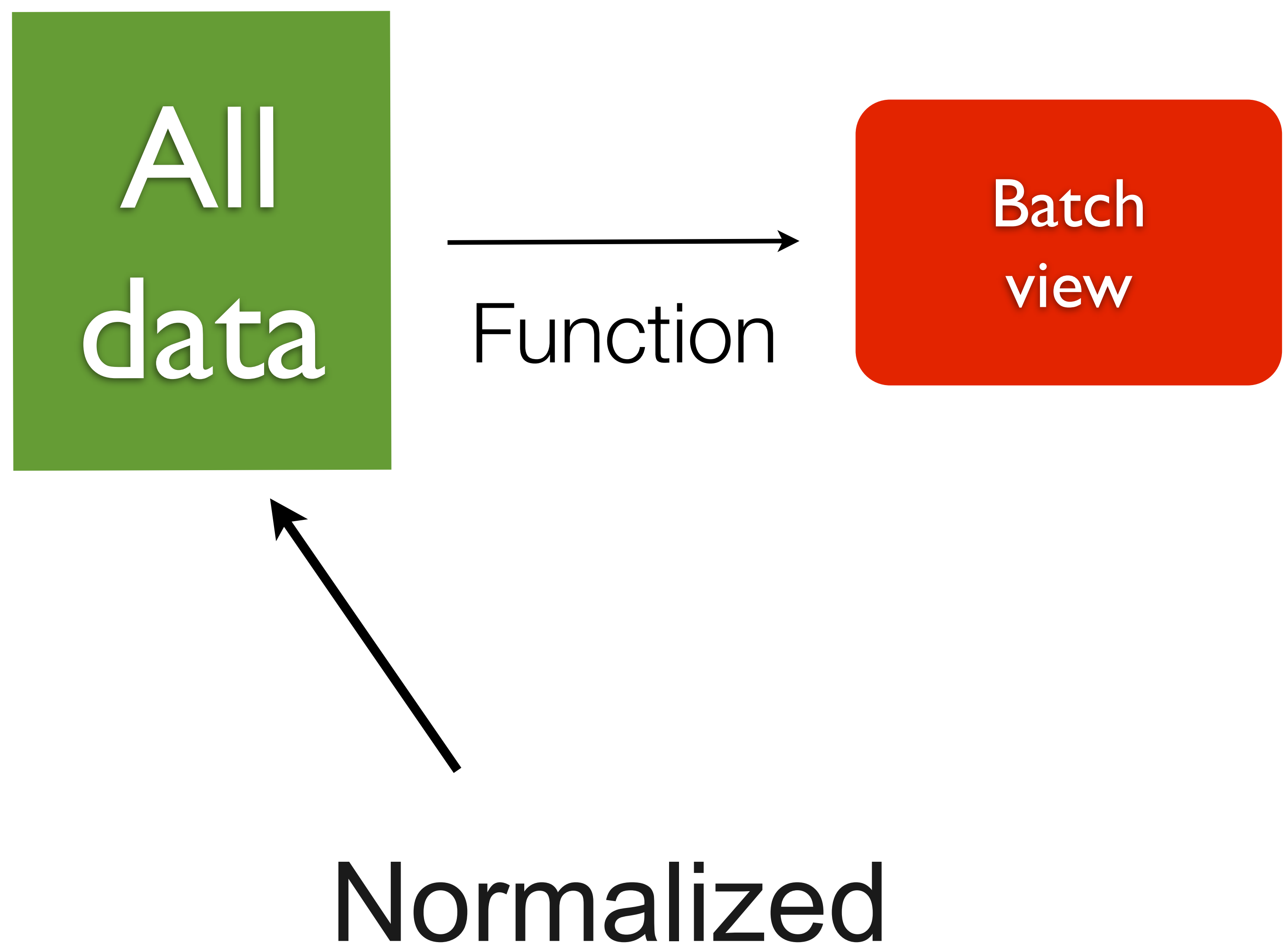


Properties



Can be heavily optimized (b/c no random writes)

Properties



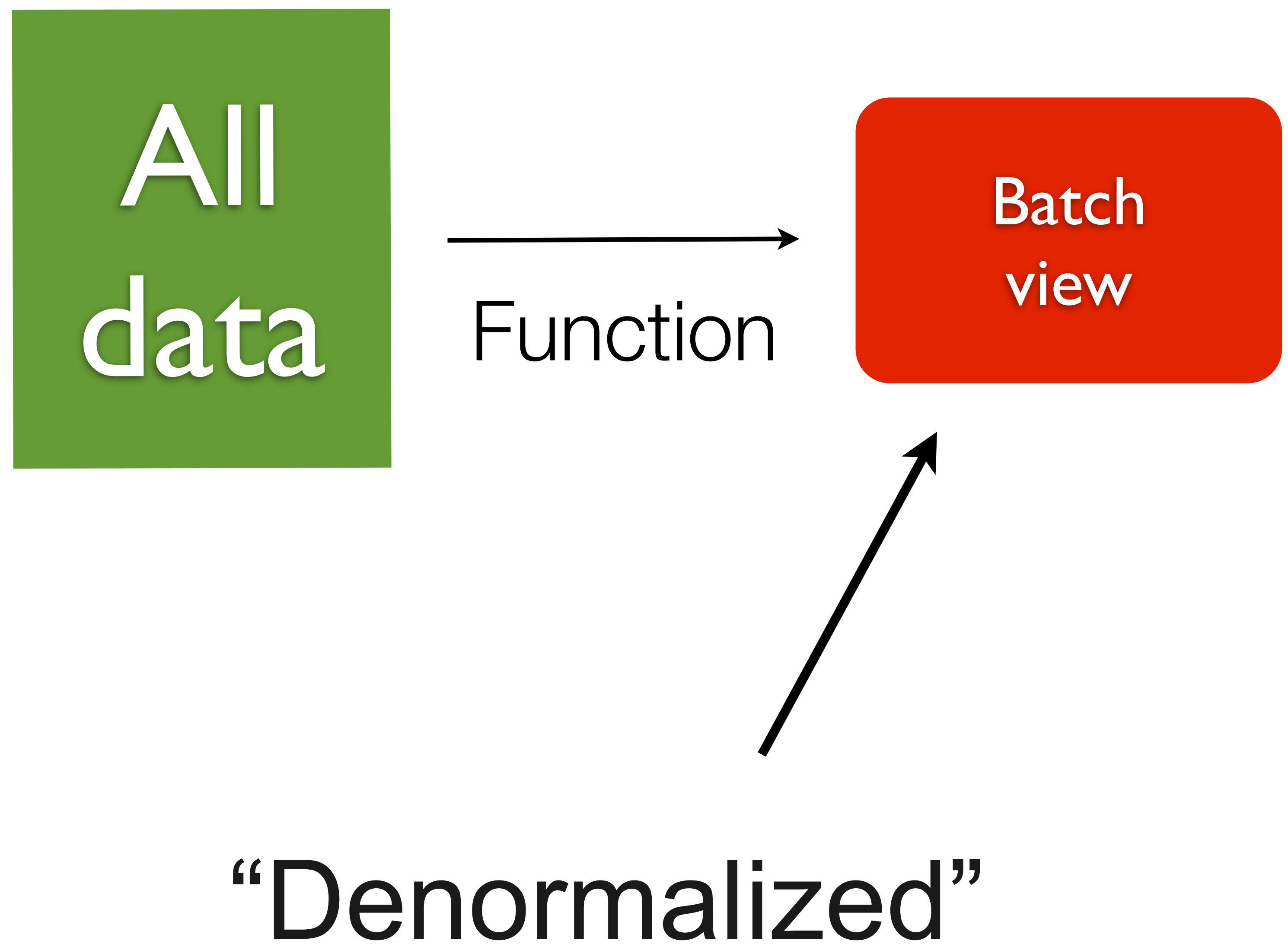
Properties

Not exactly denormalization, because you're doing more than just retrieving data that you stored (can do aggregations)

