

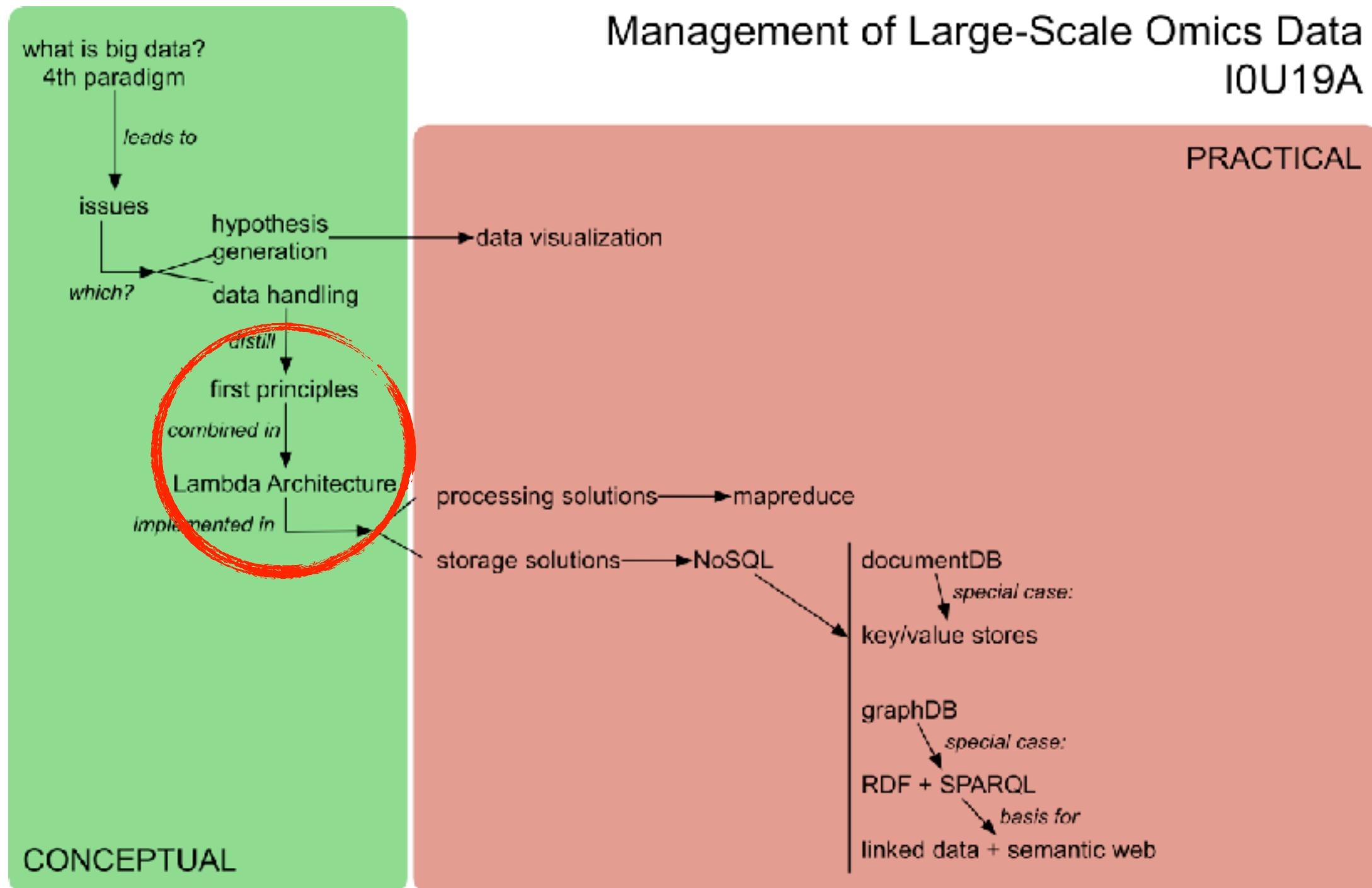
# I0U19A - Management of large-scale omics data

---

Prof Jan Aerts  
Faculty of Engineering - ESAT/SCD  
Kasteelpark Arenberg 10, 3001 Leuven

jan.aerts@kuleuven.be  
<http://vda-lab.be>

# Today - Lambda architecture



# Batch vs real-time

---

- Example Ensembl vs UCSC database showed that
  - Ensembl database allows for wider diversity in queries, but is therefore slower
  - UCSC database is optimized for speed (in a genome browser) but is less ideal for custom queries
- Issue gets more prominent once we get to really large datasets => we need a system that can compute any query (**query = function(all data)**) but at the same time be fast enough to allow interactivity (low latency)

individual	ethnicity	rs12345	rs12345_amb	chr_12345	pos_12345	rs98765	rs98765_amb	chr_98765	pos_98765	rs13579	rs13579_amb	chr_13579	pos_13579
individual_A	caucasian	A/A	A	1	12345	A/G	R	1	98765	G/T	K	5	13579
individual_B	caucasian	A/C	M	1	12345	G/G	G	1	98765	G/G	G	5	13579



primary key

individuals
id
name
ethnicity

primary key

genotypes
id
individual_id
snp_id
genotype
genotype_amb

foreign key

foreign key

primary key

snps
id
accession
chromosome
position

individuals

id	name	ethnicity
1	individual_A	caucasian
2	individual_B	caucasian

genotypes

id	individual_id	snp_id	genotype	genotype_amb
1	1	1	A/A	A
2	1	2	A/G	R
3	1	3	G/T	K
4	2	1	A/C	M
5	2	2	G/G	G
6	2	3	G/G	G

snps

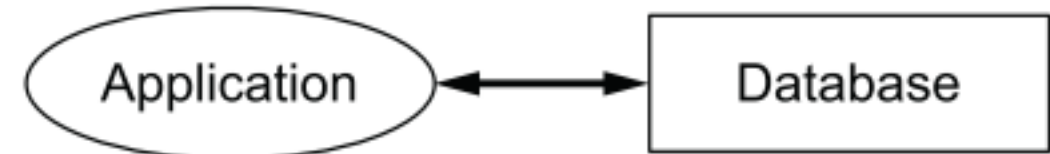
id	accession	chromosome	position
1	rs12345	1	12345
2	rs98765	1	98765
3	rs13579	5	13579

# Application + database

---

Traditional architecture:

- read/write databases
- maintaining state in these databases as new data is added



= *incremental architecture*

# Issues with incremental architecture

---

- *operational complexity*: high chance of failure when operations are complex (e.g. periodic compaction of database)
- *extreme complexity of achieving eventual consistency*: e.g. incrementing counts on distributed system.

2 replicas (A & B) contain copy of counter w/ value 10. If counter is incremented once on A, and twice on B: how are these merged? If diverged at 11: answer = 12; if diverged at 10: answer = 13; if diverged at 0: answer = 23.

- *lack of human-fault tolerance*: updates instead of inserts => can't go back to previous sane state

# Ecosystem for big data

---

- batch computation systems: e.g. Hadoop (HDFS & Hadoop MapReduce)
- serialisation frameworks: provide schema definitions, e.g. Thrift, Protocol Buffers, Avro
- NoSQL databases: e.g. Cassandra, HBase, MongoDB, Voldemort, Riak, ...
- messaging/queuing systems: important for realtime processing, e.g. Apache Kafka
- realtime computation systems: e.g. Storm

# 3 layers in Lambda Architecture

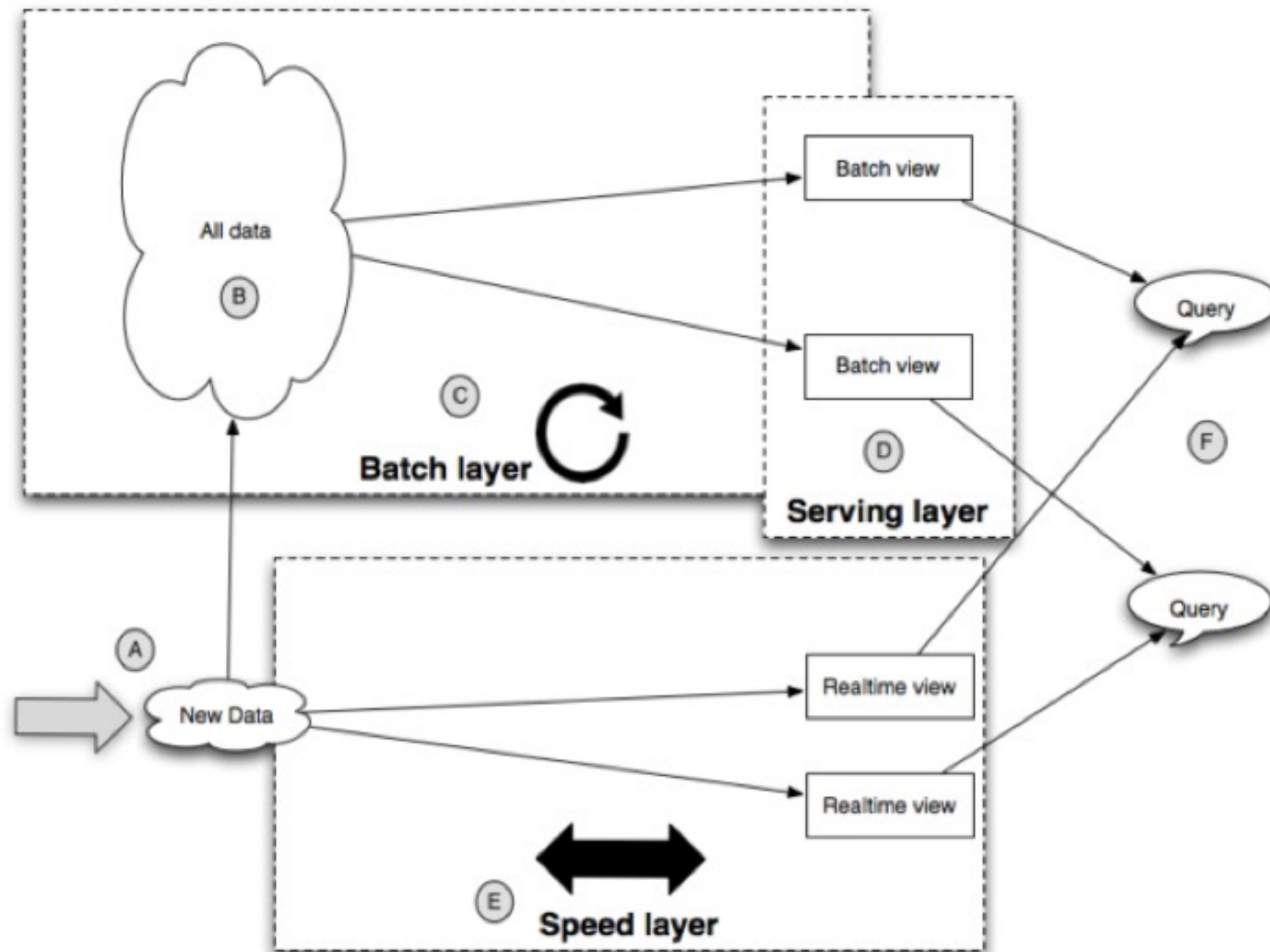
---

- No single tool can provide a complete solution => use a variety of tools and techniques to build a complete Big Data system
- Lambda Architecture: decomposes the problem of computing arbitrary functions on arbitrary data in real-time by decomposing it into 3 layers:

- **batch layer**
- **serving layer**
- **speed layer**







Lambda Architecture diagram (from Big Data, Marz & Warren)

# 1. Batch layer

---

- needs to be able to (1) **store** an **immutable**, constantly growing **master dataset**, and (2) compute **arbitrary functions** on that dataset
- “batch processing systems”, *e.g.* Hadoop
- continuously **recomputes** “views” that are exposed to the serving layer => “**batch views**”
- very **simple**: computations are single-threaded programs, but automatically parallelize across a cluster => scales to datasets of any size

# Properties of data in batch layer

---

- data is raw

The 'rawer' your data, the more questions you can ask.

What is 'raw' for sequencing data?

semantic normalisation: reshaping free-form information into structured form

e.g. unstructured string "Bel" => parsed to "Belgium"

Normalize or not?

