CDS6214

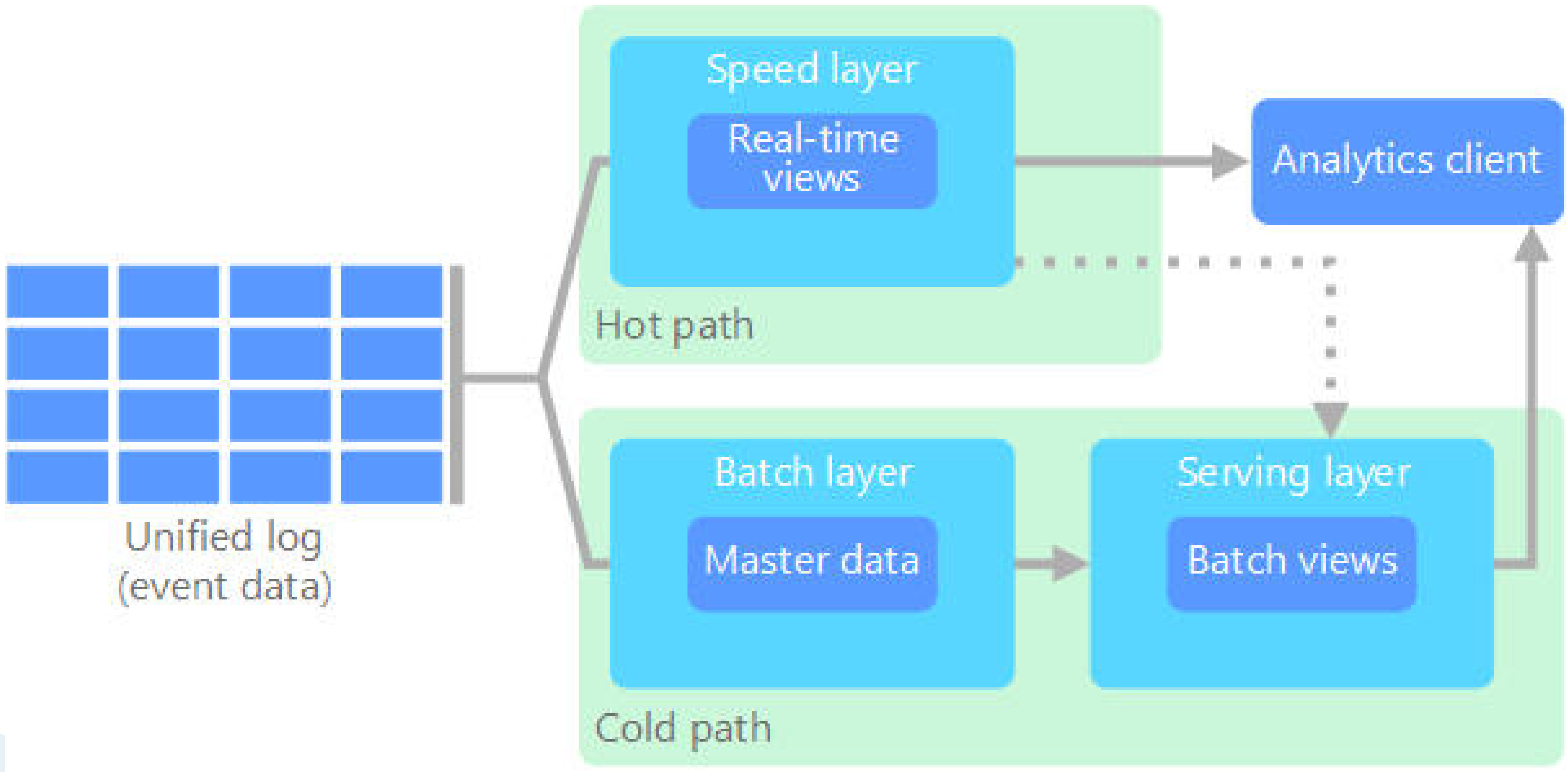
Data Science Fundamentals

Lecture 9-10

Big Data Architecture (Part 2)

Lambda Architecture

Lambda Architecture (LA)



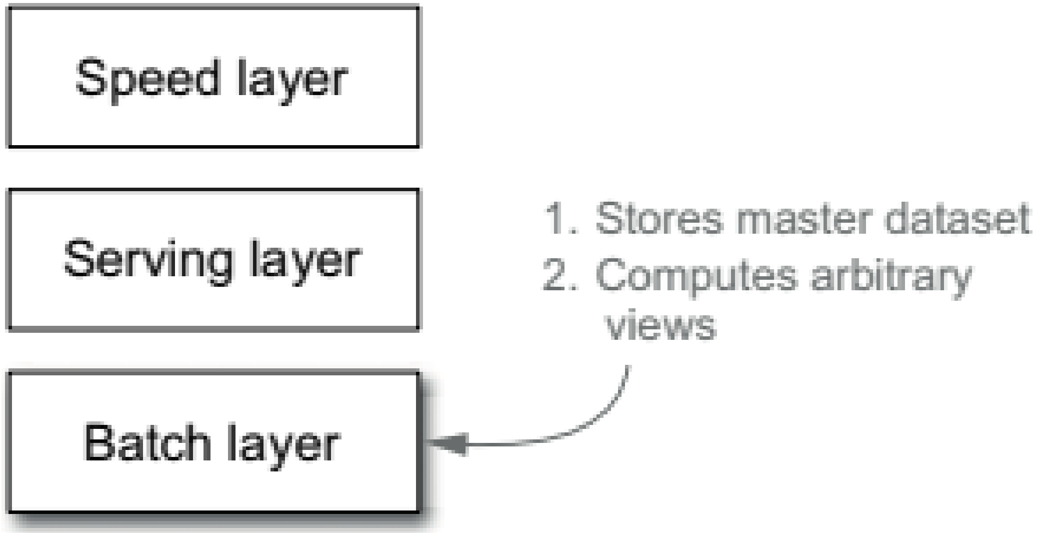
https://learn.microsoft.com/en-us/azure/architecture/databases/guide/big-data-architectures

http://lambda-

Lambda Architecture (Batch Layer)

Batch layer

- The batch layer stores the **master copy of the dataset** and pre-computes batch views on that master dataset
- Must be able to (1) store an immutable, ever-growing dataset and (2) pre-compute arbitrary functions on the data