You're able to optimize data storage separately from data modeling, without the complexity typical of denormalization in relational databases

This is because the batch view is a pure function of all data -> hard to get out of sync, and if there's ever a problem (like a bug in your code that computes the wrong batch view) you can recompute

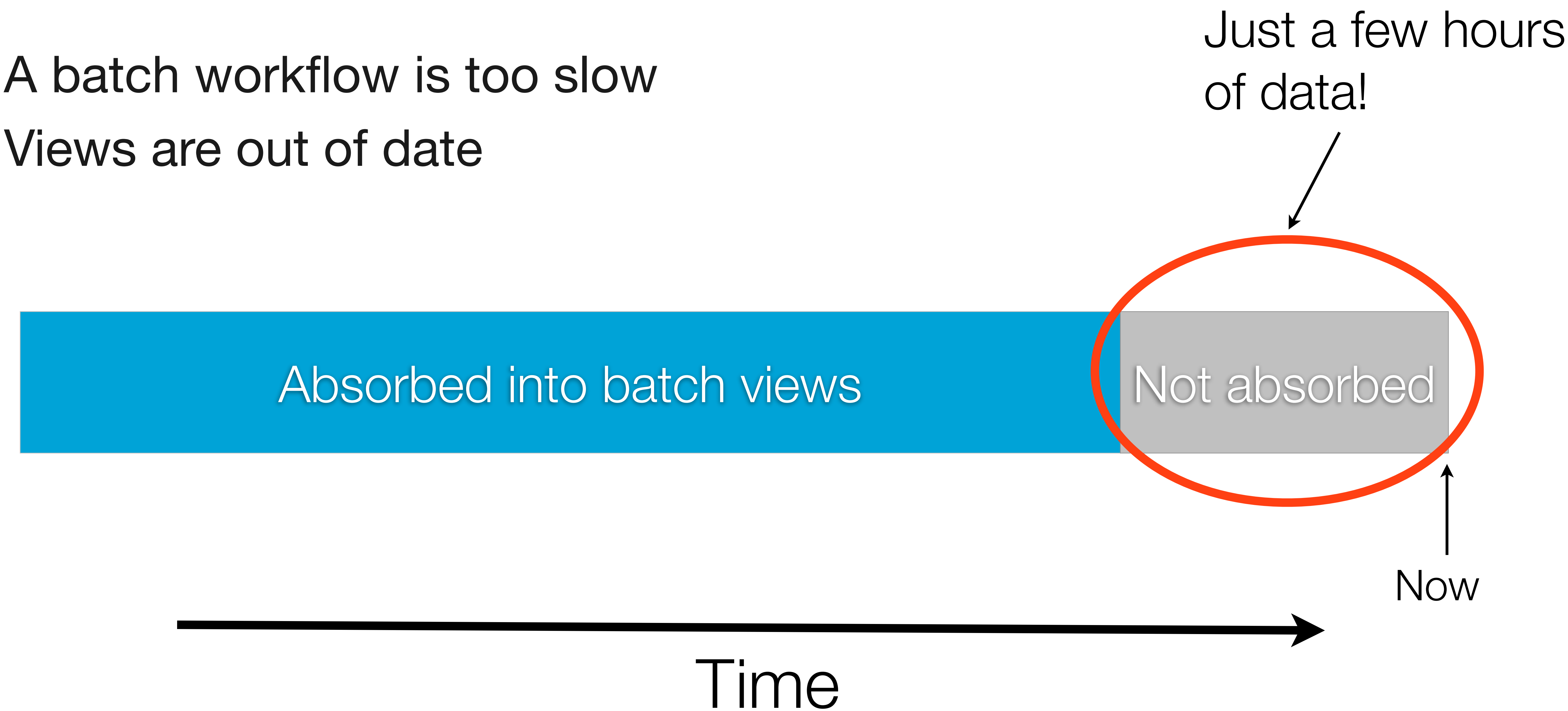
also easy to debug problems, since you have the input that produced the batch view -> this is not true in a mutable system based on incremental updates



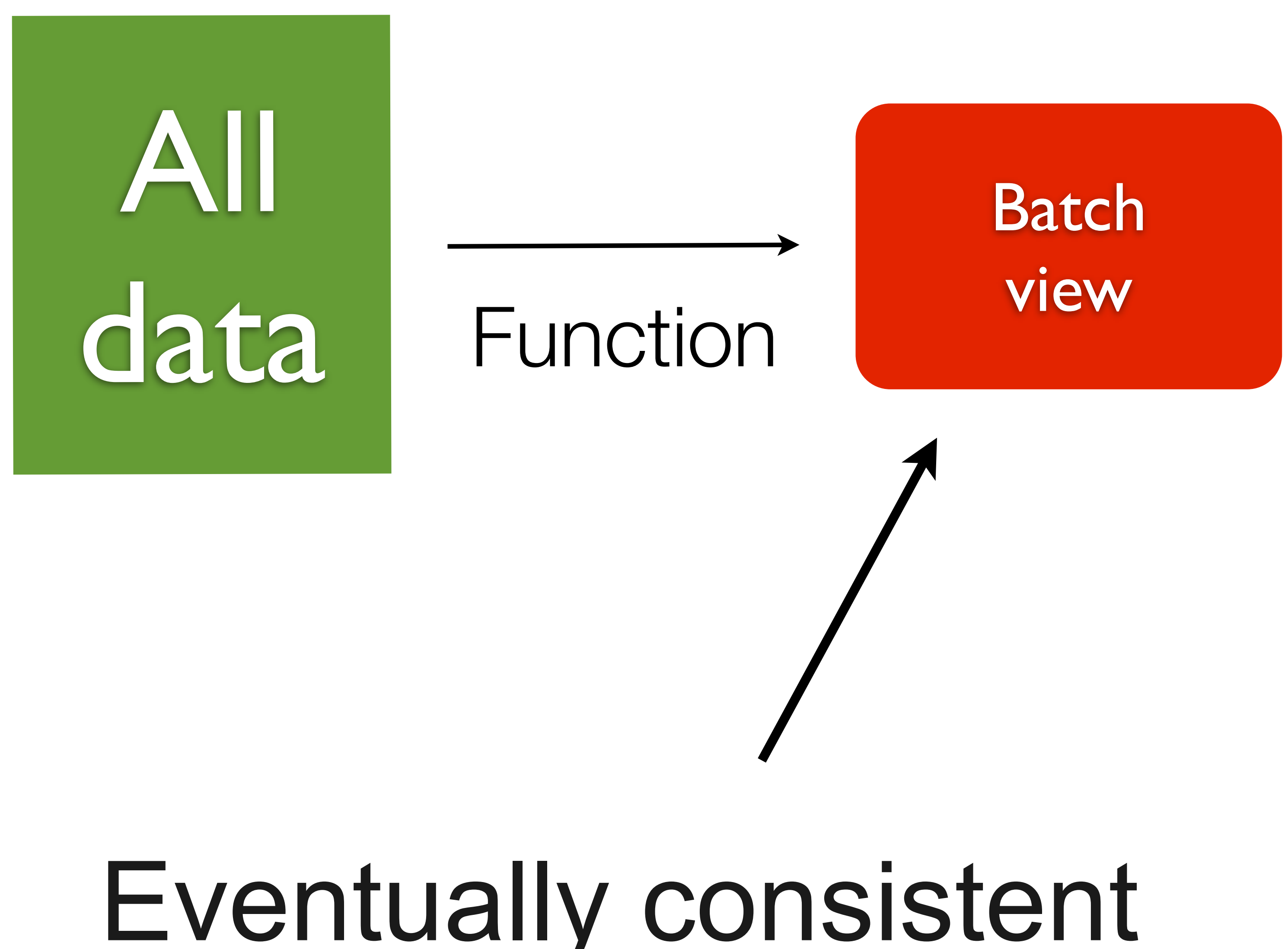
So we’re done, right?