- data is immutable

advantages: human-fault tolerance + simplicity

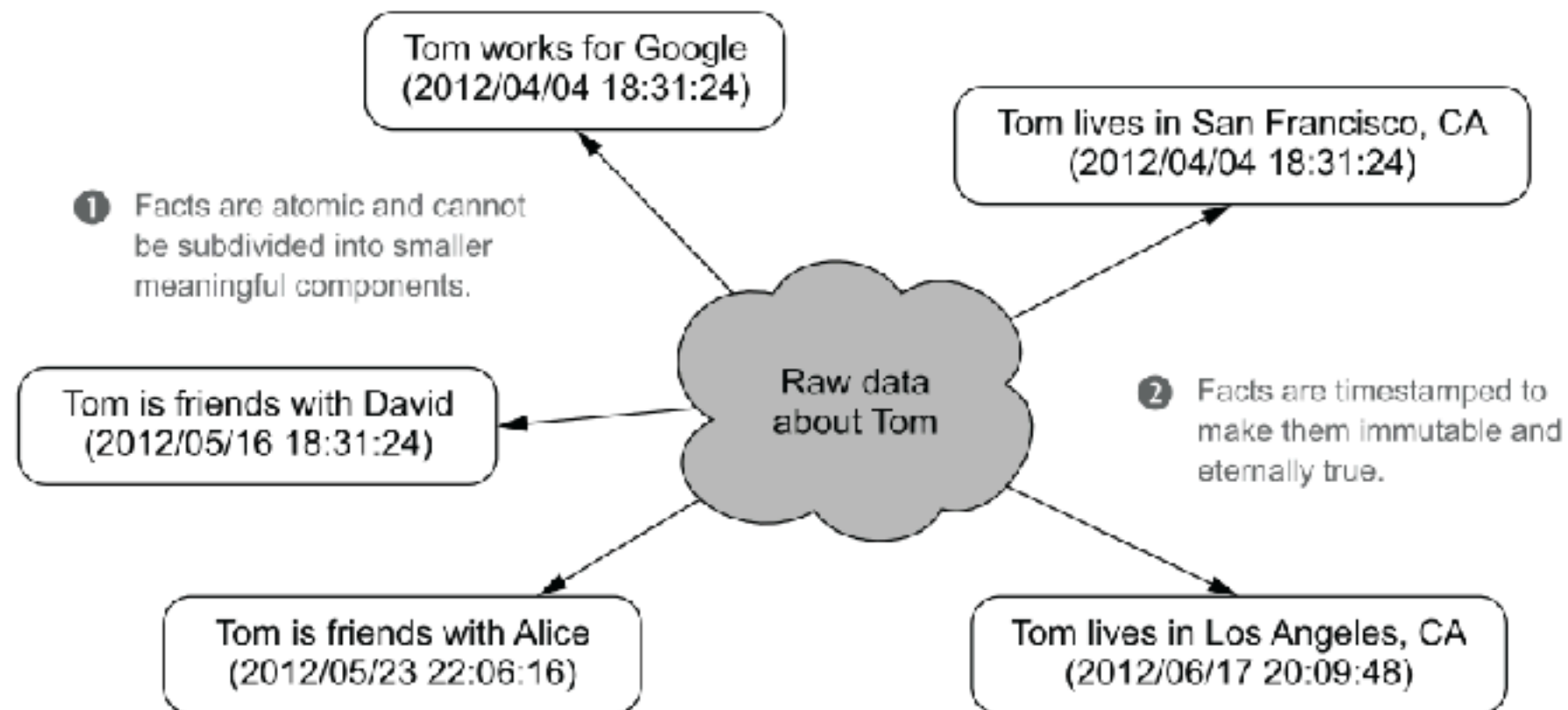
- data is eternally true
  - = result of immutable data
  - =~ learning history: “Henry VIII was King of England in 1500”

Never deleting data, unless:

- garbage collection
- complying to regulation

=> are statements about *value* of the data, not about *truthfulness*

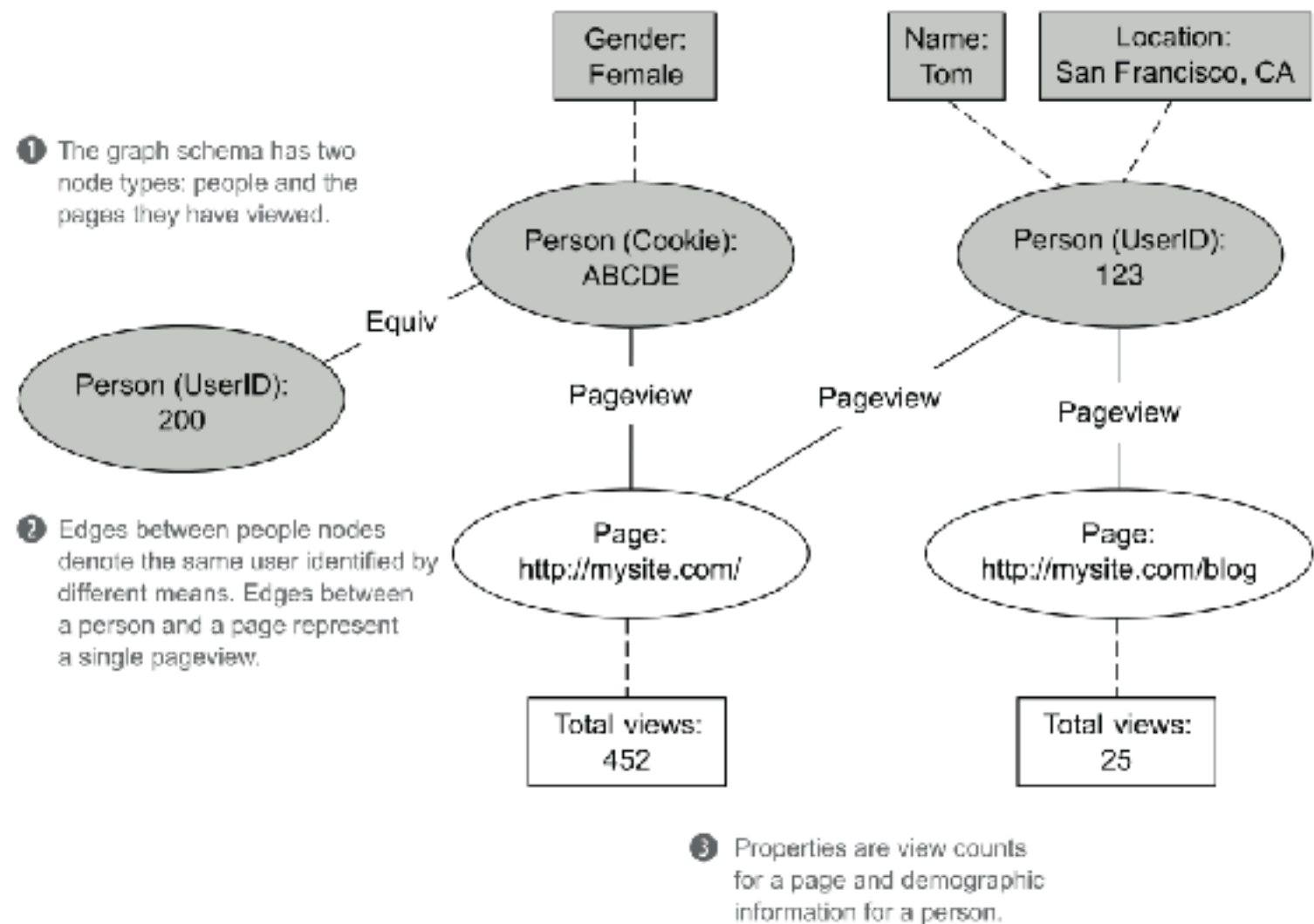
= **fact-based model** for representing data



# Graph schemas

Facts alone don't capture the structure in the data => use graph

- Nodes = entities
- Edges = relationships
- Properties on nodes/edges



## Data Serialization

Data stored in flat text files (csv, json, ...) => prone to **data corruption**, because ***format*** is defined, but not ***schema***.

Serialization frameworks: e.g. Apache Thrift, Protocol Buffers, Avro, ...

### Apache Thrift:

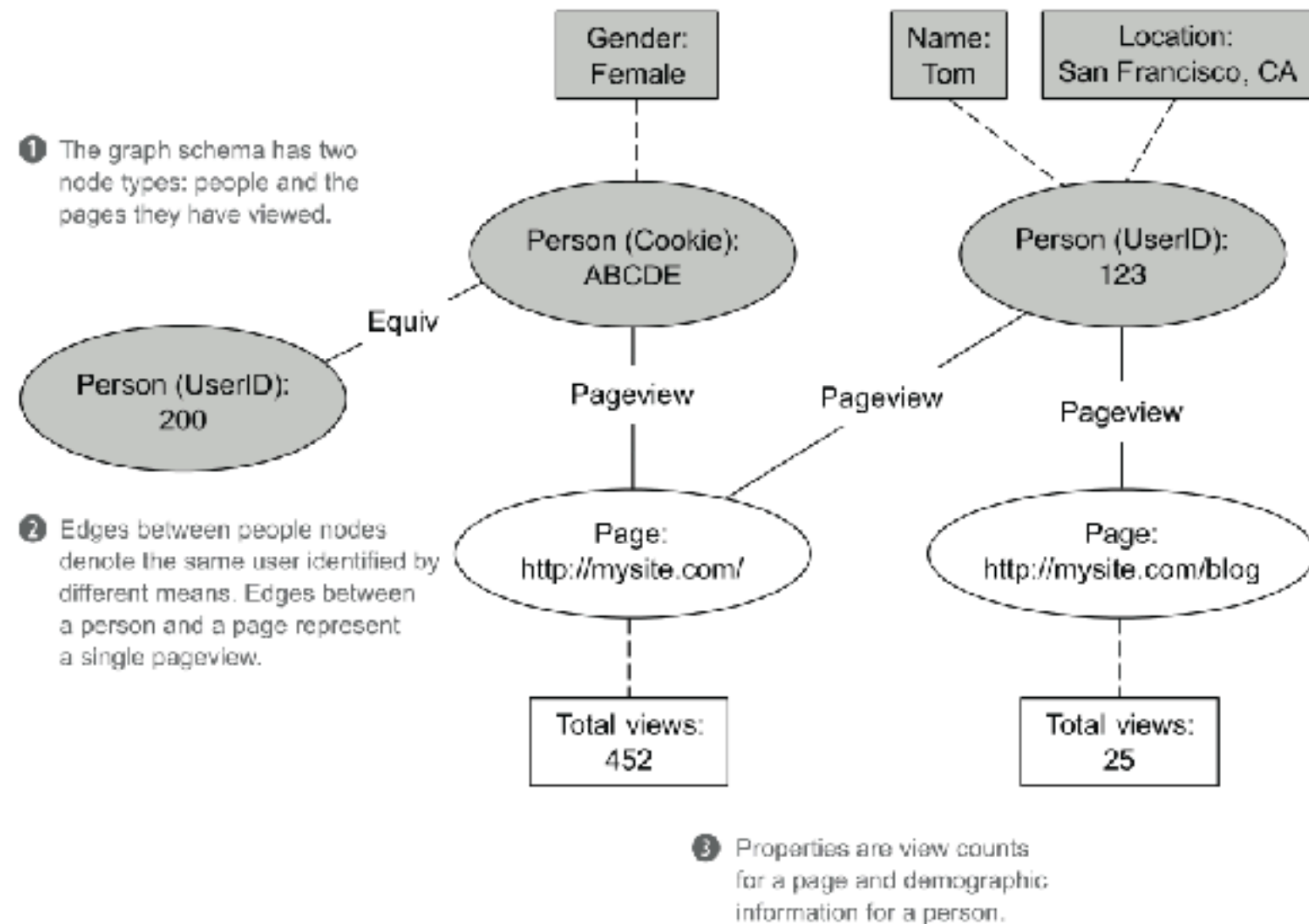
defines *struct* and *union* types, which are themselves composed of:

- primitive data types (e.g. strings, integers)
- collections of other types (e.g. lists, maps, sets)
- other structs and unions

```

union PersonID {
  1: string cookie;
  2: i64 user_id;
}
union PageID {
  1: string url;
}
struct EquivEdge {
  1: required PersonID id1;
  2: required PersonID id2;
}
struct PageViewEdge {
  1: required PersonID person;
  2: required PageID page;
  3: required i64 nonce;
}
union PagePropertyValue {
  1: i32 page_views;
}
struct PageProperty {
  1: required PageID id;
  2: required PagePropertyValue property;
}

```





## Limitations of serialisation frameworks

Frameworks can only check that all required fields are present and of the expected type. NOT: richer properties (e.g. age should be  $> 0$ )

Workarounds:

1. Wrap generated code in additional code that checks additional properties. ([-] if using multiple languages: need to duplicate code)
2. Check extra properties at beginning of batch-processing workflow ([-] splits dataset in “valid data” and “invalid data” => doesn't prevent invalid data to be written to master dataset)

## 2. Serving layer

---

- load views from batch layer and make them queryable through indexing
- together with batch layer: satisfy almost all properties we need
- only requires batch updates and random reads (does not need to support random writes)

raw data

**batch layer**

can be files on file system, ...

id	who	timestamp	action	who
1	Tom	20100402	add	Frank
2	Tony	20100404	add	Frank
3	Tom	20100407	remove	Frank
4	Tim	20100409	add	Frank
5	Tom	20100602	add	Freddy
6	Tony	20100818	add	Francis
7	Tom	20100819	add	Frank
8	Tony	20101021	add	Flint
9	Tony	20110101	add	Fletcher

no updates!

friend counts

name	friends
Tom	2
Tim	1
Tony	4

How many friends does Tom have?

friend lists

name	friends
Tom	[Frank, Freddy]
Tim	[Frank]
Tony	[Frank, Fletcher, Flint, Francis]

Is Fletcher a friend of Tim?