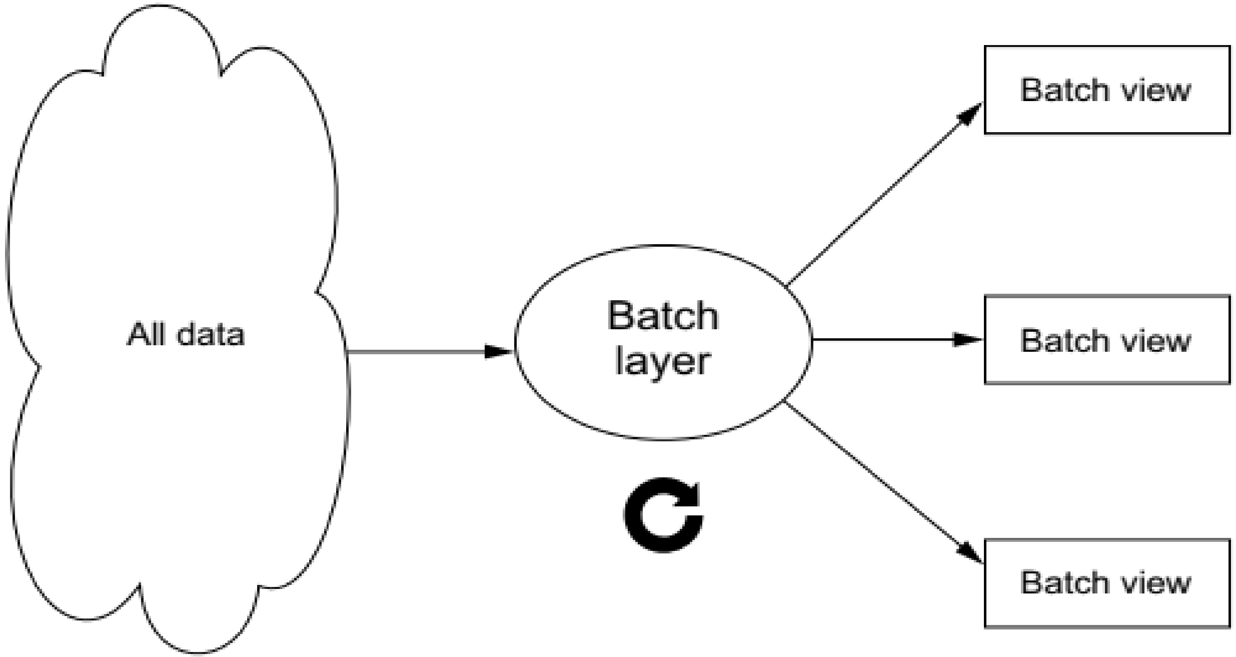


The master dataset can be thought of as a very large list of records

- Basis of LA is that ***query=function(all data)*** not feasible because "all data" may be HUGE!
- So lets pre-compute the function as views of the data, i.e. batch view

batch view = function(all data)

Therefore, ***query = function(batch view)***



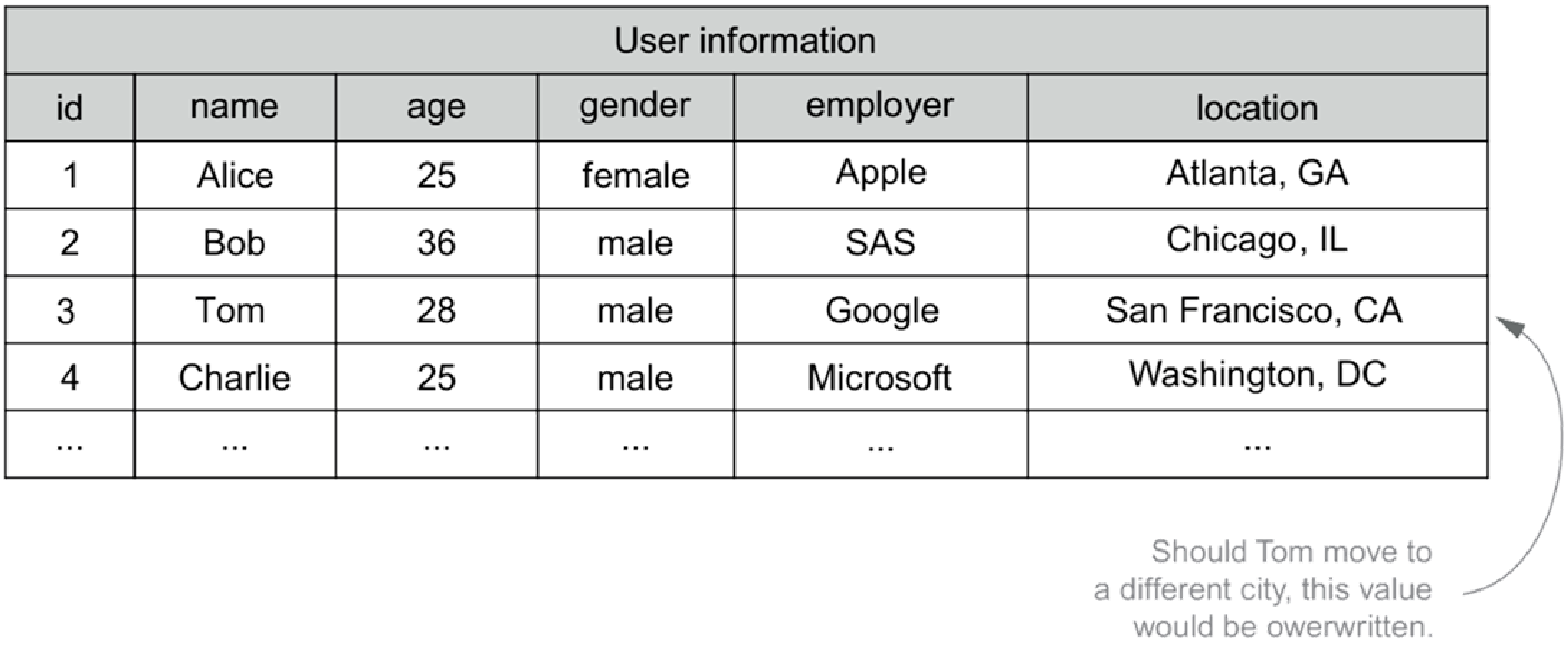
The precomputed view is indexed so that it can be accessed with random reads.

- The master dataset is the only part of the Lambda Architecture that absolutely must be safeguarded from corruption. Overloaded machines, failing disks, and power outages all could cause errors, and human error with dynamic data systems is an intrinsic risk and inevitable eventuality.

- Data is kept immutable(i.e. unchanging)

- Advantages:
  - Human-fault tolerance– earlier good data is still available
  - Only append function is needed

Importance of immutability



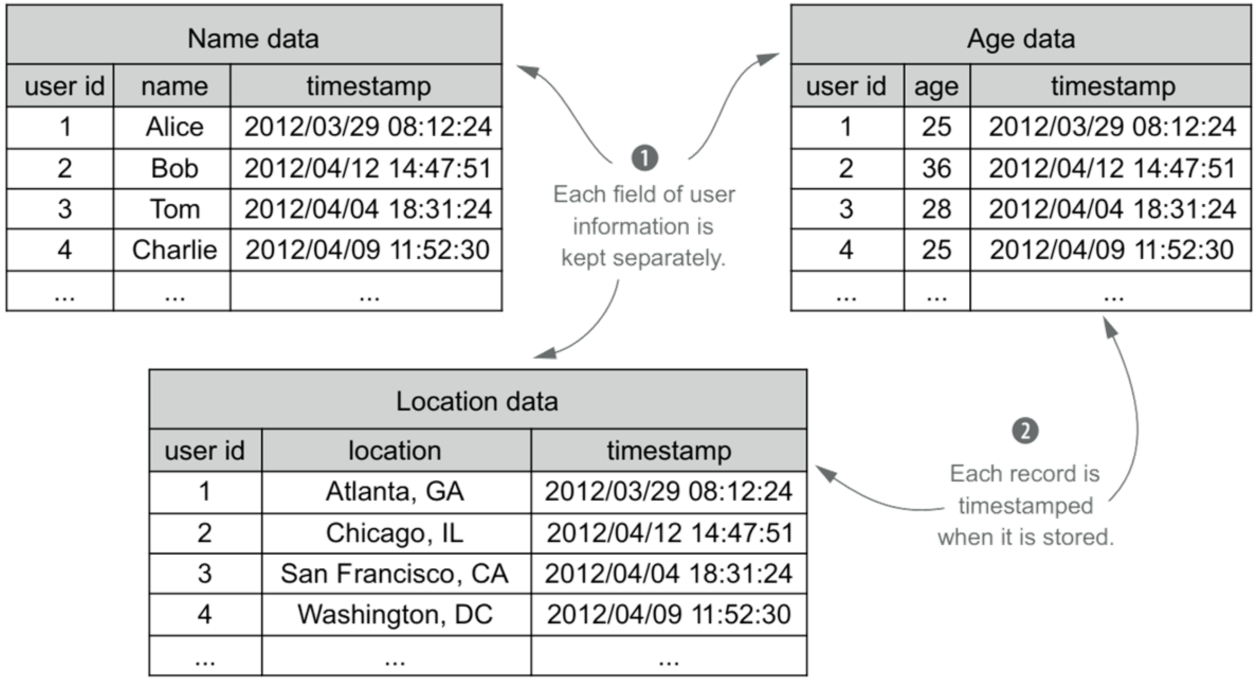
User information					
id	name	age	gender	employer	location
1	Alice	25	female	Apple	Atlanta, GA
2	Bob	36	male	SAS	Chicago, IL
3	Tom	28	male	Google	San Francisco, CA
4	Charlie	25	male	Microsoft	Washington, DC
...	...	...	...	...	...