Not quite...

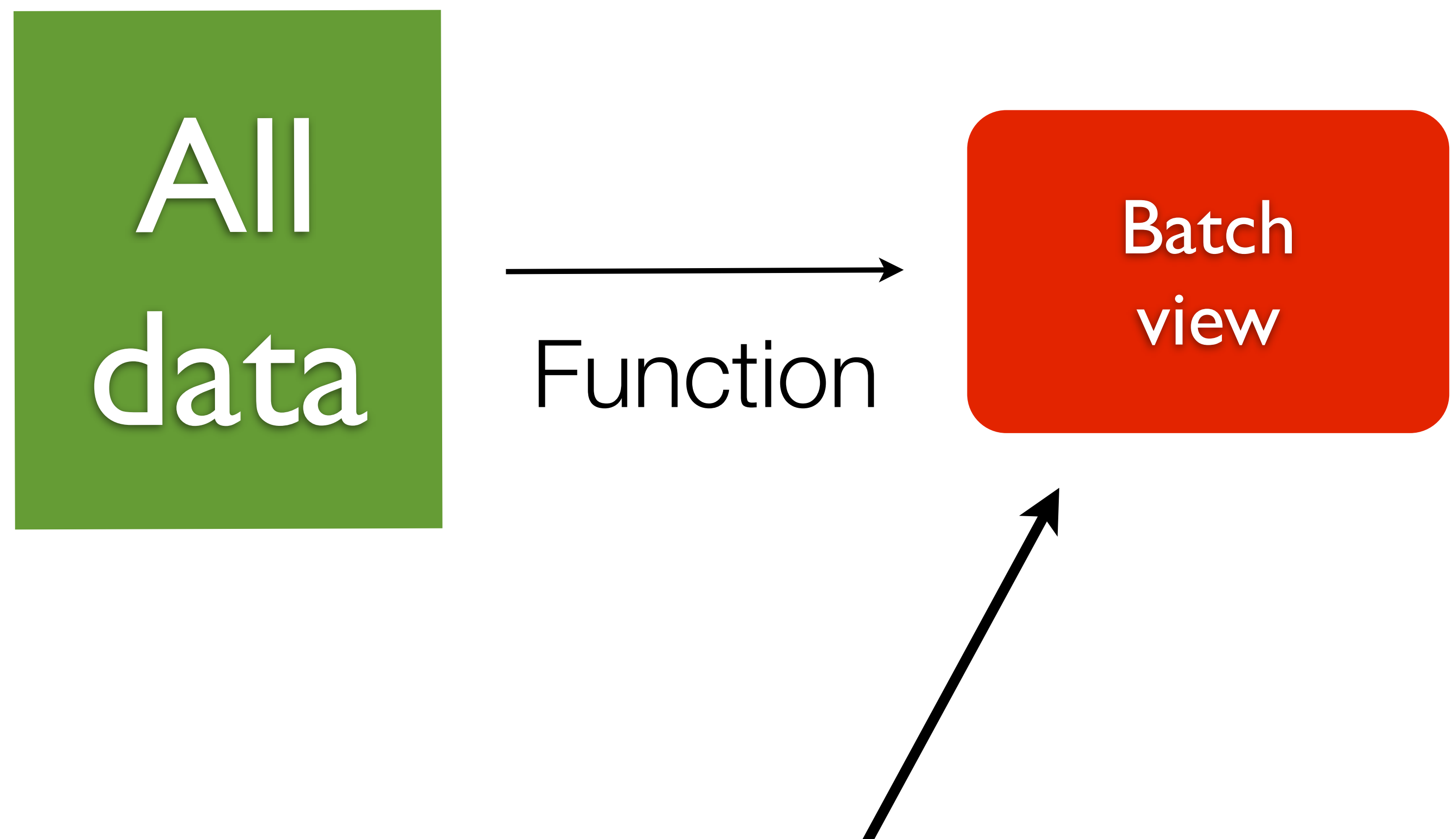
- A batch workflow is too slow
- Views are out of date