**serving layer**

## **Serving layer = solution to normalization/denormalization dilemma**

Relational databases: choose between unacceptable trade-offs

normalised data: high consistency, low performance

denormalised data: low consistency, high performance

=> split between master dataset and serving layer =~ between normalised data and denormalised data

# Batch & serving layers satisfy almost all properties

---

- **robust & fault tolerant:** simple system; views can be recomputed; easily distributable
- **scalable:** can be implemented as fully distributed systems
- **general:** can compute and update arbitrary views of arbitrary data
- **extensible:** adding new view = adding new function on data
- **allows ad hoc queries:** batch view allows any function
- **minimal maintenance:** very few pieces
- **debuggable:** you always have the inputs and outputs of computations run on the batch layer (<-> traditional DBs: you *update* the data)

### 3. Speed layer

---

- Serving layer updates whenever batch layer finishes precomputing views => the only data not represented in the batch views is the data that came in while the precomputation was running

=> only have to compensate for these last few minutes/hours of data

- speed layer is similar to batch layer: produces views based on data it receives

raw data

batch layer

id	who	timestamp	action	who
1	Tom	20100402	add	Frank
2	Tony	20100404	add	Frank
3	Tom	20100407	remove	Frank
4	Tim	20100409	add	Frank
5	Tom	20100602	add	Freddy
6	Tony	20100818	add	Francis
7	Tom	20100819	add	Frank
8	Tony	20101021	add	Flint
9	Tony	20110101	add	Fletcher

add record:  
10 | Tina | 20140313 | add | Fiona

friend counts

friend lists

name	friends
Tom	2
Tim	1
Tony	4

name	friends
Tom	[Frank, Freddy]
Tim	[Frank]
Tony	[Frank, Fletcher, Flint, Francis]

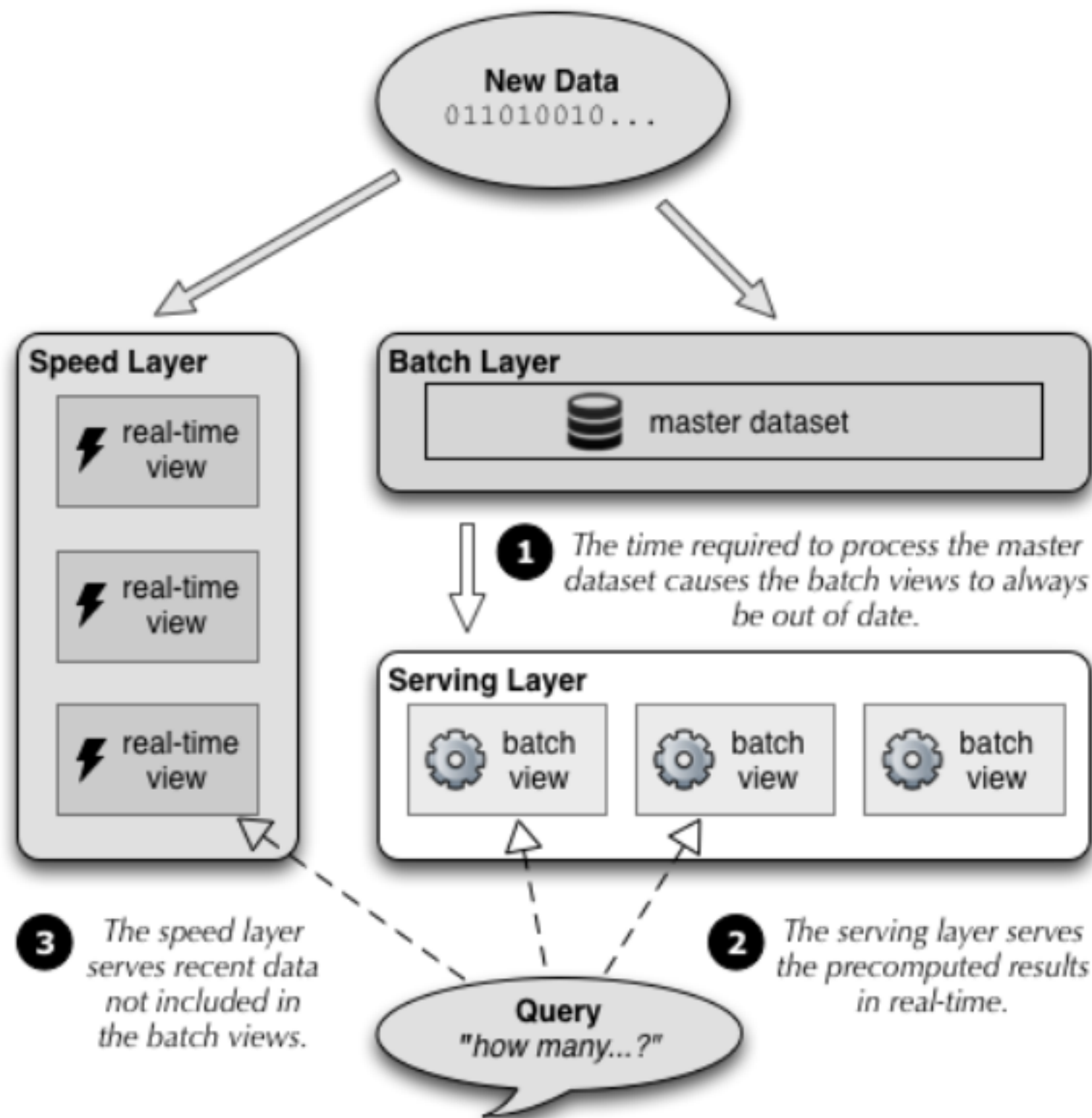
serving layer

takes time

How many friends does Tom have?

Is Fletcher a friend of Tim?

out of date!



Serving layer provides low-latency access to results of calculations performed on master dataset. Serving layer views are slightly out-of-date due to time required for batch computation.



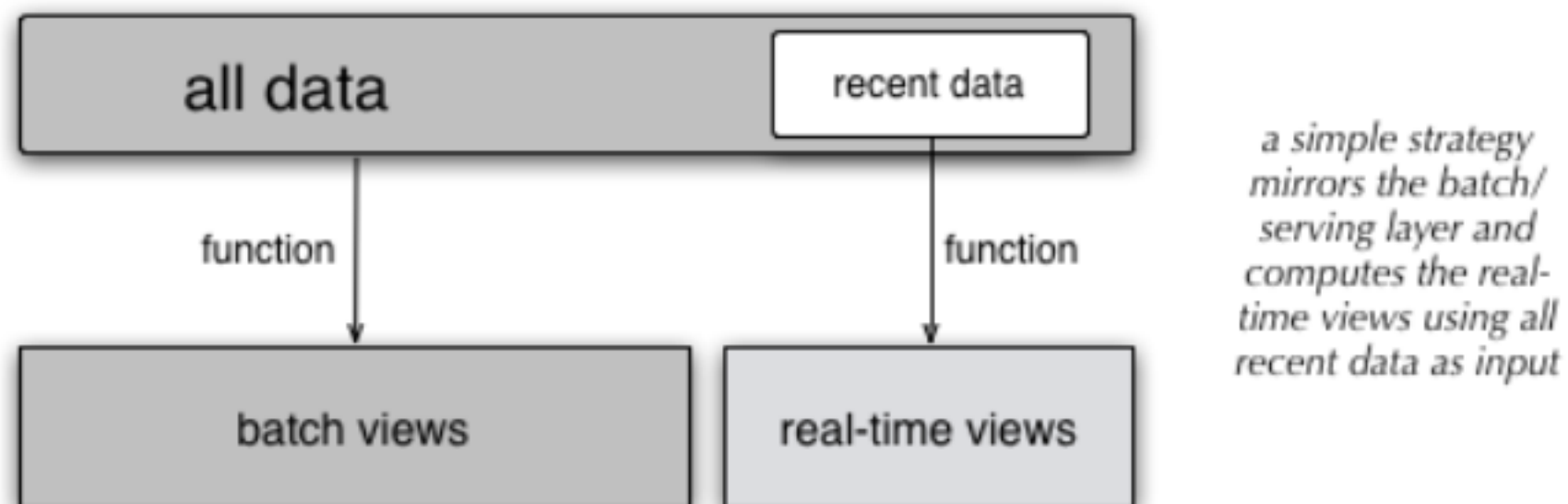
- Fundamental different approach than batch and serving layers: **incremental updates** (<-> batch/serving: **recomputation updates**)
- Consequence: significantly more complex!
  - reason: needs random reads and random writes
- Fortunately: narrow requirements => important **advantages**:
  - speed layer is only responsible for data that is not yet included in serving layer => vastly **smaller dataset**
  - speed layer views are transient => any **errors are short-lived**

- **complexity isolation:** complexity is pushed into a layer whose results are only temporary
- last piece of Lambda Architecture: merging results from serving and speed layers

## how to compute real-time views

- simple approach: same *recomputation* function as batch layer, but only on recent data => real-time view = function(recent data)
- problem: will still have some latency => sub-second speed is difficult

=> we need *incremental algorithm*: does not need to constantly recompute when new data arrives



## how to store real-time views

- storage layer must meet these requirements:
  - *random reads* - data must be indexed
  - *random writes* - must be possible to modify a real-time view with low latency
  - *scalability* - real-time views can be distributed across many machines
  - *fault tolerance* - data must be replicated across machines in case one machine fails

raw data

**batch layer**

id	who	timestamp	action	who
1	Tom	20100402	add	Frank
2	Tony	20100404	add	Frank
3	Tom	20100407	remove	Frank
4	Tim	20100409	add	Frank
5	Tom	20100602	add	Freddy
6	Tony	20100818	add	Francis
7	Tom	20100819	add	Frank
8	Tony	20101021	add	Flint
9	Tony	20110101	add	Fletcher

add record:  
10 | Tom | 20140313 | add | Fiona

**new data:** Fiona is now friend of Tom

update speed views

friend lists

name	friends
Tom	[Frank, Freddy]
Tim	[Frank]
Tony	[Frank, Fletcher, Flint, Francis]

**serving layer**

friend counts

name	friends
Tom	2
Tim	1
Tony	4

How many friends does Tom have?

friend lists

name	friends
Tom	[Fiona]

**speed layer**

friend counts

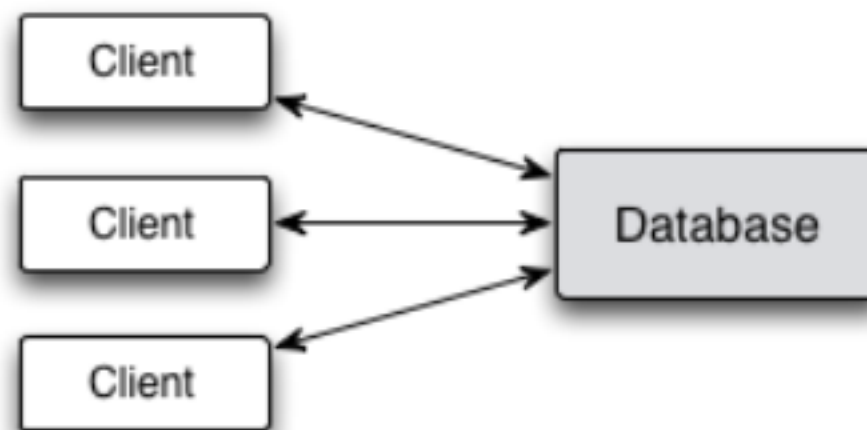
name	friends
Tom	1

## eventual accuracy

- often very difficult to incrementally compute functions which are easily computed in batch (e.g.: word count)
- possible approach: *approximate* the correct answer
  - will be continuously corrected through batch/serving
- common with sophisticated queries such as those requiring real-time machine learning