Should Tom move to
a different city, this value
would be owerwritten.

How do we retain this information without having to normalize the table?

Source: Data Management course

Importance of immutability



Name data

user id	name	timestamp
1	Alice	2012/03/29 08:12:24
2	Bob	2012/04/12 14:47:51
3	Tom	2012/04/04 18:31:24
4	Charlie	2012/04/09 11:52:30
...	...	...

Age data

user id	age	timestamp
1	25	2012/03/29 08:12:24
2	36	2012/04/12 14:47:51
3	28	2012/04/04 18:31:24
4	25	2012/04/09 11:52:30
...	...	...

Location data

user id	location	timestamp
1	Atlanta, GA	2012/03/29 08:12:24
2	Chicago, IL	2012/04/12 14:47:51
3	San Francisco, CA	2012/04/04 18:31:24
4	Washington, DC	2012/04/09 11:52:30
...	...	...

1 Each field of user information is kept separately.

2 Each record is timestamped when it is stored.

- To create an immutable data storage model, create a separate record every time a user’s information evolves. Accomplishing this requires two changes:
  - Track each piece of user information in a separate table
  - Tie each unit of data to a moment in time when the information is known to be true (i.e. timestamp)

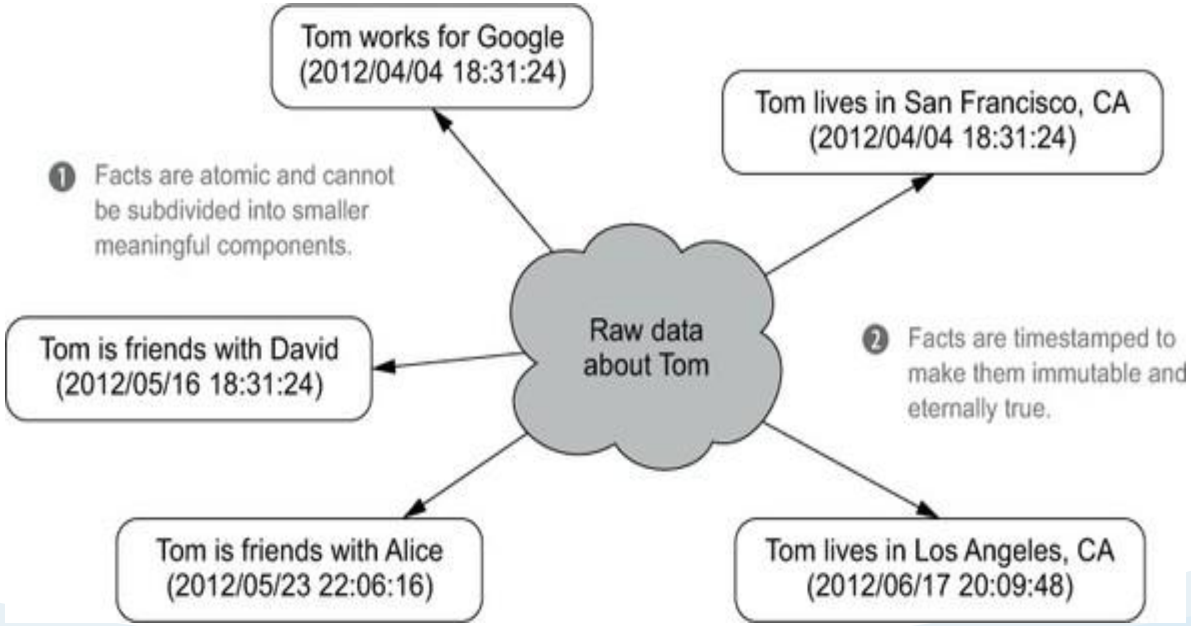
Source: Data Management course

How do we represent the data in the master dataset ?

- Traditional relational tables
- Structured XML
- Semi-structured JSON documents
- Fact-based model

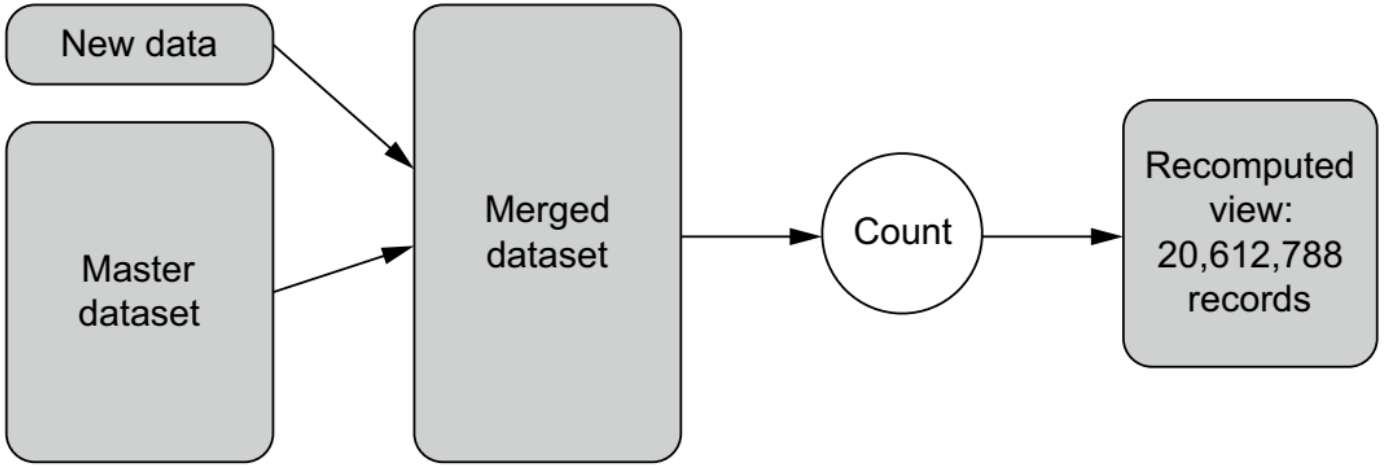
Example of Fact-based model

- Facts are atomic because they can’t be subdivided further into meaningful components.
- As a consequence of being atomic, there’s no redundancy of information across distinct facts. Facts having timestamps should come as no surprise, given our earlier discussion about data—the timestamps make each fact immutable and eternally true.