Properties

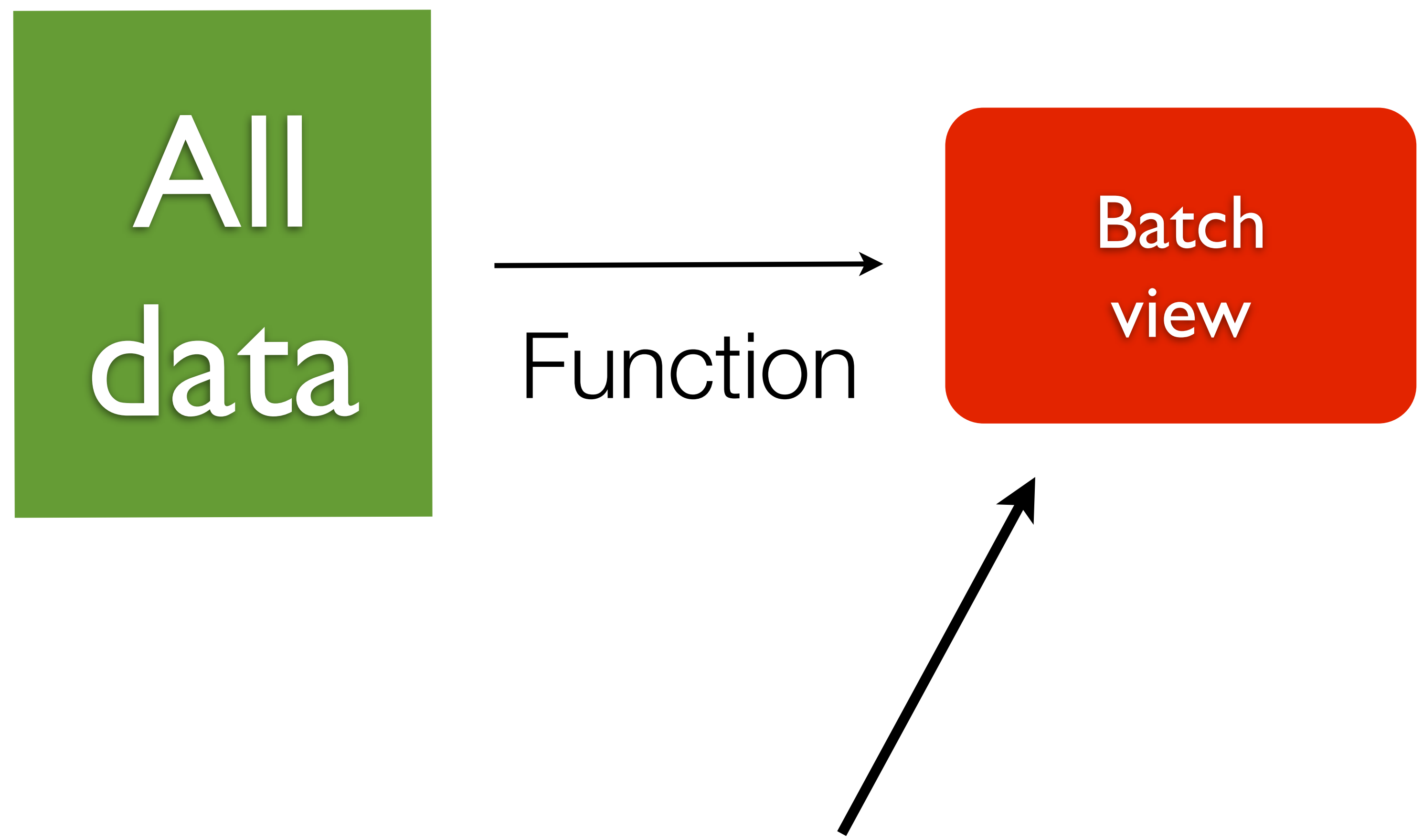


Properties



(without the associated complexities)

Properties

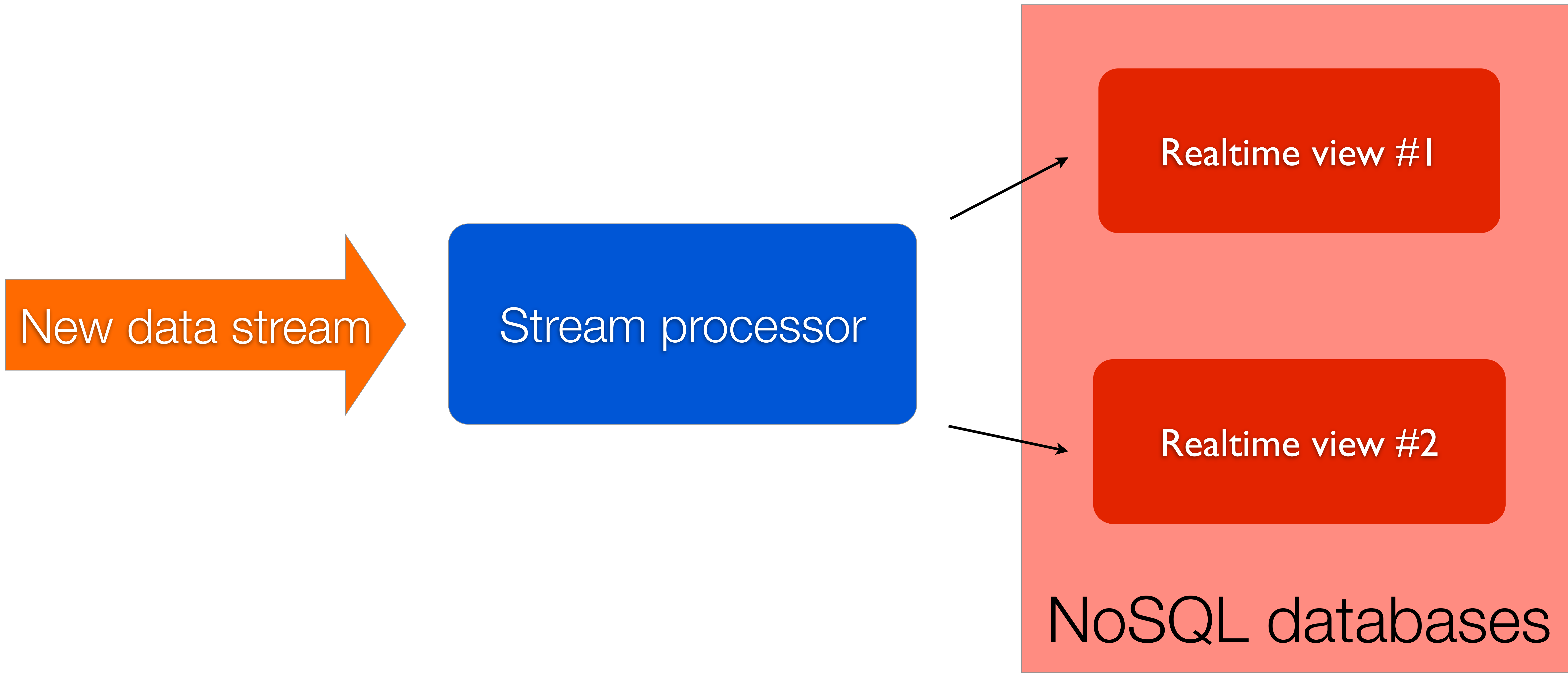


(such as divergent values, vector clocks, etc.)