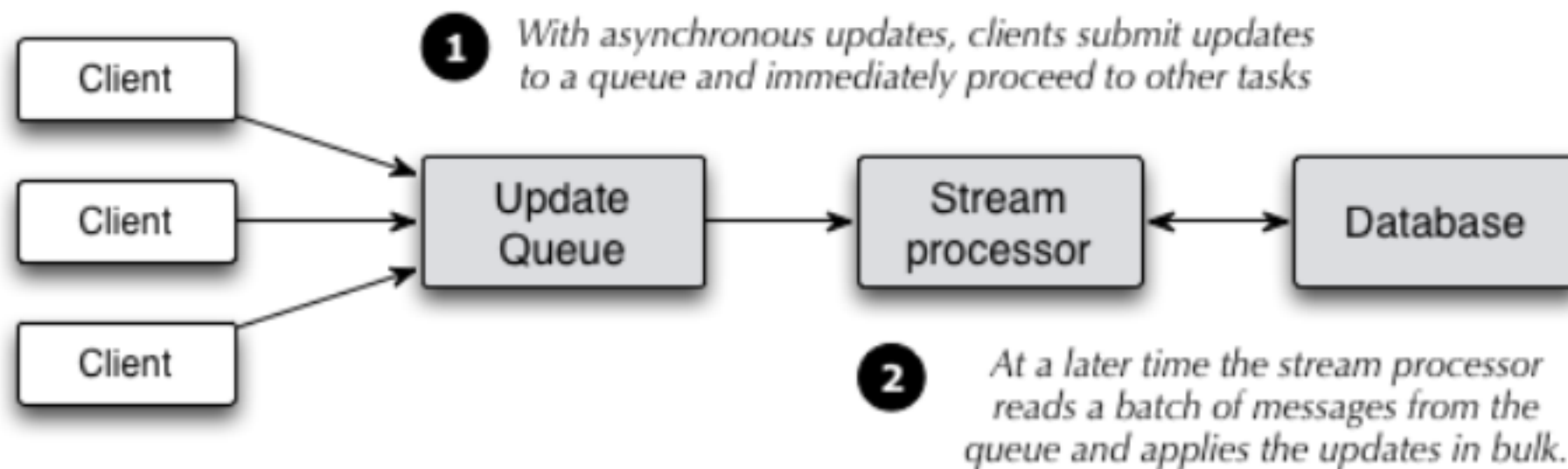
## asynchronous vs synchronous updates

- synchronous update: application issues request directly to the database and *blocks* until the update is processed
- update can be coordinated with other aspects of the application (such as displaying a spinning cursor while waiting for the update to complete)



*With synchronous updates, clients communicate directly with the database and block until the update is completed*

- asynchronous update: put in *queue*
- impossible to coordinate with other actions since you cannot control when they are executed
- but advantages: greatly increases throughput + can handle varying load more easily



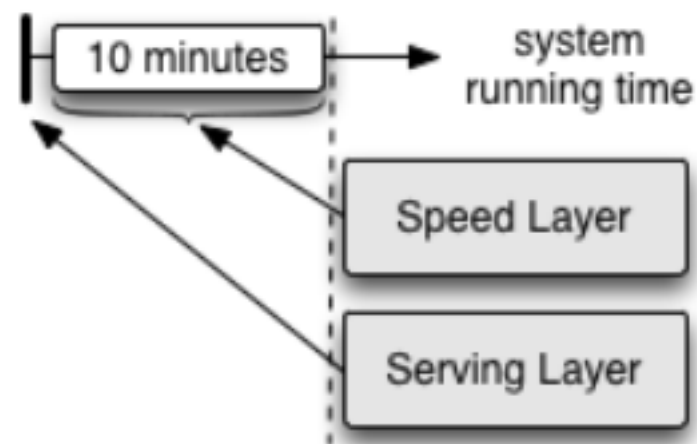


## When to use synchronous vs asynchronous?

- synchronous update: typical for transactional systems that interact with users and require coordination with user interface
- asynchronous update: typical for analytics-oriented workloads or workloads not requiring coordination
- asynchronous is architecturally simpler => use asynchronous unless you have a good reason not to do so

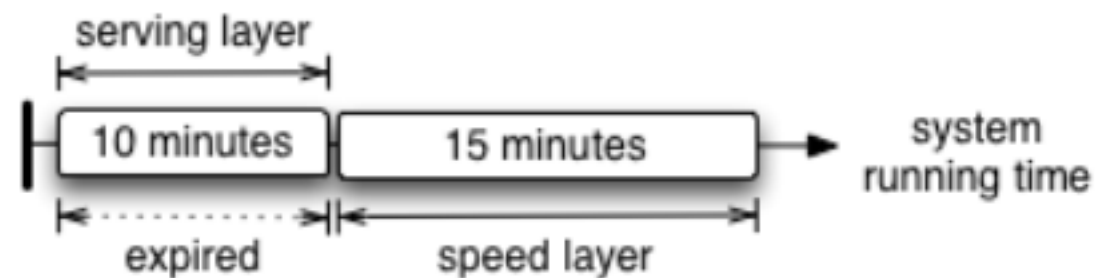
## how to expire real-time views

- once batch computation run finishes: you can discard a portion of the speed layer views, but you need to keep everything else => what exactly needs to be expired?
- suppose: just turned on application => no data  
=> when batch layer runs **first**: operates on no data  
=> if batch takes 10 minutes: when finished: batch views empty + 10min worth of data in speed



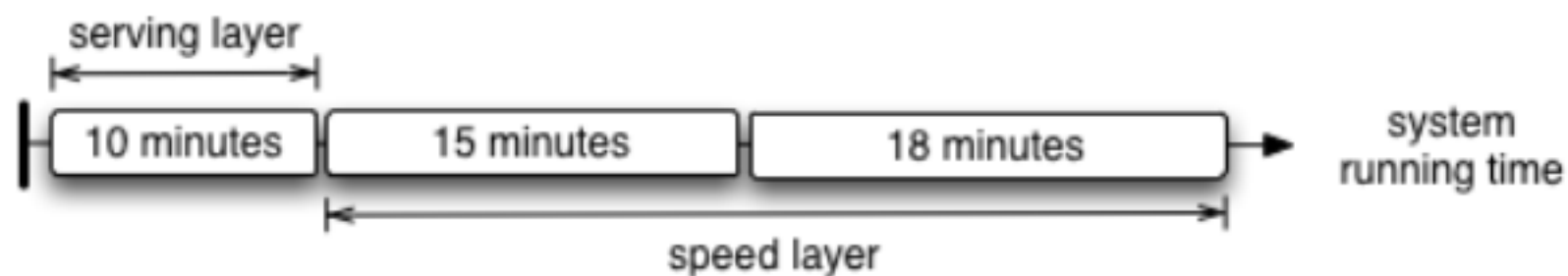
*After the first batch computation run, the serving layer remains empty but the speed layer has processed recent data.*

=> **2nd batch run** starts immediately after 1st  
=> say 2nd run takes 15 minutes => when finished: batch view covers 1st 10 minutes + speed layer views represent 25 minutes of data  
=> first 10 minutes can be expired



*When the second batch computation run finishes, the first segment of data has been absorbed into the serving layer and can be expired from the speed layer views.*

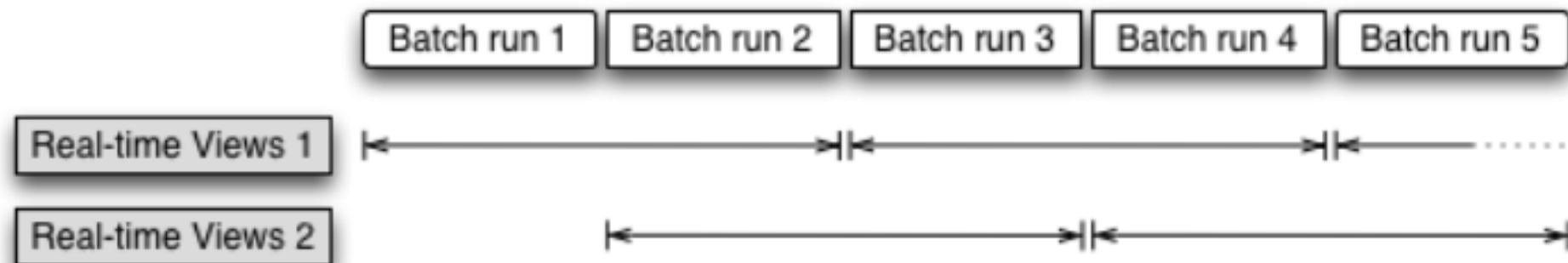
- => **3rd batch run** starts immediately after 2nd
- => say 3rd run takes 18 minutes
- => just after 3rd run started: speed layer responsible for single run (i.e. 2nd)
- => just *before* completion: speed layer responsible for data that accumulated in previous 2 runs
- => when 3rd run completes: data from 3 runs ago can be discarded

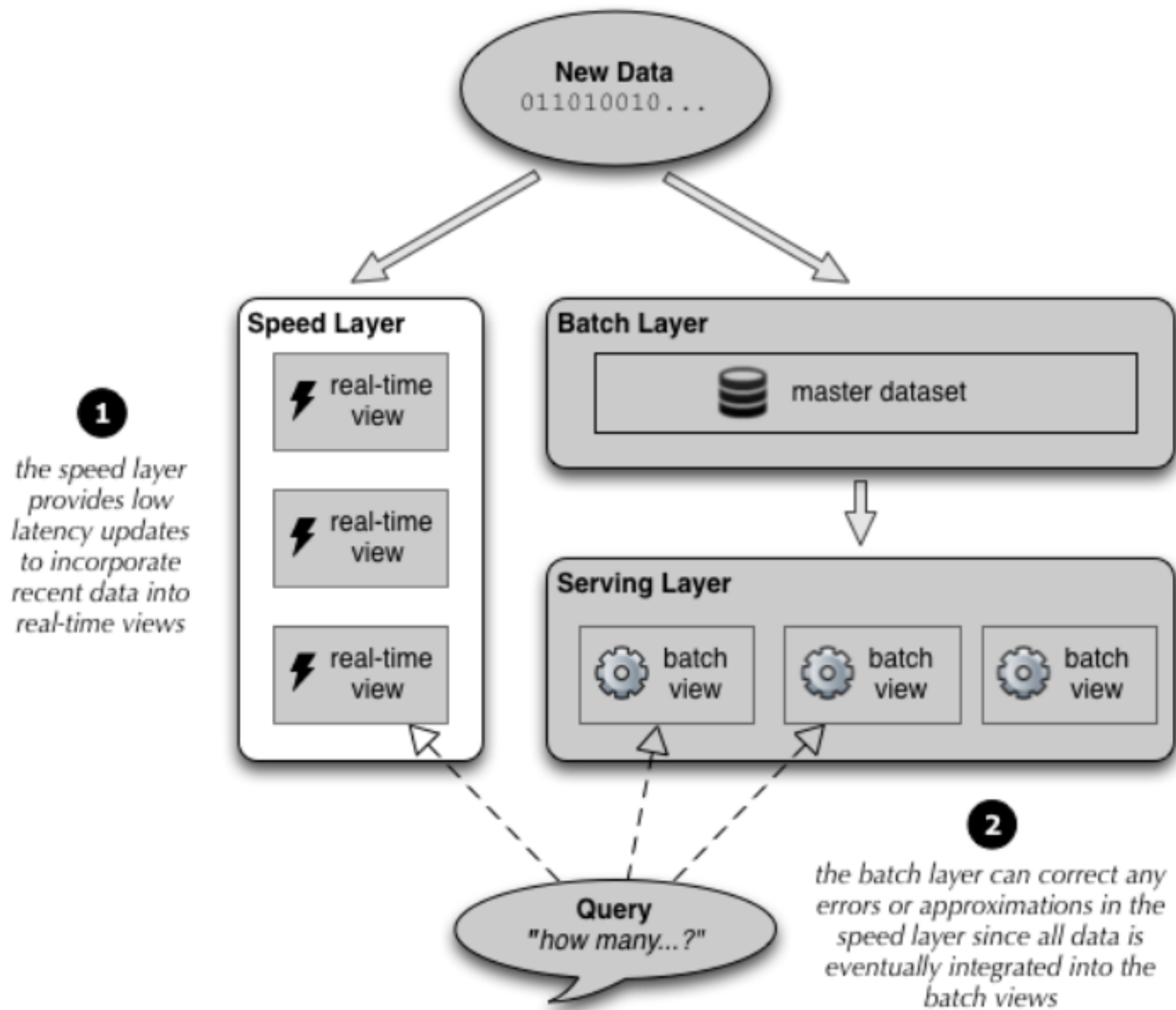


*Just prior to the completion of the batch layer computation, the speed layer is responsible for data that accumulated for the prior two runs.*

simplest way to do this: maintain *two* sets of real-time views and alternate clearing them after each batch run