Tom works for Google (2012/04/04 18:31:24)
Tom lives in San Francisco, CA (2012/04/04 18:31:24)
1 Facts are atomic and cannot be subdivided into smaller meaningful components.
Tom is friends with David (2012/05/16 18:31:24)
Raw data about Tom
2 Facts are timestamped to make them immutable and eternally true.
Tom is friends with Alice (2012/05/23 22:06:16)
Tom lives in Los Angeles, CA (2012/06/17 20:09:48)

Recomputation/Incremental algos

- As the master dataset is changed, a strategy is needed on when/how to update the batch views
  - Recomputation algorithm – throw away old batch views and perform recomputation on entire updated master dataset
  - What are the advantage(s) and disadvantage(s) of this strategy?



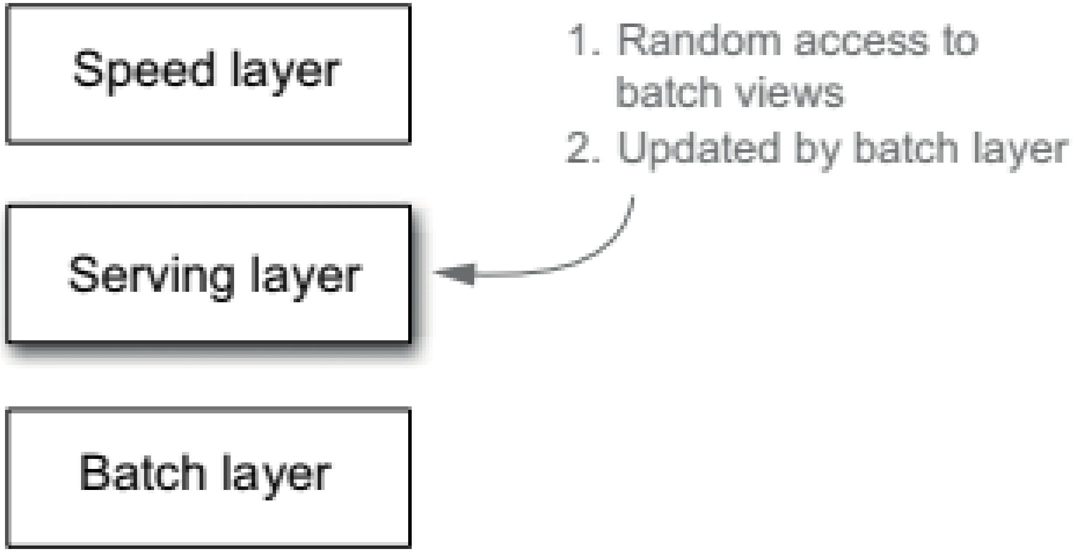
Concluding the Batch Layer

The batch layer pre-computes results using a distributed processing system that can handle very large quantities of data. The batch layer aims at perfect accuracy by being able to process all available data when generating views. This means it can fix any errors by recomputing based on the complete data set, then updating existing views. Output is typically stored in a read-only database, with updates completely replacing existing pre-computed views The batch view is created by running batch jobs on the raw data stored in a distributed file system, such as Hadoop or S3

Lambda Architecture (Serving Layer)

Serving layer

- Output from batch layer feeds the serving layer to be queried upon. It is actually the last component in the batch section in Lambda architecture.
- The serving layer is a specialized distributed database that loads in a batch view and makes it possible to do random reads on it when batch views are updated, the serving layer then updates its results

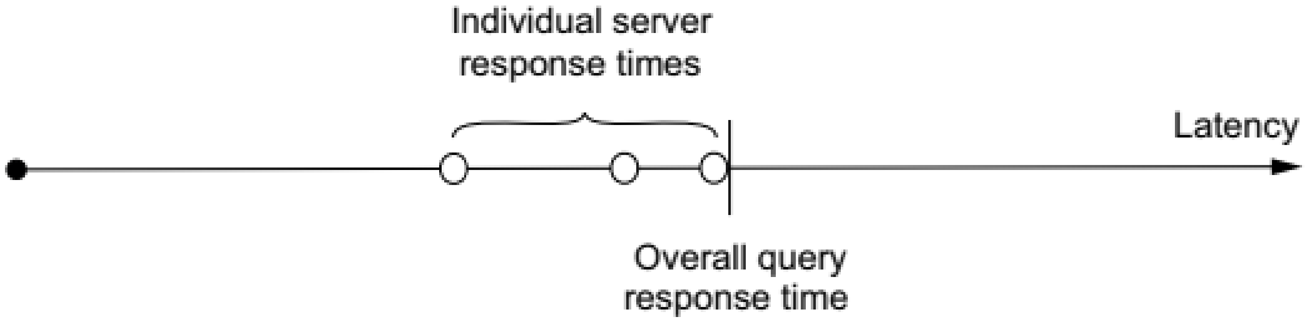


Serving Layer Indexing Strategies

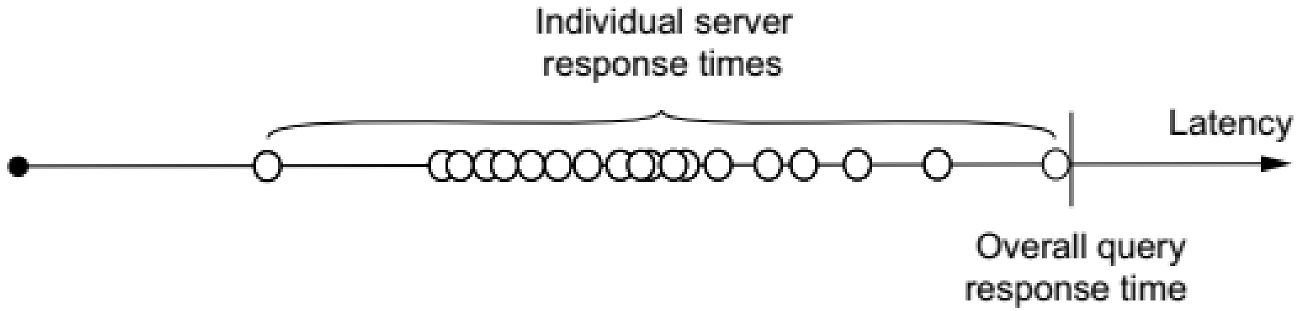
- The serving layer is usually distributed across multiple machines similar to the batch layer indexes are also distributed as they are created, loaded and served on each machine involved
- Because of this indexing must take into account
  - Latency – time required to answer/satisfy a query
  - Throughput – number of queries served in a given period
- Data is distributed and processed across multiple scalable machines so retrieval would also involve fetching from multiple machines

Serving Layer Latency

- Latency due to distributed operations can be high – this is because server response times may vary
    - one server may be slightly more loaded than the others
    - another may be performing garbage collection at the time.
- The latency is further increased when the operations ‘touch’ more servers as the likelihood (i.e. probability) of encountering a slow server also increases



Query that involves 3 servers



Query that involves 20 servers

Serving queries usually involves data I/O – typically disk seeks and reads

Improving Latency/Throughput

- Different/better indexing strategies may improve the latency and throughput characteristics
- E.g.
  - Storing contiguous data in sequential order
  - The same data is also stored on the same partition/block to reduce the number of servers/machines involved in the query to reduce the variance in response time

Serving Layer Requirements

- Lambda architecture places a set of requirements on the serving layer database
  - Batch writable
    Batch views for the serving layer are produced from scratch. It should be possible to replace an old view version with a new one when new data is made available at the batch layer.