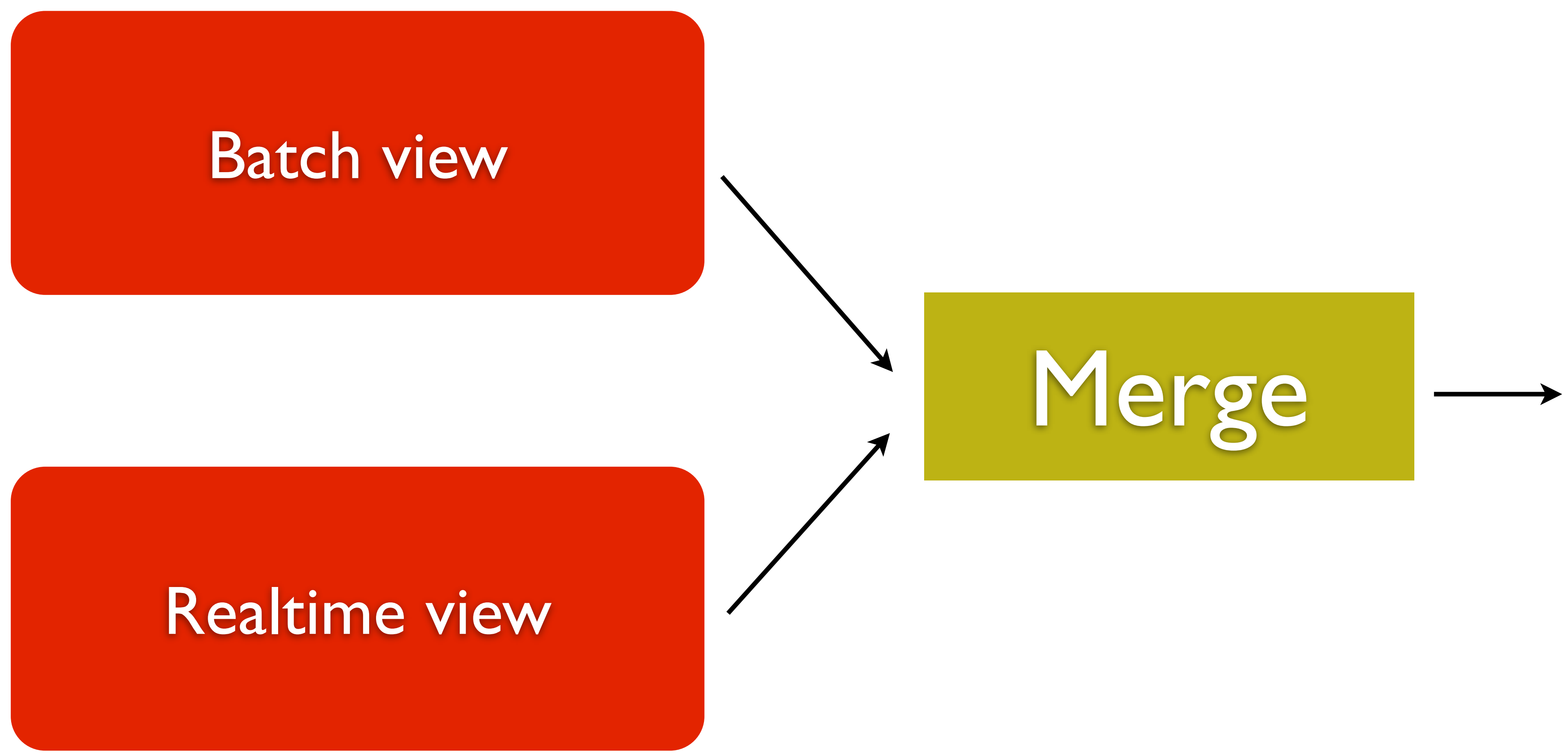
What’s left?

Precompute views for last few hours of data

Realtime views

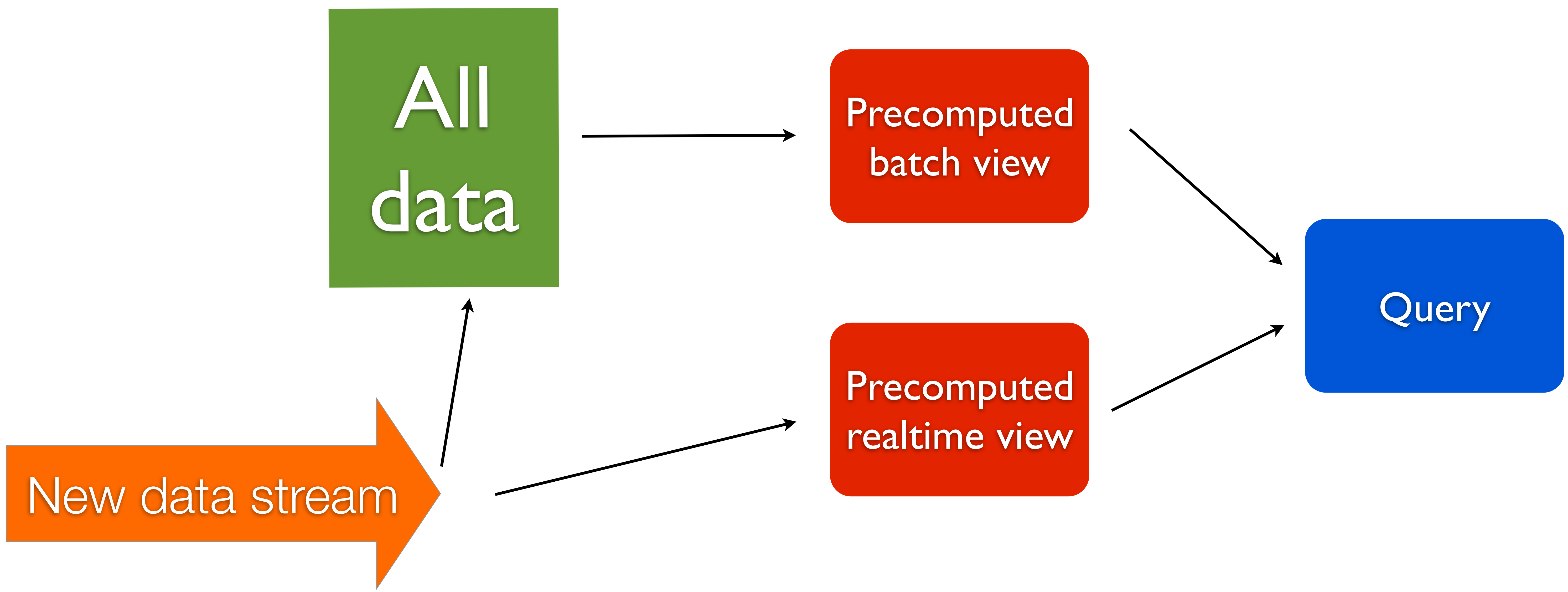

New data stream
Stream processor
Realtime view #1
Realtime view #2
NoSQL databases