=> after each batch layer run: application should switch to reading from the real-time view with more data





Speed layer allows Lambda Architecture to serve low-latency queries over up-to-date data

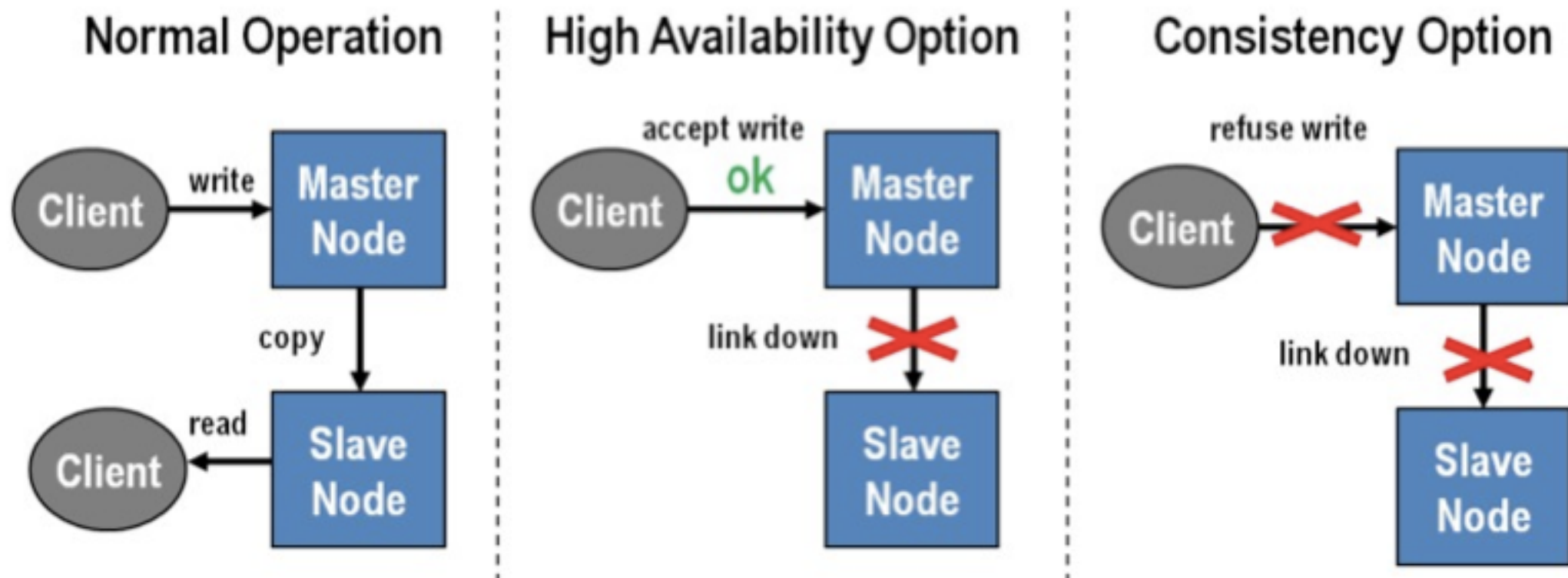
## Brewer's CAP theorem: understanding trade-offs in **distributed systems**

- Using a formal process to understand the trade-offs in your selection process will help drive your focus towards things most important.
- “When a distributed data system is partitioned, it can *consistent* or *available*, but *not both*.”

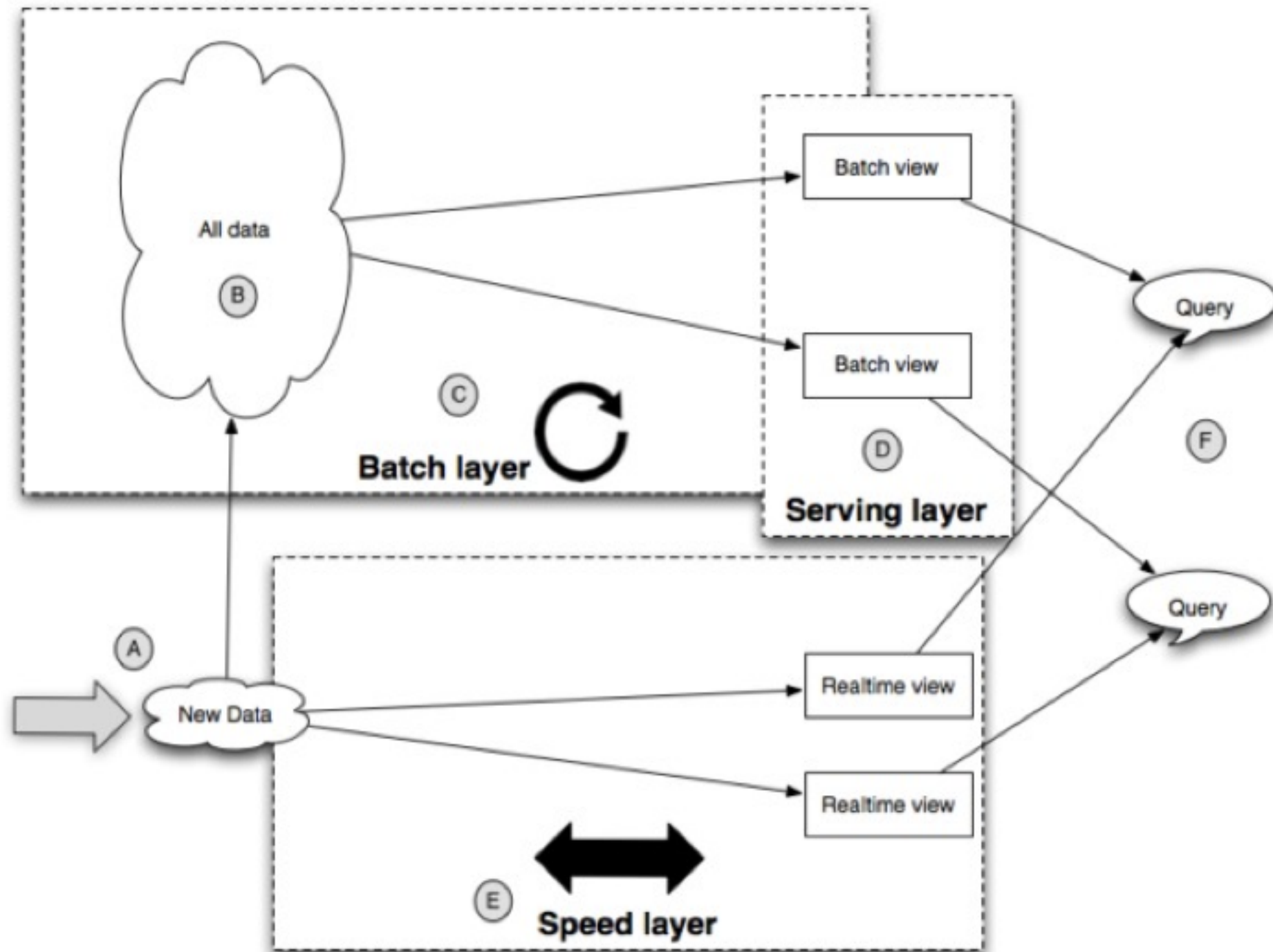
- **Consistency** (!= the consistency in ACID) - multiple clients reading the same items from **replicated partitions** get consistent results
- **high Availability** - knowing that the distributed database will always allow database clients to update items without delay (=> internal communication failures between replicated data should not prevent updates)
- **Partition tolerance** - ability for the system to keep responding to client requests even if there is a communication failure between database partitions
- CAP theorem only applies when something goes wrong in communication => the more reliable your network the less chance you need to think about CAP



- => CAP theorem helps you understand that once you partition your data you must consider the availability <-> consistency spectrum in case of network failure



- to recapitulate:



# Data Processing in Lambda Architecture

- batch systems: mapreduce
  - very simple conceptually: map phase + reduce phase
- real-time systems: spark, storm, ...
  - technologically much more complex

see next 2 sessions

# Data Storage in Lambda Architecture

# NoSQL - Not Only SQL

---

See following lectures

= set of concepts that allow the rapid and efficient processing of datasets with a focus on performance, reliability and agility. Core themes:

- free of joins
- schema-free
- works on many processors
- uses shared-nothing commodity computers
- supports linear scalability

# NoSQL concepts

---

## 1. **keeping components simple** to promote reuse: linux pipe philosophy

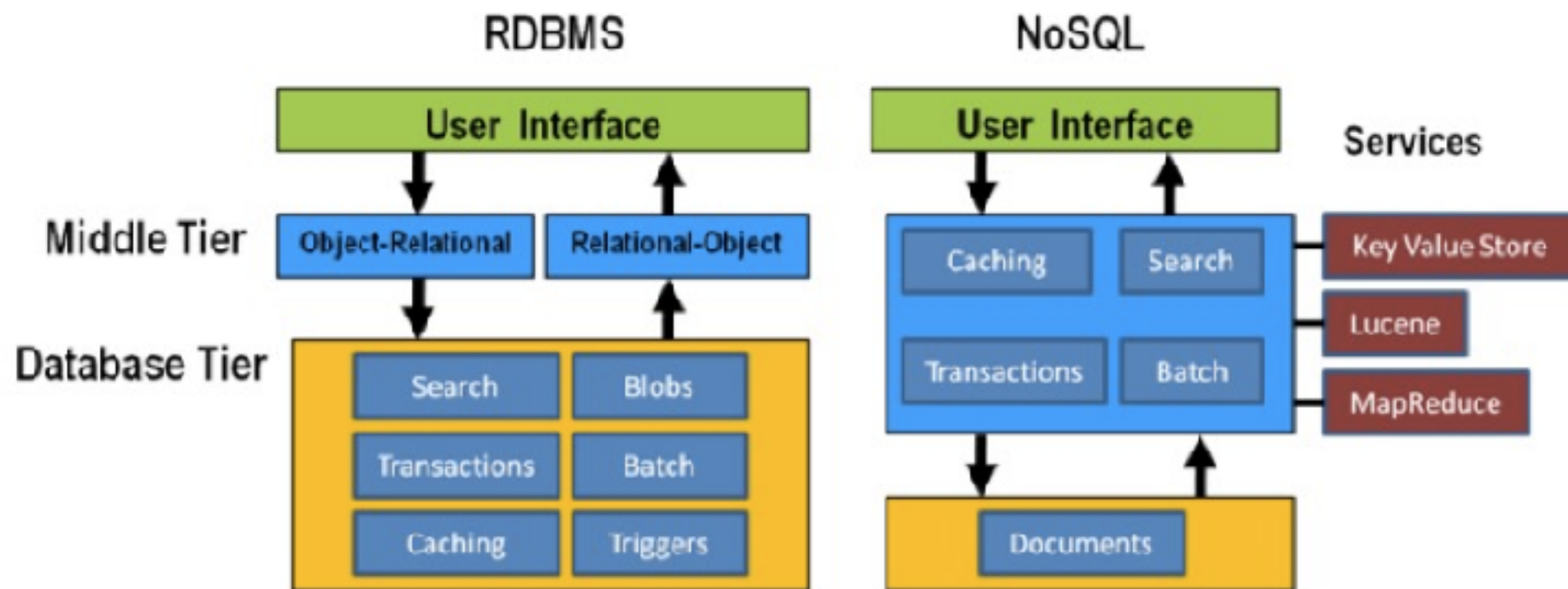
- NoSQL systems are often created by integrating a large number of modular functions that work together  $\Leftrightarrow$  traditional RDBMS: mammoth system  
`cat data.csv | grep "chr1" | cut -f 2 | sort | uniq -c`
- *e.g.* one function allows sharing of objects in RAM, another function to run batch jobs, yet another function for storing documents
- NoSQL system interfaces are broader than regular STDIN and STDOUT (*i.e.* using line delimiters): can be documents, REST, JSON, XML, ... services