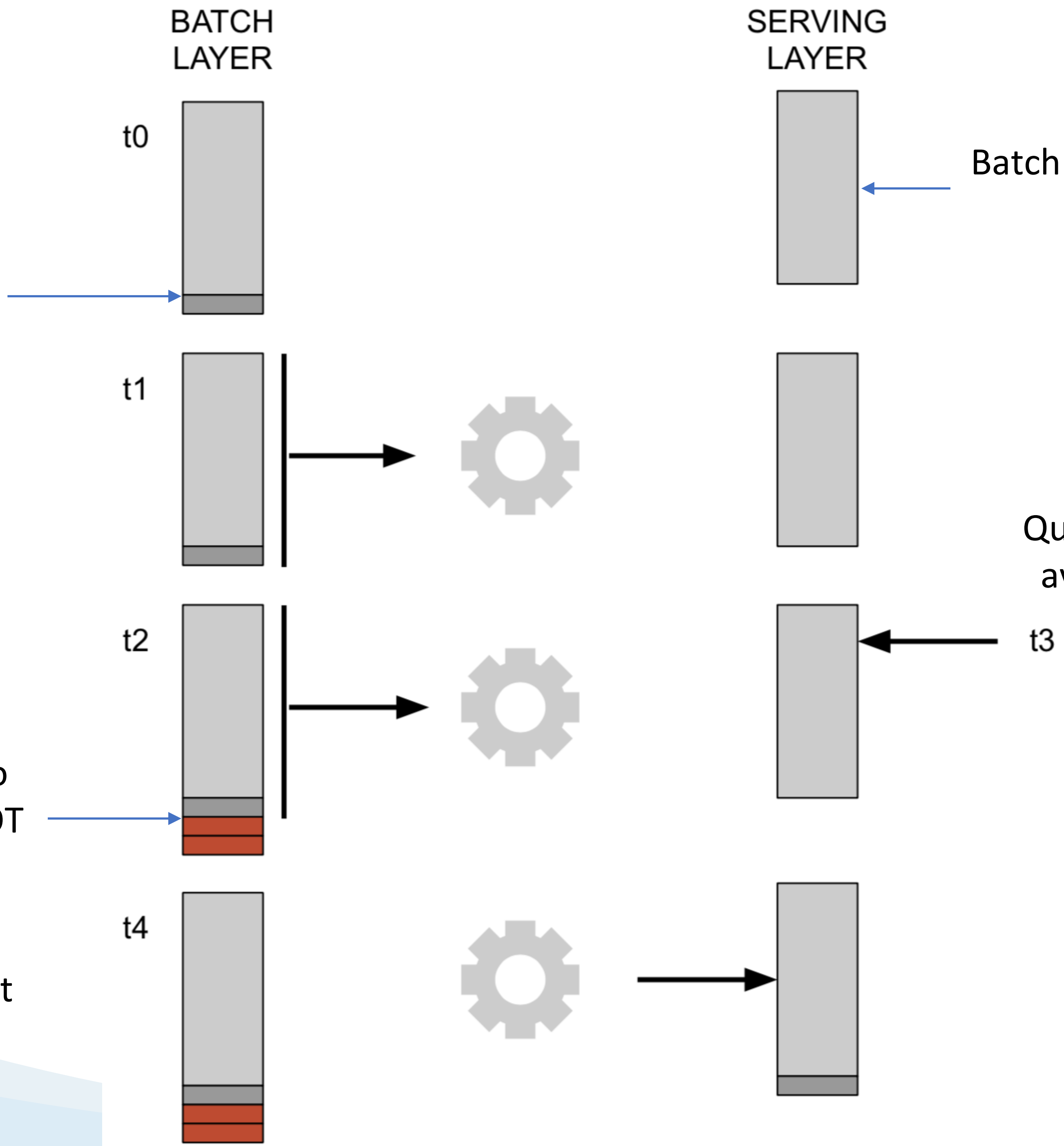
  - Scalable
    The serving layer database should be able to handle database of any arbitrary size. Similar to distributed filesystems and processing discussed in earlier lectures, the database should also support distributed architecture

- Random reads
  Serving layer database must support random reads with indexes providing direct access to small portions of the view. This is in order to support low latency queries

- Fault-tolerance
  Because of its distributed nature, fault tolerance should be supported in the database to avoid failures

Serving Layer Problems

- Most notably, any calculation for a batch view only takes into consideration the data available at that point of time of recomputation – any data added afterwards is not in the batch view generated
    - To update the batch views, recomputations are performed – which brings up a new problems. When to recompute? What to recompute? How to recompute?
- The query may not always contain the latest set of data – especially if it is being updated by a separate agent/function.



New data added to batch layer

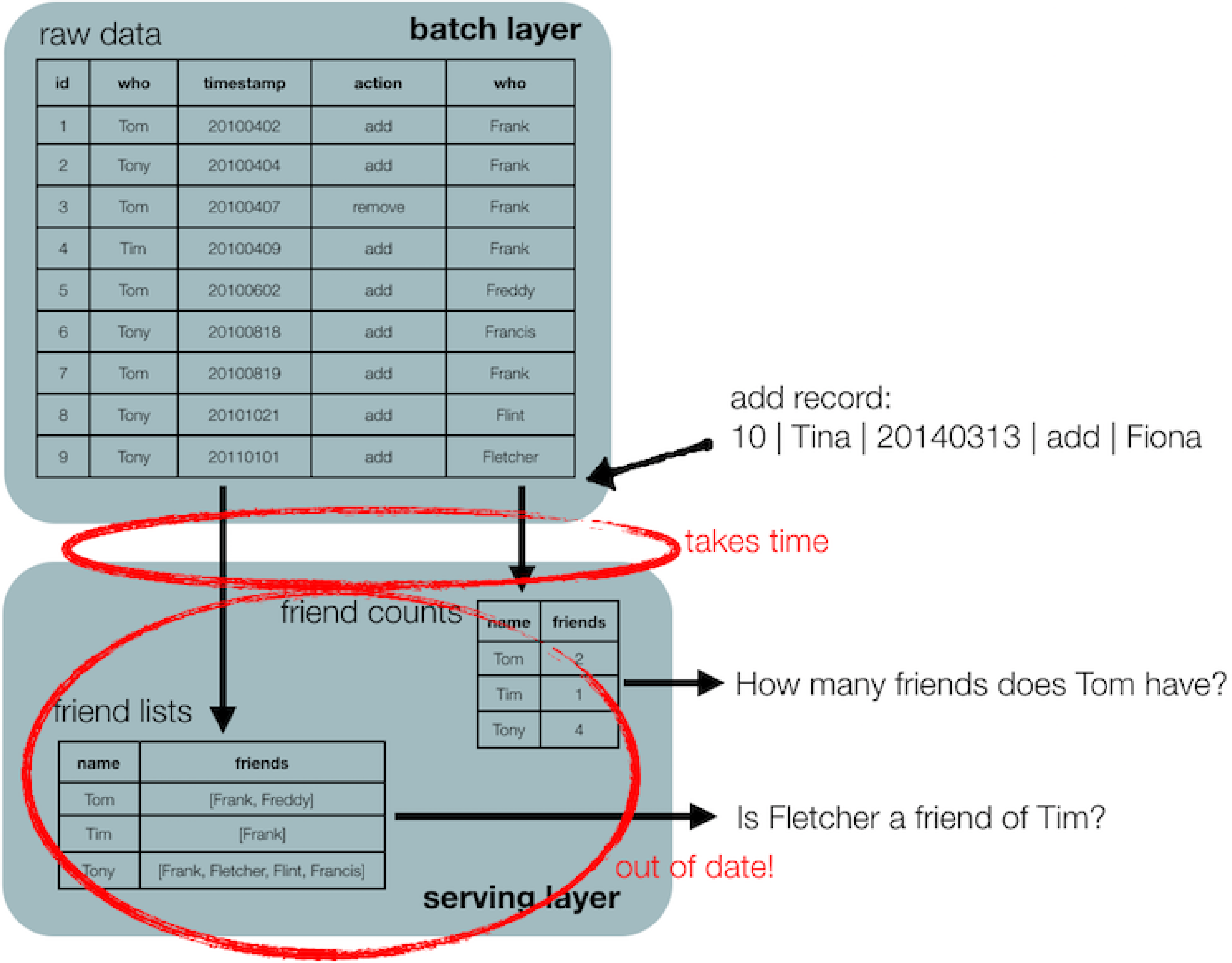
Batch view at serving layer available at t0