Application queries



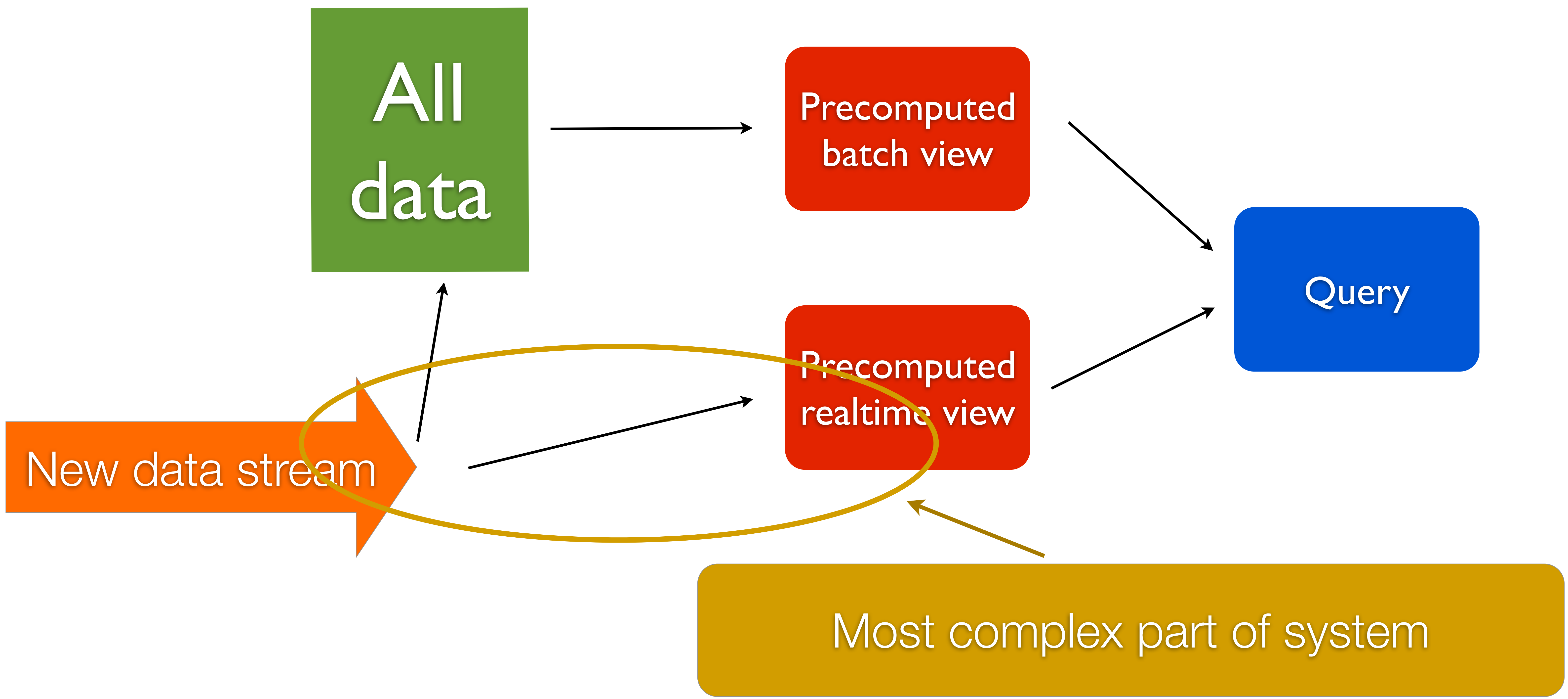
Precomputation

Precomputation

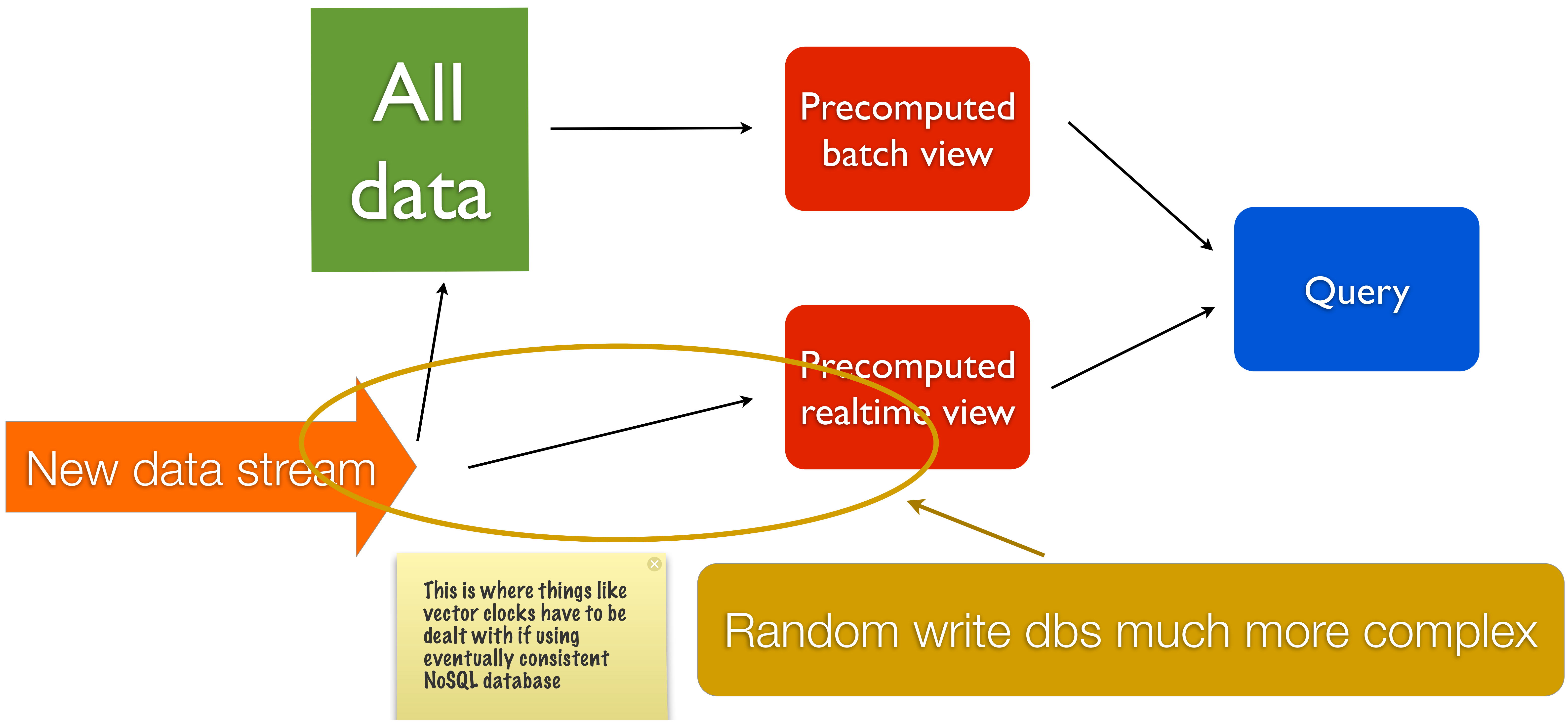


"Lambda Architecture"