## 2. using **application tiers** to simplify design

- by segregating an application into tiers you have the option of modifying or adding a specific layer instead of reworking an entire application => **separation of concerns**
- Lambda Architecture
- user interface <-> middle tier <-> database layer (e.g LocusTree)



- where do you put functionality? trade-offs
- RDBMS: have been along for long time and are mature => much functionality added to database tier
- NoSQL: most of application functionality is in middle tier



### 3. speeding performance by **strategic use of RAM, SSD, and disk** (e.g. Spark)

- you: in Leuven
  - getting something from RAM = your backyard
  - getting something from SSD (solid state drive) = somewhere in your neighbourhood
  - getting something from disk = traveling to Saudi Arabia
  - getting something from the network = traveling to Jupiter

#### 4. using **consistent hashing** you keep your cache current

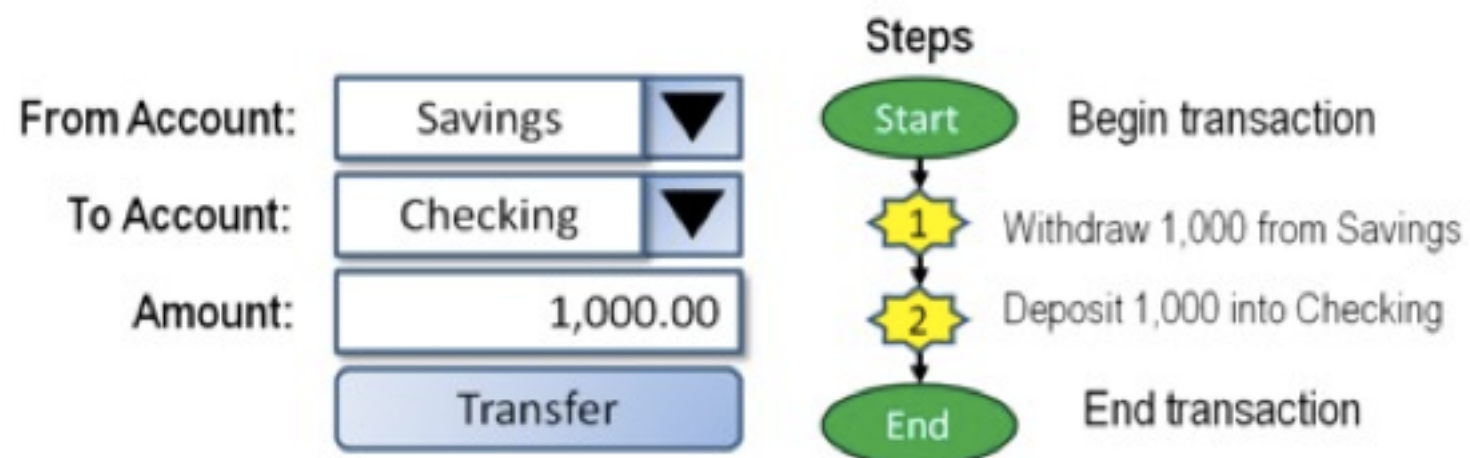
- hash string or checksum = process that calculates sequence of letters by looking at each byte of a document (e.g. MD5, SHA-1)
- consistent hashing: quickly tells you if a new query or document is the same as one already in your cache. Consistent hashing occurs when two different processes running on different nodes in your network create the same hash for the same object
- changing a single bit in the object/file changes the MD5 sum

md5sum my\_file.txt => 5a13448726555d031061aef7432b45c3

- in principle: possible that 2 different documents create the same hash value = **hash collision**
  - MD5 algorithm: 128 bit string => occurs once every  $10^{38}$  documents => if you generate a billion documents a second it would take 10 trillion times the age of the universe for a single accidental collision to occur
- consistent hashing: can also be used to assign documents to specific database nodes

## 5. ACID vs BASE - two methods of reliable database transactions

- ACID & BASE = “transaction control models”
- RDBMS: ACID  
NoSQL: BASE
- example: bank transaction: making sure that the two processes in figure happen either together or not at all



- **ACID**

- **Atomicity** - exchange of funds in example must happen as an all-or-nothing transaction
- **Consistency** - your database should never generate a report that shows a withdrawal from saving without the corresponding addition to the checking account => block all reporting during atomic operations (=> impact on speed!)
- **Isolation** - each part of the transaction occurs without knowledge of any other transaction
- **Durability** - once all aspects of transaction are complete, it's permanent

- software to handle these rules = very complex (50-60% of the codebase => new databases often do not support database-level transaction management in their first release)
- all transaction control strategies: depend on resource locking
- ACID systems: focus on consistency and integrity of data above all other considerations (temporarily blocking reporting mechanisms is a reasonable compromise to ensure systems return reliable and accurate information) => *pessimistic*

- **BASE**

- **Basic Availability** - information and service capability are “basically available” (you can *always* generate a report)
- **Soft-state** - some inaccuracy is temporarily allowed and data may change while being used to reduce the amount of consumed resources
- **Eventual consistency** - eventually, when all service logic is executed, the systems is left in a consistent state

*e.g.* shopping carts: no problem if back-end reports are inconsistent for a few minutes; it's more important that the customer can actually purchase things



- ACID: focuses on consistency <-> BASE: focuses on availability
- BASE = *optimistic*: eventually all systems will catch up and be consistent
- BASE systems tend to be much simpler and faster <= don't have to write code that deals with locking and unlocking resources
- ACID vs BASE != black vs white; continuum

## ACID

- **Get transaction details right**
- **Block any reports while you are working**
- **Be pessimistic: Anything might go wrong!**
- **Detailed testing and failure mode analysis**
- **Lots of locks and unlocks**



VS.

## BASE

- **Never block a write**
- **Focus on throughput not consistency**
- **Be optimistic: if one service fails it will eventually get caught up**
- **Some reports may be inconsistent for a while but don't worry**
- **Keep things simple and avoid locks**



## 6. **Horizontal scalability** through **database sharding**

- In RDMBS: automatic sharding
- As number of servers grows: higher chance that one will go down => duplicate data to backup or mirrored system = data **replication**

# Types of NoSQL data stores

---

- **document stores:** for storing hierarchical data structures (e.g. document search)
- **column stores:** sparse matrix systems that use a row and column as key
- **graph stores:** for relationship intensive problems (e.g. social network, gene network)
- **key/value stores:** simple data storage systems that use a key to access a value (e.g. image stores, Amazon S3)

# Database Landscape Map – February 2013

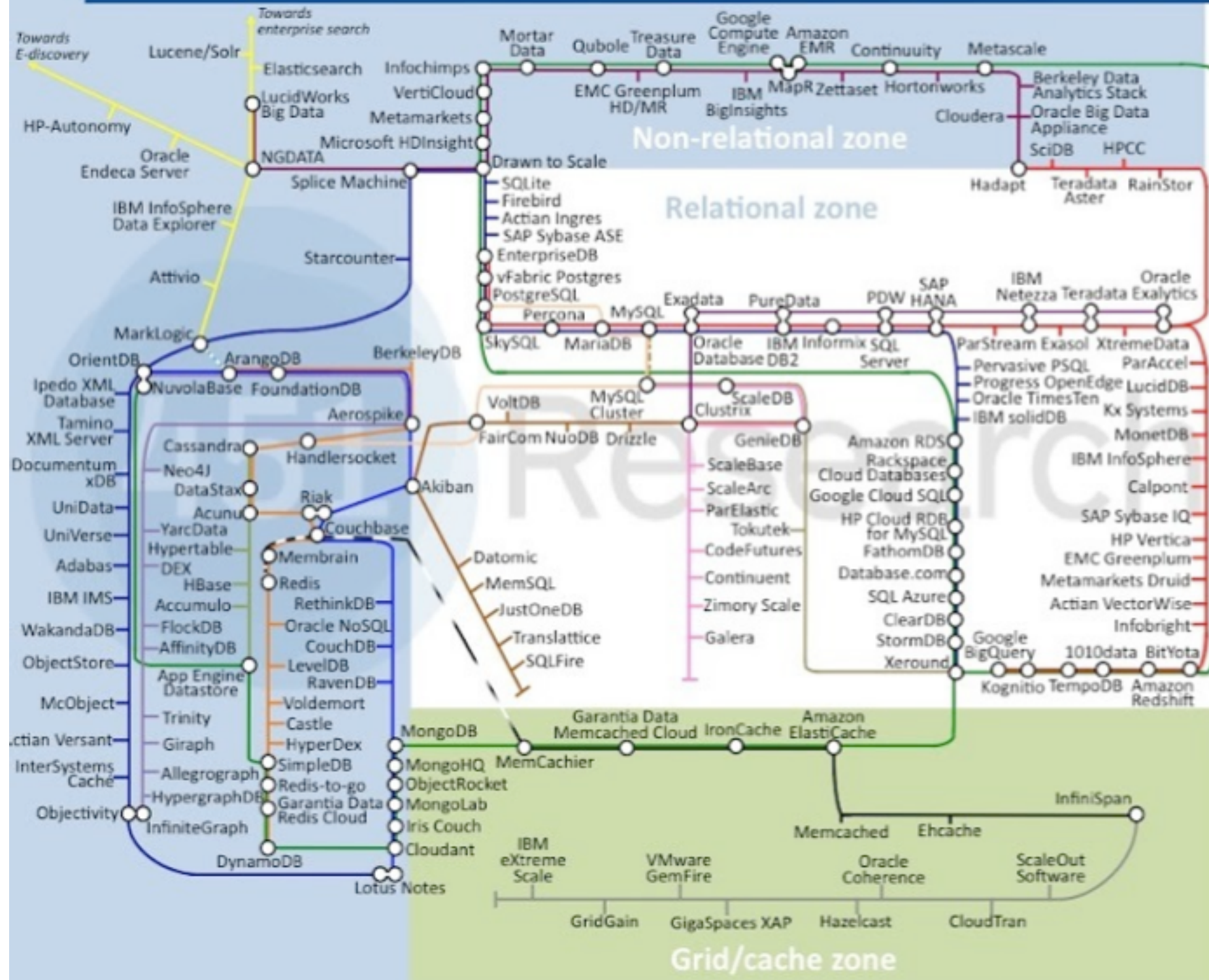
451 Research

## Key:

- Operational
- Analytic
- -as-a-Service
- NoSQL extension
- BigTables
- Graph
- Document
- Key value stores
- Key value direct access
- Hadoop
- NewSQL extension
- Storage engines
- Advanced clustering/sharding
- New SQL databases
- Data caching extension
- Data caching
- Data grid
- Index-based data management
- Appliances

www.451research.com

@maslett



## document stores

- focus on storage of *documents* rather than rows and columns
- some: provide SQL-like interface
- some: provide mapreduce processing

## column stores

- proprietary Google BigTable implementation
- database = multiple tables, each containing addressable rows, each row with a set of values in columns
- important differences with RDBMS:
  - each row can have a different set of columns (but common set of “column groups”)
  - tables are intended to have many ( $>1,000-1,000,000$ ) columns

## graph stores

- use *nodes*, *edges* and *properties* as primary elements
- different underlying infrastructures