Recompute is triggered (automatically/manually)

Query at t3 uses batch view available which is original dataset from t0

Additional data is added to batch view (orange) but NOT part of recompute at t1

Batch layer updates view at serving layer



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

takes time

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?

out of date!

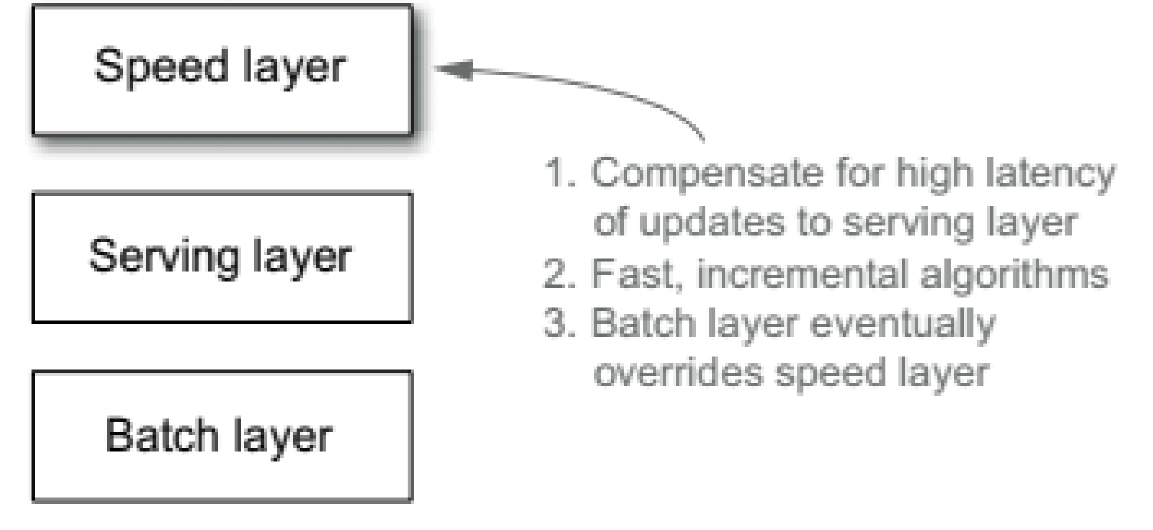
serving layer

Lambda Architecture (Speed Layer)

Speed layer

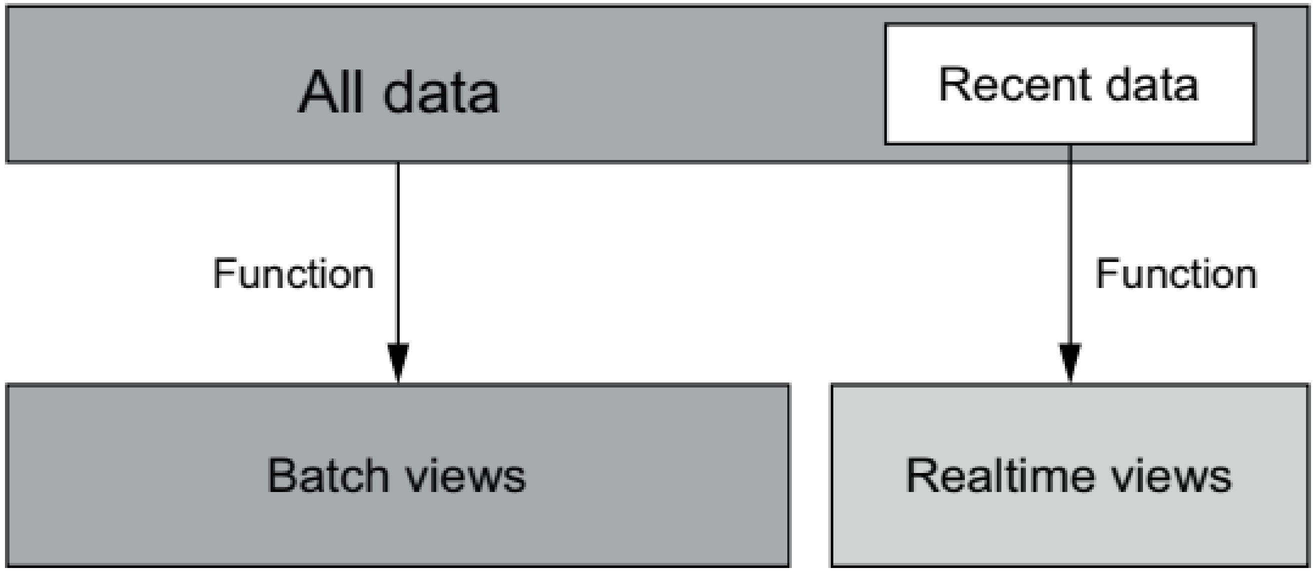
- Previous two layers usually process 'old' data – what about incoming new data? → this is the responsibility of the speed layer
- To ensure new data is represented in query functions as quickly as needed for the application requirements
- We can represent the speed layer with the following equation:

realtime view = function(realtime view, new data)


Speed layer
Serving layer
Batch layer
1. Compensate for high latency of updates to serving layer
2. Fast, incremental algorithms
3. Batch layer eventually overrides speed layer