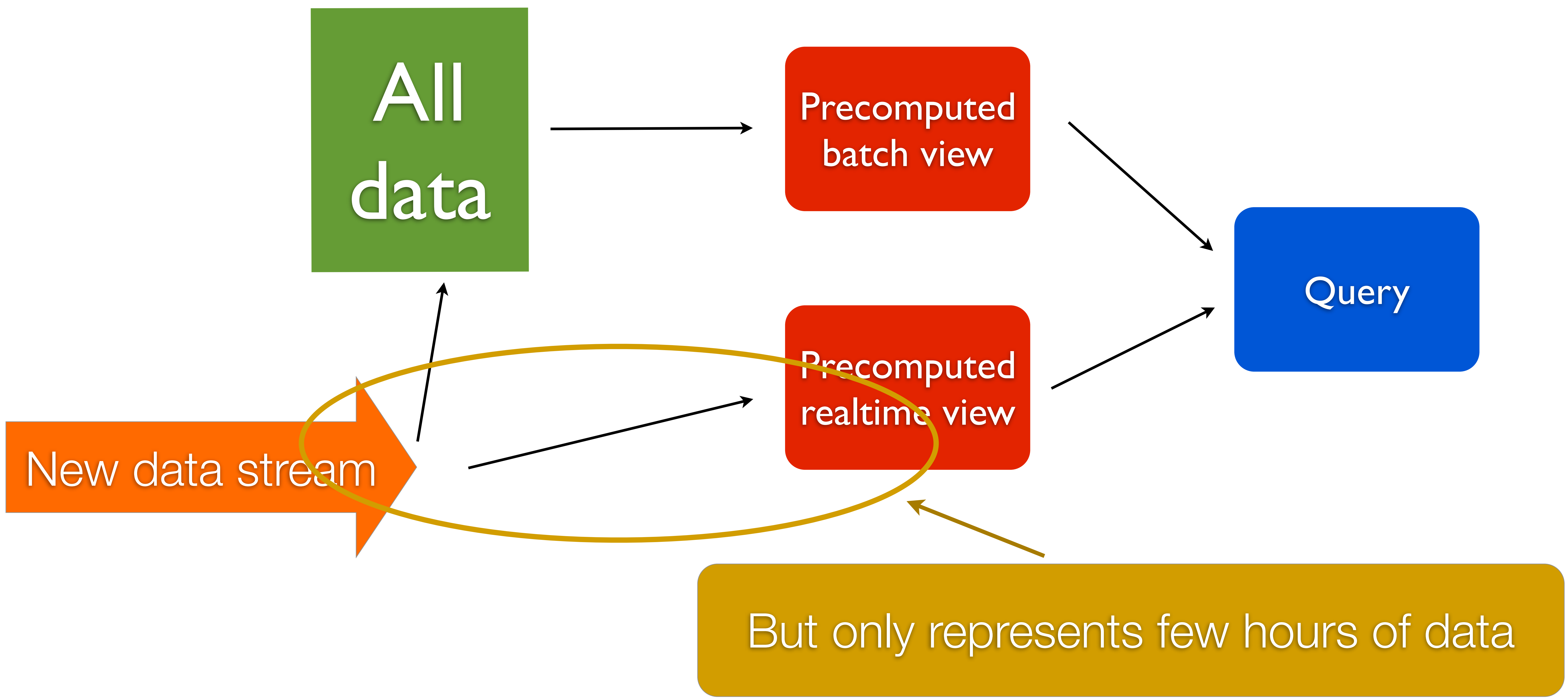
Precomputation



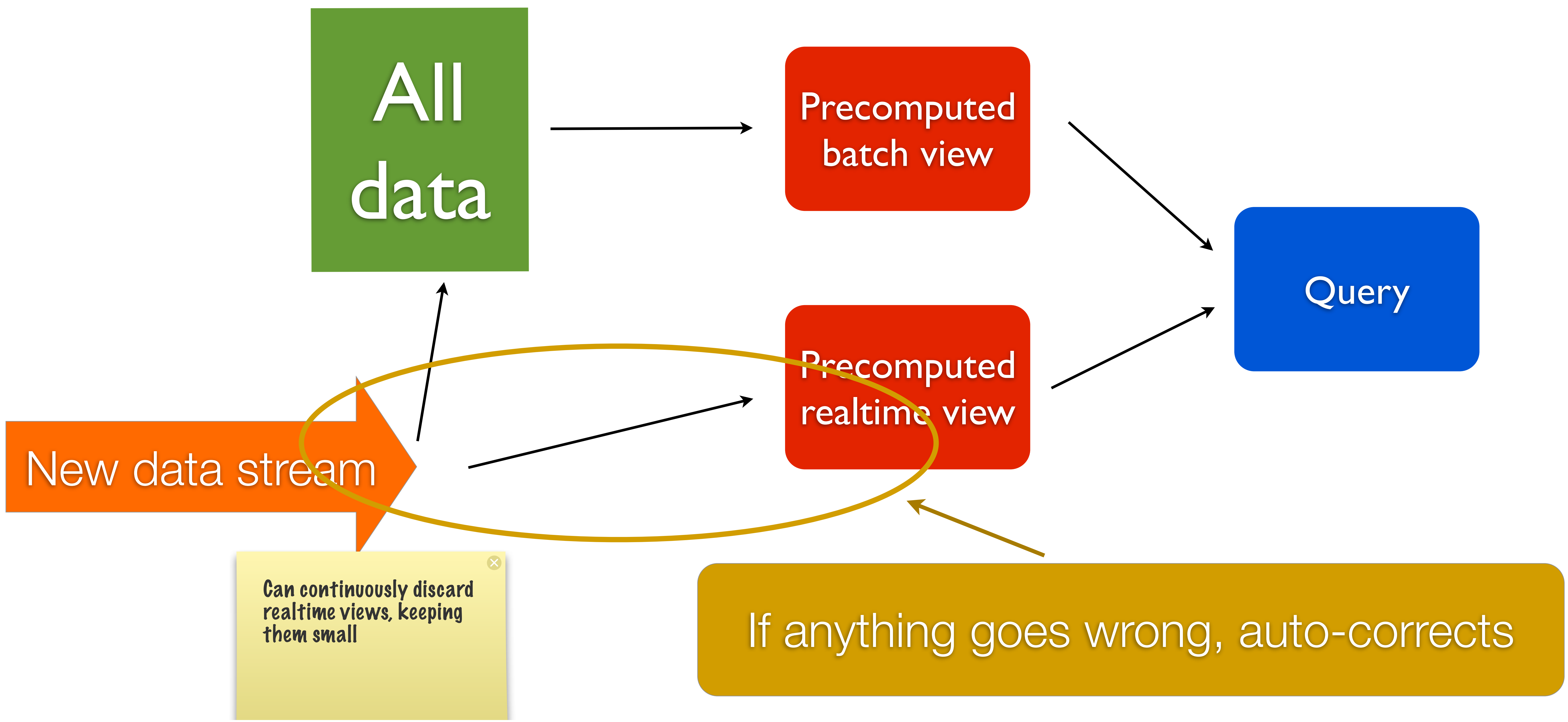
Precomputation



Precomputation



Precomputation



CAP

Realtime layer decides whether to guarantee C or A

- If it chooses consistency, queries are consistent
- If it chooses availability, queries are eventually consistent

All the complexity of *dealing* with the CAP theorem (like read repair) is isolated in the realtime layer. If anything goes wrong, it's *auto-corrected*

CAP is now a choice, as it should be, rather than a complexity burden. Making a mistake w.r.t. eventual consistency *won't corrupt* your data