## key/value stores

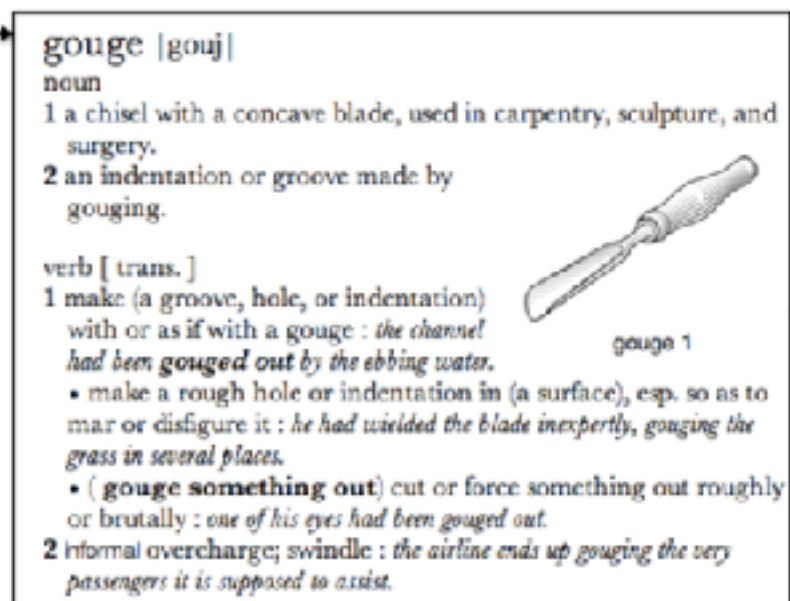
- very specific use
  - provide only a single way to access values (only commands: put, get, delete)
  - not possible to query value
- very simple
- very fast



- simple database that when presented with a simple string (the key) returns an arbitrary large blog of data (the value)
- has no query language; can only put stuff in and get stuff out
- like a dictionary

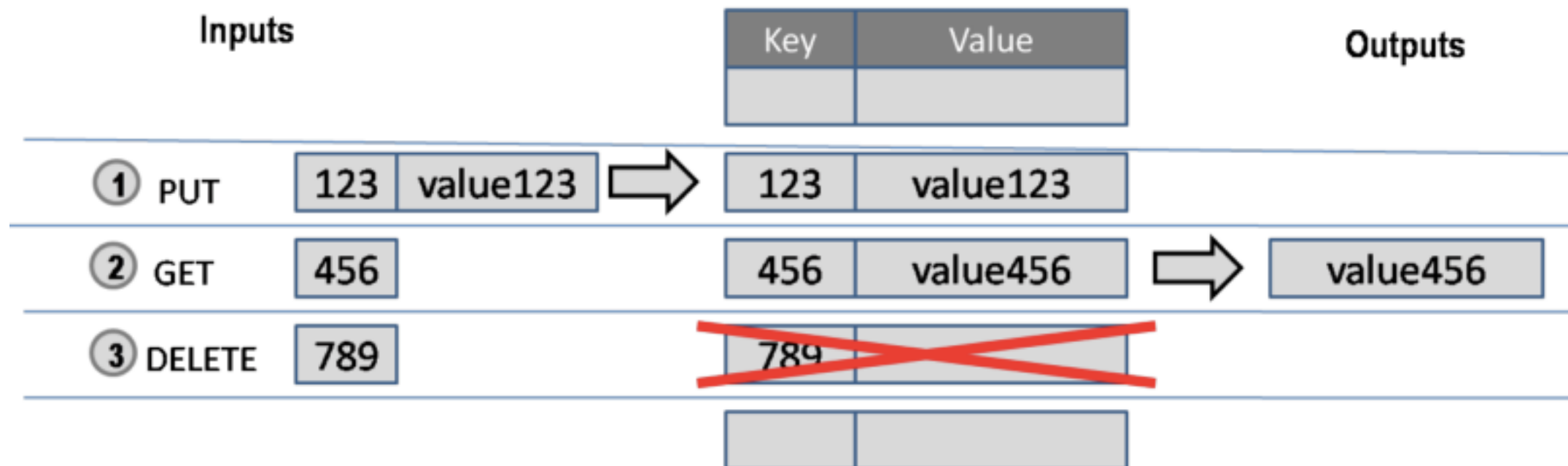
The "key" is just the word "gouge"

The "value" is all the definitions and images



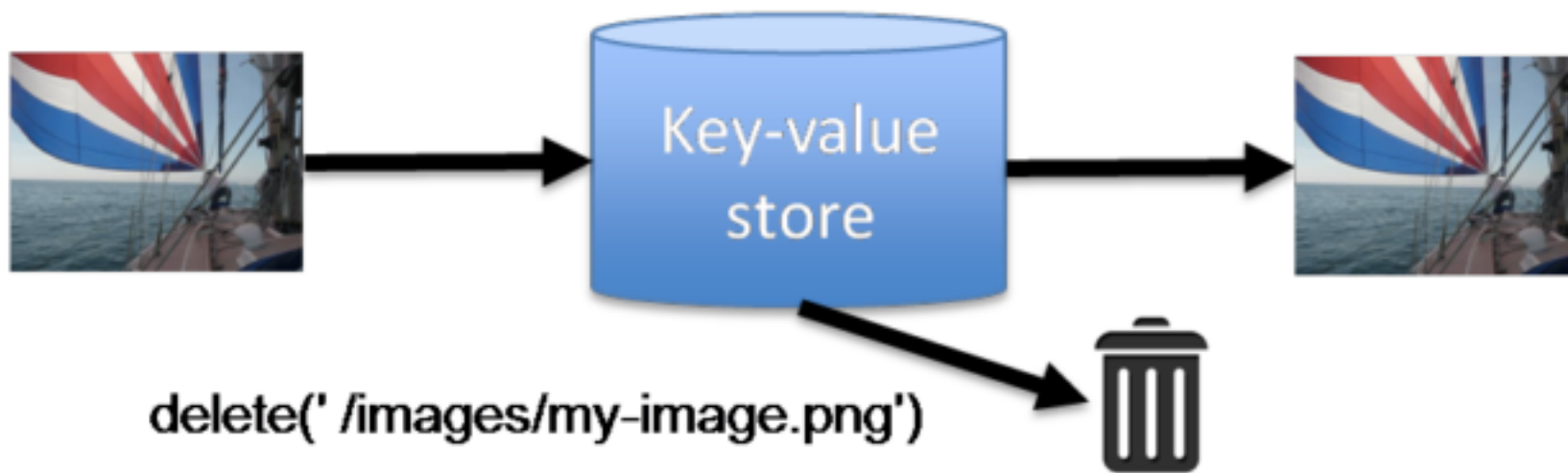
- very scalable: fast retrieval of values regardless of the number of items in your store
- don't have to specify data type for the value => you can store whatever you want
  - system stores the information as a blob (e.g. Amazon S3)
  - up to the application to determine what type of data it is and what to do with it
  - keys can be many things: path names to files, artificially generated strings, SQL queries (as string), ...

- only 3 operations in API: put, get and delete



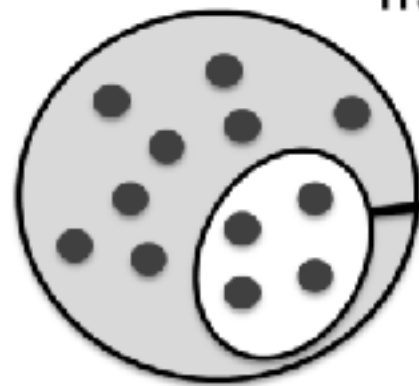
`put(' /images/my-image.png', $image-data)`

`get(' /images/my-image.png')`



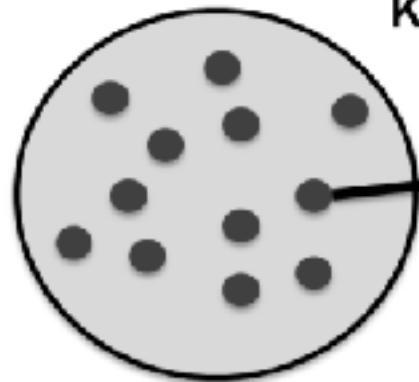
- only 2 rules:
  - distinct keys: you can never have two items with the same key
  - no queries on values: you cannot select a key/value pair using the value
- <-> RDBMS: you can constrain a result set using the WHERE clause

#### Traditional Relational Model



- Result set based on row values
- Value of rows for large data sets must be indexed
- Values of columns must all have the same data type

#### Key-Value Store Model



- All queries return a single item
- No indexes on values
- Values may contain any data type

# Use cases

---

- storing web pages
  - key = URL; value = HTML code of webpage
- Amazon Simple Storage Service (S3)
  - uses a simple REST API (see later in course)
  - = key/value store, plus metadata and access control

# Implementations



# Redis

- <http://redis.io>

Redis is an open source (BSD licensed), in-memory **data structure store**, used as database, cache and message broker. It supports data structures such as **strings**, **hashes**, **lists**, **sets**, **sorted sets** with range queries, **bitmaps**, **hyperloglogs** and **geospatial indexes** with radius queries. Redis has built-in **replication**, **Lua scripting**, **LRU eviction**, **transactions** and different levels of **on-disk persistence**, and provides high availability via **Redis Sentinel** and automatic partitioning with **Redis Cluster**.

- <http://try.redis.io>

```
> set server:name "fido"
OK
> get server:name
"fido"
> SET tom '{name: "Tom Willems", age: 47}'
OK
> get tom
"{name: \"Tom Willems\", age: 47}"
```

```
$ sudo pip install redis
```

- Python API

```
>>> import redis
>>> r = redis.StrictRedis(host='localhost', port=6379, db=0)
>>> r.set('foo', 'bar')
True
>>> r.get('foo')
'bar'
```

- Docker:

```
docker run -it --link some-redis:redis --rm redis sh -c 'exec redis-cli -h "$REDIS_PORT_6379_TCP_ADDR" -p "$REDIS_PORT_6379_TCP_PORT"'
```



An example - small scale

# Employee database

---

- Database containing all information about employees: name, address, contract(s), ...
  - One individual can have several subsequent contracts
- Can be kept in relational database (e.g. mysql), but we can still use Lambda Architecture principles
  - key: immutability

# Using *views*

---

ID	name	phone	address
1	Tim Janssen	+32 272618273	St-Jobsweg 143, Berchem
2	Filip Beckaert	+32 918274917	Karmstraat 2, Vissenaken
3	Bart Willard	+32 726291726	Kipdorpstraat 101, St Niklaas
...	...	...	...

individuals

contracts

ID	individual_id	start_date	stop_date
1	1	20141001	20150930
2	1	20151001	20260930
3	2	20140401	20150331
...	...	...	...

```
CREATE VIEW v_current_individuals AS
SELECT i.id, i.name, i.phone, i.address
FROM individuals i
WHERE EXISTS (
    SELECT * FROM contracts c
    WHERE c.individual_id = i.id
    AND c.start_date <= CURDATE()
    AND CURDATE() <= c.stop_date
)
```

v\_current\_individuals

ID	name	phone	address
1	Tim Janssen	+32 272618273	St-Jobsweg 143, Berchem
3	Bart Willard	+32 726291726	Kipdorpstraat 101, St Niklaas
...	...	...	...

Filip Beckaert (ind 2)  
doesn't work here  
anymore

=> every individuals who ever worked at the company stays in the "individuals" table

An example - larger scale

# What do we have? What do we need?

---

- The data: twitter at Belgian scale (= not optimised/tweaked)
  - profile data (username, real name, gender, language, ...)
  - user clicks “follow” on profile of other user
  - actual (re)tweets (timestamp, sender, text)
- The uses
  - key influencers (= those with most retweets)
  - suggest new people to follow
  - what are the trending hashtags?
  - twitter analytics

## 28 day summary with change over previous period



# Batch layer - the ground truth

---

- immutable; add, don't update
  - necessary for profile data?
  - can e.g. be regular csv files

## profile

name, twitter handle, gender, language

Tim Janssen;tjanssen;male;Dutch  
Filip Beckaert;beck;male;Dutch  
Bart Willard;bartw;male;English  
...

## followers

timestamp, twitter handle, object, action

20150105T0345;tjanssen;bartw;follow  
20150105T0532;tjanssen;beck;follow  
20150512T1917;tjanssen;sanders;follow  
20150512T1921;tjanssen;sanders;unfollow  
20150512T2331;bartw;tjanssen;follow  
...

## tweets

timestamp, twitter handle, text

20150105T0348;tjanssen;"Just followed @bartw"  
20150105T0529;bartw;"Moules-frites.. The best..."  
20150512T1924;tjanssen;"Is anyone going to #tedxbrussels?"  
20150513T0815;bartw;"#crispr now identified in viruses as well"  
...