Speed layer

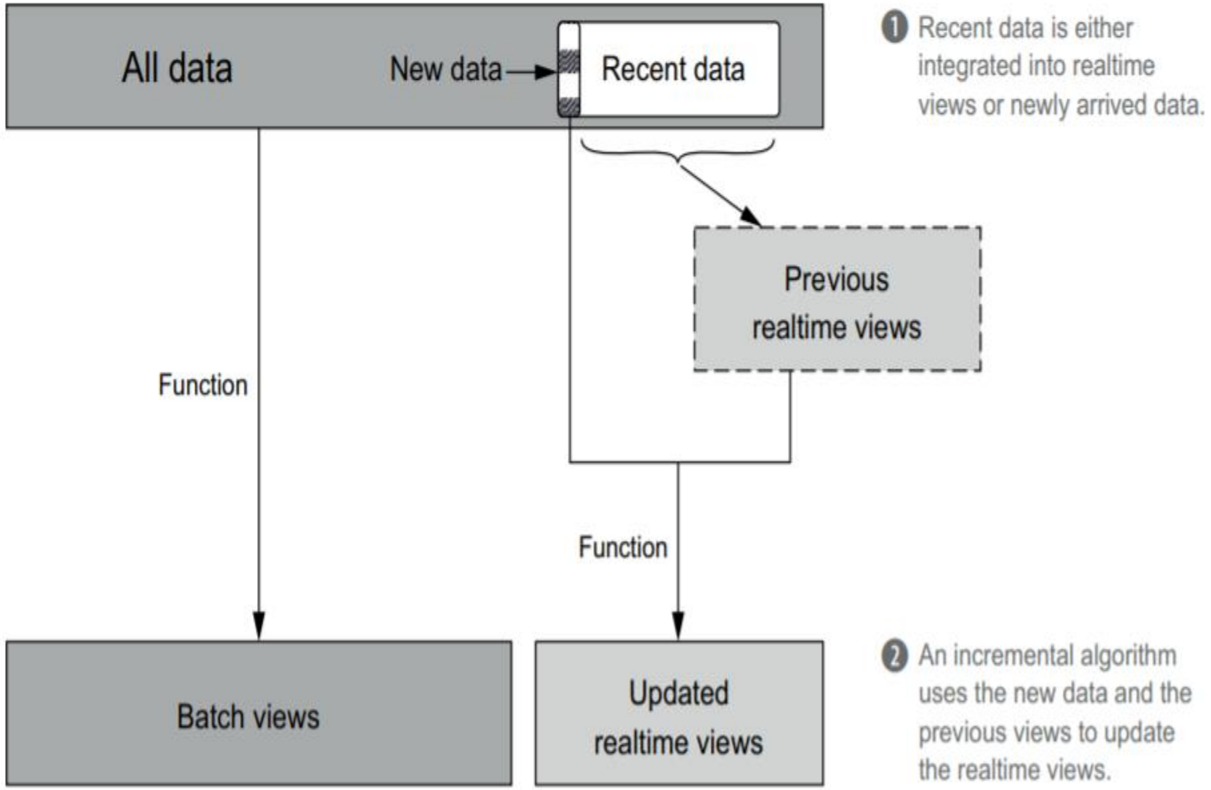
- You can think of the speed layer as being similar to the batch layer in that it produces views based on data it receives.
- The speed layer only looks at recent data, whereas the batch layer looks at all the data at once.



A simple strategy mirrors the batch/serving layer and computes the realtime views using all recent data as input.

Speed layer

- The speed layer does incremental computation instead of the recomputation done in the batch layer



1 Recent data is either integrated into realtime views or newly arrived data.

2 An incremental algorithm uses the new data and the previous views to update the realtime views.

Speed Layer

- The beauty of the Lambda Architecture is that once data is absorbed into the serving layer, the realtime views *are no longer needed*
  - This property of the Lambda Architecture is called *complexity isolation*, meaning that complexity is pushed into a layer whose results are only temporary. If anything ever goes wrong, you can discard the state for the entire speed layer, and everything will be back to normal within a few hours.

Speed layer

- Note:
  - Speed layer is only responsible for data NOT in the master dataset making it vastly smaller
  - The precomputed views in the speed layer are transient and once delivered to the serving layer it is deleted
- So even though the speed layer is more complex and more prone to errors, the errors are short-lived and ultimately corrected by batch layer

Computing realtime views

- Objective of speed layer is similar to the other two layers - to produce views that can be efficiently queried
- Difference is the scope – views only represent recent data and are updated shortly after new data arrives (anywhere between a few milliseconds to few seconds) – which varies according to application
- How do you determine this scope duration? Not an easy thing to do …. ☹
- Let’s see an example of a speed layer that produces views by running a function over all recent data

Example

- Suppose you have a system that collects 32GB of data every day and the data takes 6 hours to be incorporated into the serving layer
  - Speed layer therefore needs to process at least 6 hours of data for the serving layer = 32 GB of data
  - 32 GB is not very huge (in comparison) but affects performance when you want a system to achieve sub-second latency
  - If the average size of a data unit is 100B, the 32 GB of data at the speed layer equates to ~YY units that need processing every 6 hours!
  - Can possibly be reduced by performing a similar 'batch' update in the speed layer at the expense of latency but that defeats the purpose of a 'speed' layer

Storing realtime views

- To serve the speed layer requirements (which are quite demanding), the storage has to provide
  - Random reads – realtime views should support fast random reads to answer queries quickly. This means data has to be indexed
  - Random writes – to support incremental algorithms, it must be possible to modify a realtime view with low latency
  - Scalability – realtime views should scale easily with the amount of data stored and the read/write rates required by the applications
  - Fault tolerance – realtime views should continue to function in the face of failure and this is usually provided through replication
- The decision on how/where/what to do in the speed layer depends on understanding about
  - Eventual accuracy
  - Amount of state store

Eventual (Delayed) Accuracy

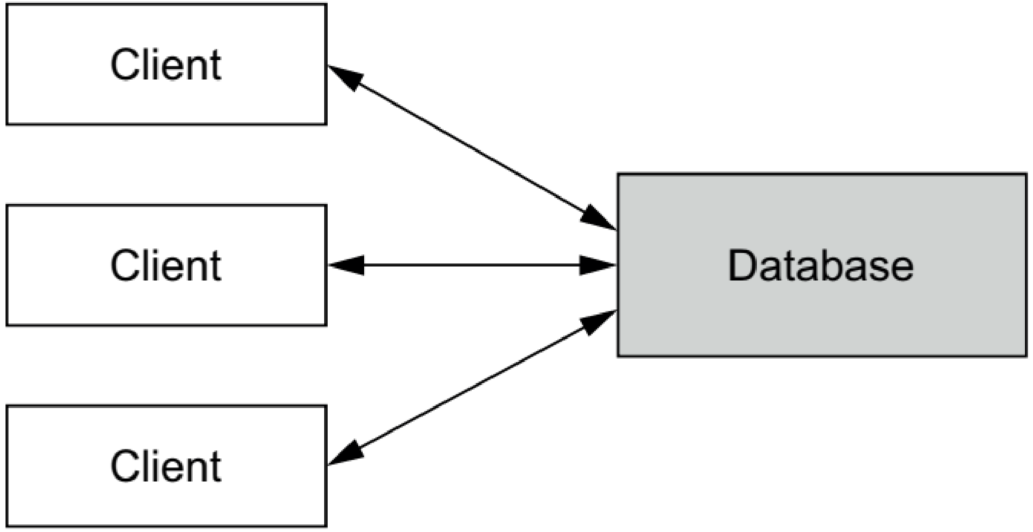
- Remember that the goal of the speed layer is to provide low-latency, real-time views of the most recent data, the data that the batch views have yet to take into account.
- The batch layer strives for exact computation while the speed layer strives for *approximate* computation. The approximate computation of the speed layer will eventually be replaced by the next set of batch views, moving the system towards “eventual accuracy.”
- Processing the stream in real-time in order to produce views that are constantly updated as new data streams in (on the order of milliseconds) is an incredibly complex task