Eventual accuracy

Sometimes hard to compute exact answer in realtime

Eventual accuracy

Example: unique count

Eventual accuracy

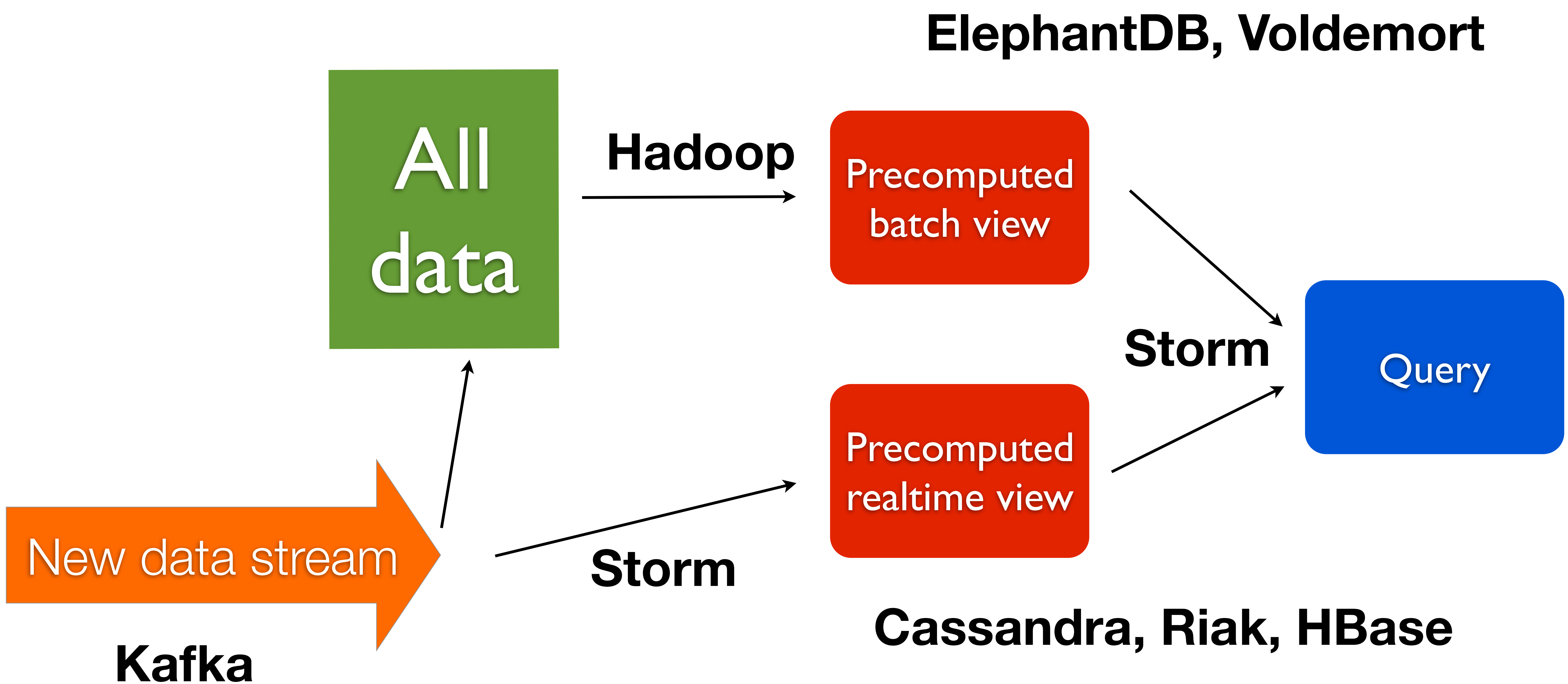
Can compute exact answer in batch layer and approximate answer in realtime layer

Though for functions which can be computed exactly in the realtime layer (e.g. counting), you can achieve full accuracy

Eventual accuracy

Best of both worlds of performance and accuracy

Tools



“Lambda Architecture”

Lambda Architecture

- Can discard batch views and realtime views and recreate everything from scratch

- Mistakes corrected via recomputation

- Data storage layer optimized independently from query resolution layer

what mistakes can be made?

- write bad data? - remove the data and recompute the views

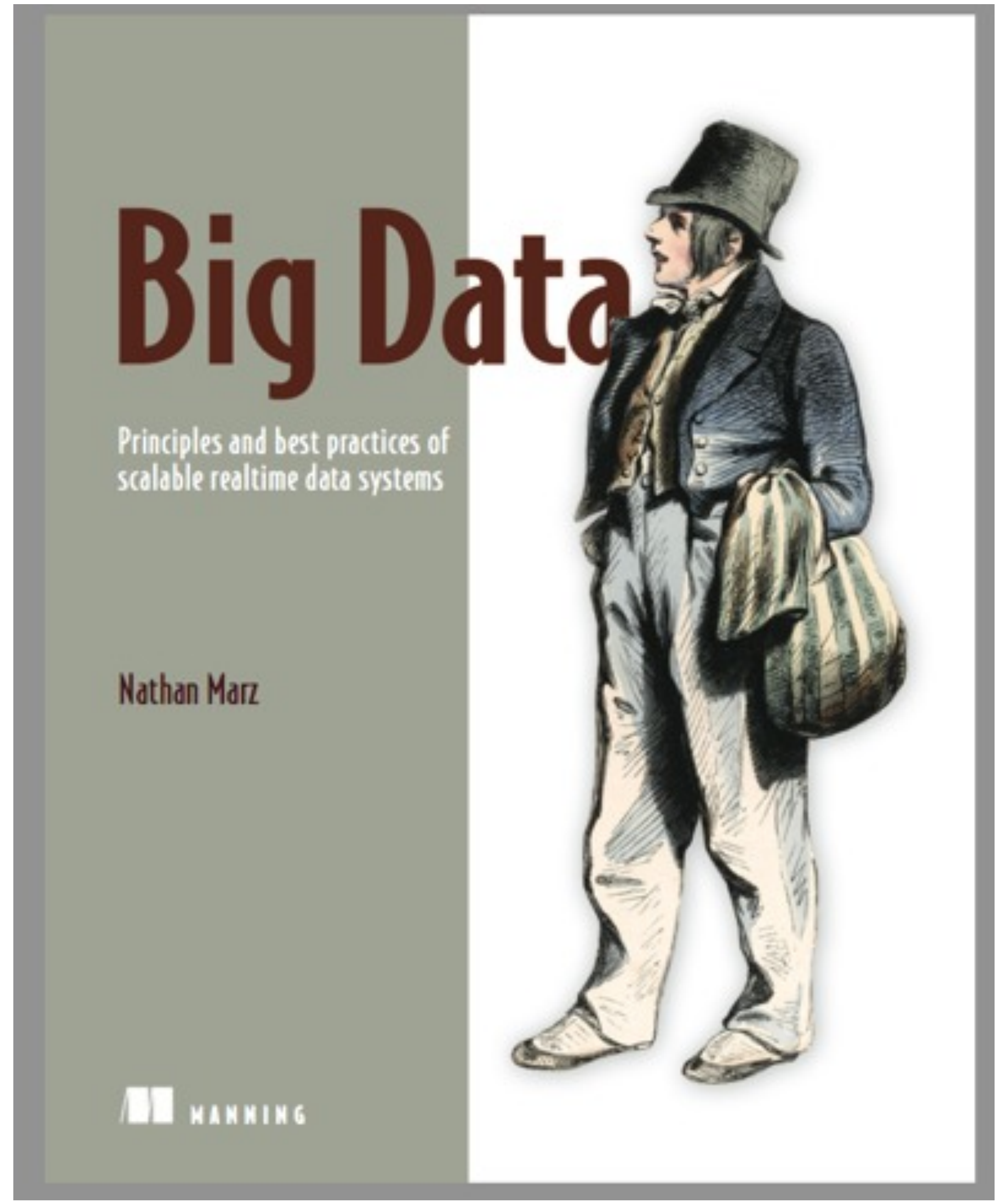
- bug in the functions that compute view? - recompute the view

- bug in query function? just deploy the fix

Future

- Abstraction over batch and realtime
- More data structure implementations for batch and realtime views

Learn more



http://manning.com/marz

Questions?