# Serving layer

- convert data in batch layer into form that can answer the uses

- **Twitter analytics**

*One per month: run over data  
in batch layer and count =>  
store in key-value store*

```
tjanssen_201602  
=> {tweets: 14,  
    impressions: 10391,  
    visits: 591,  
    mentions: 12,  
    new_followers: 16}  
bartw_201602  
=> ...
```

## FEB 2016 SUMMARY

Tweets

14

Tweet impressions

10.3K

Profile visits

561

Mentions

12

New followers

16

## 28 day summary with change over previous period

Tweets

13 ↑44.4%

Tweet impressions

11.3K ↑30.7%

Profile visits

533 ↓9.2%

Mentions

11 ↑37.5%

Followers

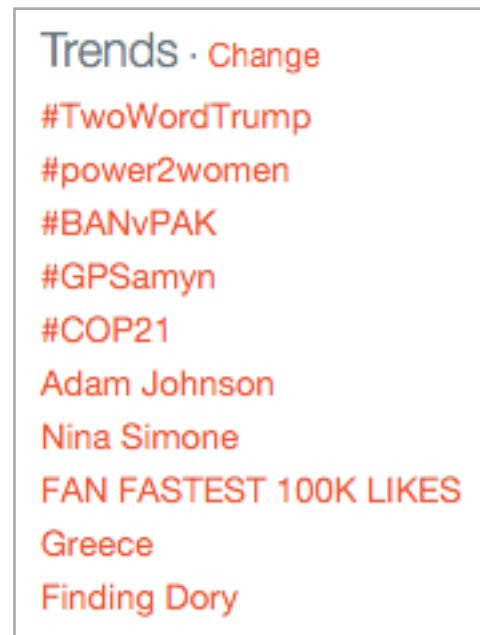
2,149 ↑14

- trending hashtags

```
20150105T0348;tjanssen;"Just followed @bartw"  
20150105T0529;bartw;"Moules-frites... The best..."  
20150512T1924;tjanssen;"Is anyone going to #tedxbrussels?"  
20150513T0815;bartw;"#crispr now identified in viruses as well"  
...
```



```
Greece;2837  
Nina Simone;5922  
#crispr;1998  
#GPSamin;9283  
#TwoWordTrump;27102  
#tedxbrussels;2692  
...
```



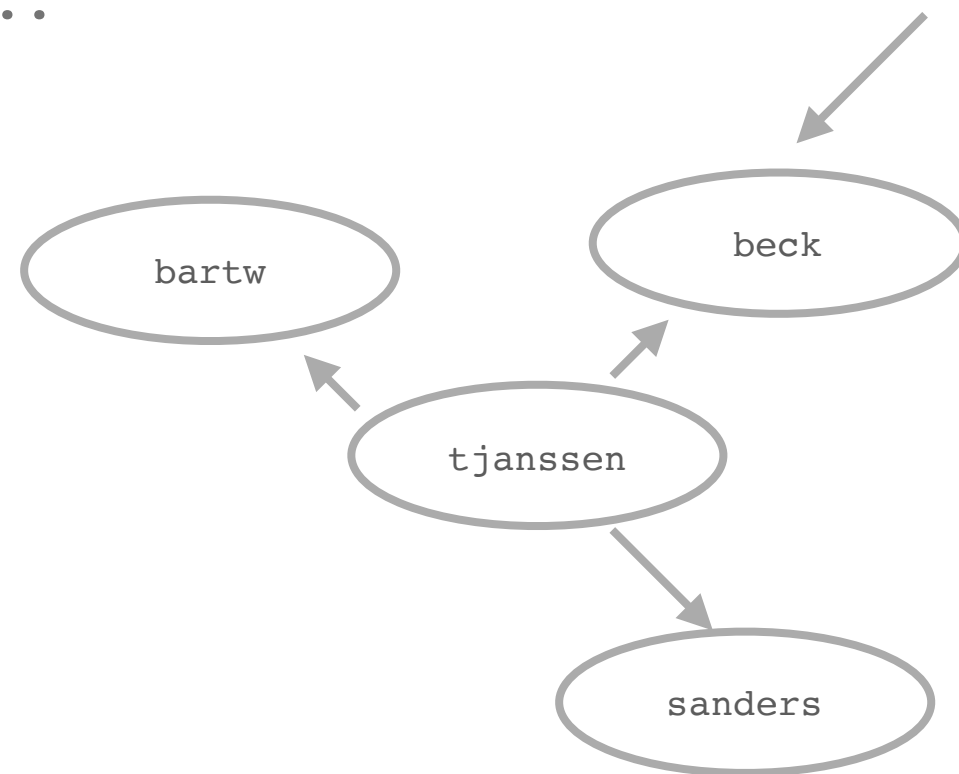
*Continuously:* run over data in batch layer,  
count and sort;

if takes too long: include speed layer  
(either same computation as serving but  
only on new data, or completely different  
incremental algorithm)

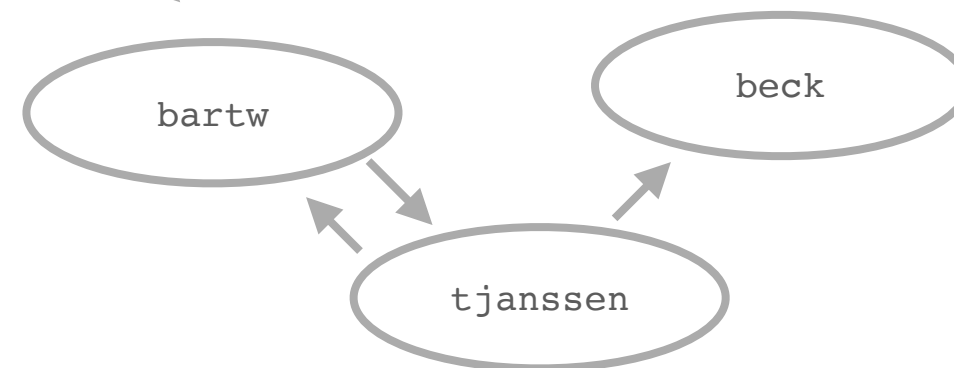
- **suggest new people to follow**  
**who are key influencers?**  
**who are more popular?**

*Continuously:* run over data in batch layer, create nodes and links in graph

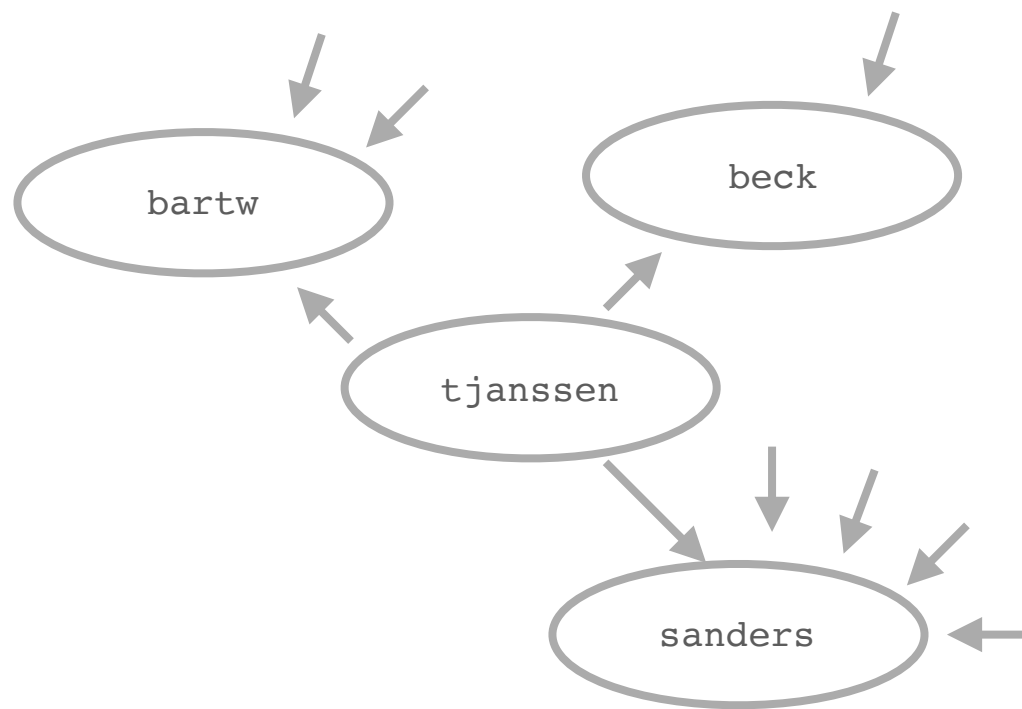
```
20150105T0345;tjanssen;bartw;follow
20150105T0532;tjanssen;beck;follow
20150512T1917;tjanssen;sanders;follow
20150512T1921;tjanssen;sanders;unfollow
20150512T2331;bartw;tjanssen;follow
...
```



depending on when process is run



in graph database (e.g. Neo4j)



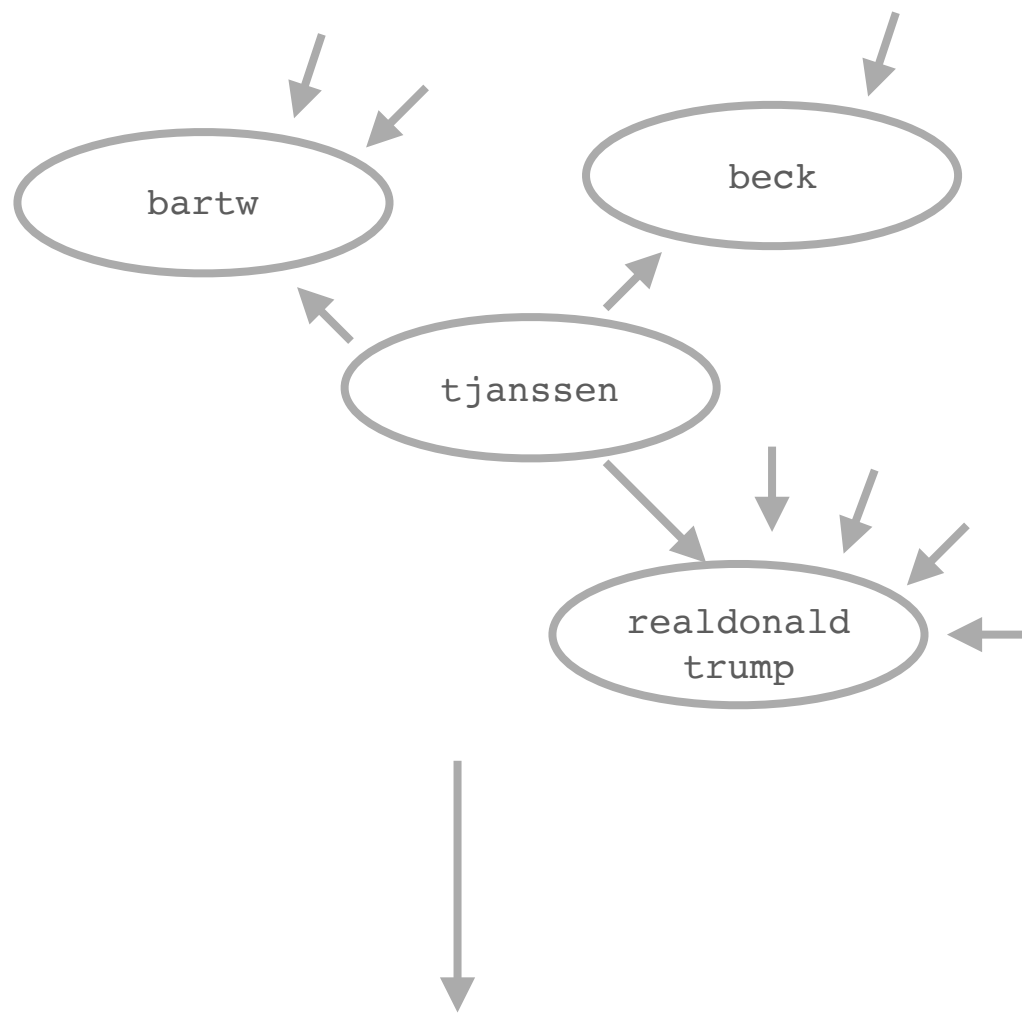
- who are more popular?

tjanssen => 1,928  
sanders => 6,570,000  
bartw => 3,102  
beck => 2,281  
...

in key/value store?  
cannot sort by value

tjanssen;1928  
sanders;6570000  
bartw;3102  
beck;2281  
...

in file?



in file?

tjanssen;24918  
 sanders;87102817  
 bartw;28172  
 beck;49812  
 ...

- **who are key influencers?**

e.g. defined as how many followers in three steps?

(easy when using graph database; see later)

in SQL database?

handle	influences
tjanssen	24918
sanders	87102817
bartw	28172
beck	49812
...	...