Amount of *State* in Speed Layer

- Speed layers store relatively small amounts because they only represent views on recent data.
- This is a benefit because realtime views are much more complex than serving layer views
  - Requires indexing – to speed up database queries
  - Online compaction – Compaction is required to reclaim space and this is a resource intensive process
  - Concurrency - databases need to coordinate simultaneous reads/writes to the same value for consistency

Update Synchronicity

- Another thing to take into consideration in the speed layer is whether the realtime views are updated synchronously or asynchronously
- Synchronous
  - application issues a request directly to the database and blocks until the update is processed
  - are fast because they communicate directly with the database, and they facilitate coordinating the update with other aspects of the application
- Asynchronous
  - update requests are placed in a queue with the updates occurring at a later time
  - slower than synchronous updates because they require additional steps before the database is modified



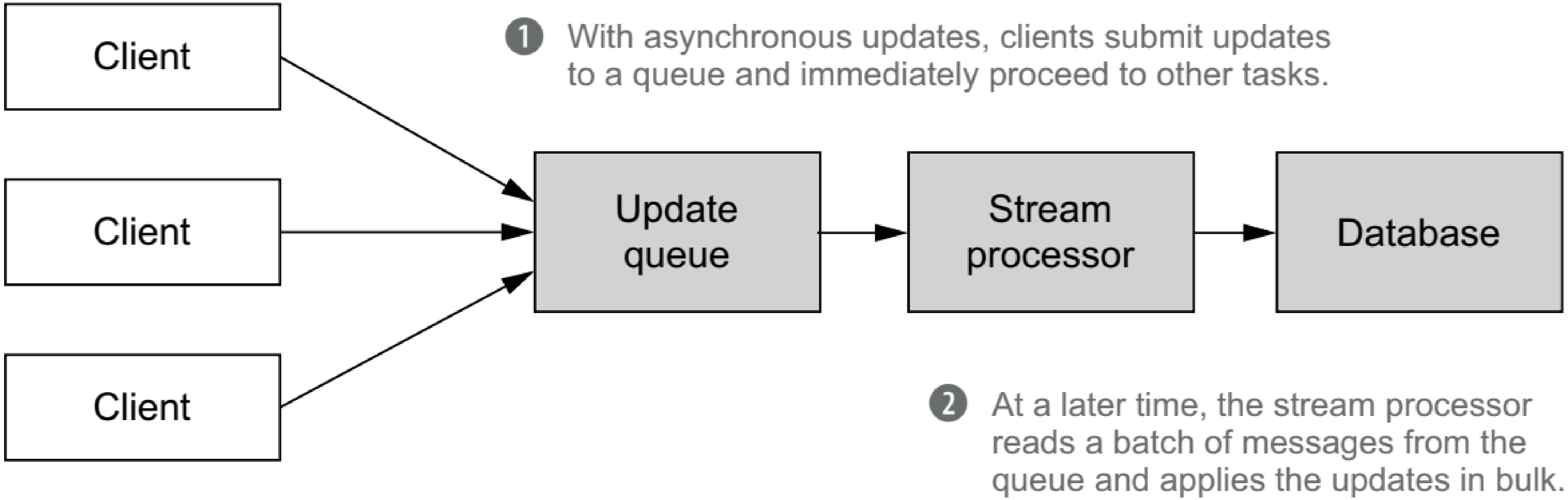
Client

Client

Client

Database

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



Client

Client

Client

Update queue

Stream processor

Database

❶ With asynchronous updates, clients submit updates to a queue and immediately proceed to other tasks.

❷ At a later time, the stream processor reads a batch of messages from the queue and applies the updates in bulk.

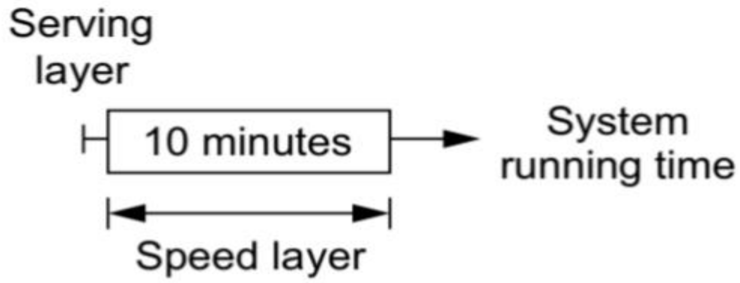
- Because async updates are slower and have additional steps, it is difficult to coordinate across multiple nodes but it does provide some advantages
    - Multiple messages from the queue can be read to perform batch updates to the database, greatly increasing throughput
    - The queue buffers allow for traffic spikes from incoming data thus allowing the system to handle varying loads
- There are uses for both synchronous and asynchronous updates
    - Synchronous updates are typical among transactional systems that interact with users and require coordination with the user interface.
    - Asynchronous updates are common for analytics-oriented workloads or workloads not requiring coordination.
    - The architectural advantages of asynchronous updates—better throughput and better management of load spikes—suggest implementing asynchronous updates unless you have a good reason not to do so

Realtime validity

- Last but not least, how long do we keep realtime views in the speed layer?
  - Speed layer views only need to represent the data that is yet to be processed by the batch computation in batch layer
  - Once a batch computation run finishes, you can then discard a portion of the speed layer views (the parts now absorbed into the serving layer) but obviously you must keep everything else.
- Ideally the speed layer should provide support to directly expire entries but this is not a typical operation of current databases
  - Most databases cache entries for a fixed timed period before expiring them but in big data, we can only safely remove entries AFTER processing by the batch layer else we risk losing important data

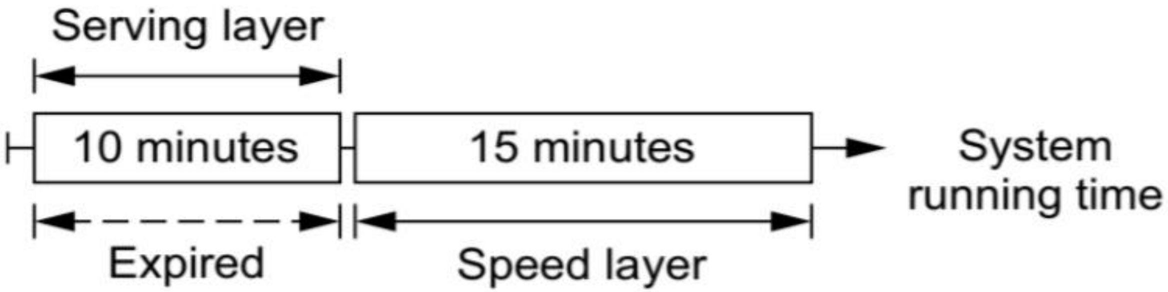
Expiration considerations

- Suppose your Lambda architecture system runs for the first time and has no data but starts collection → speed and serving layer views are empty, batch layer has no data to pre-compute
- Assuming the first batch pre-compute process takes 10 minutes, eventually:
  - Serving layer views remain empty, batch layer view is also empty
  - The speed layer has 10 minutes worth of data as its first speed layer view



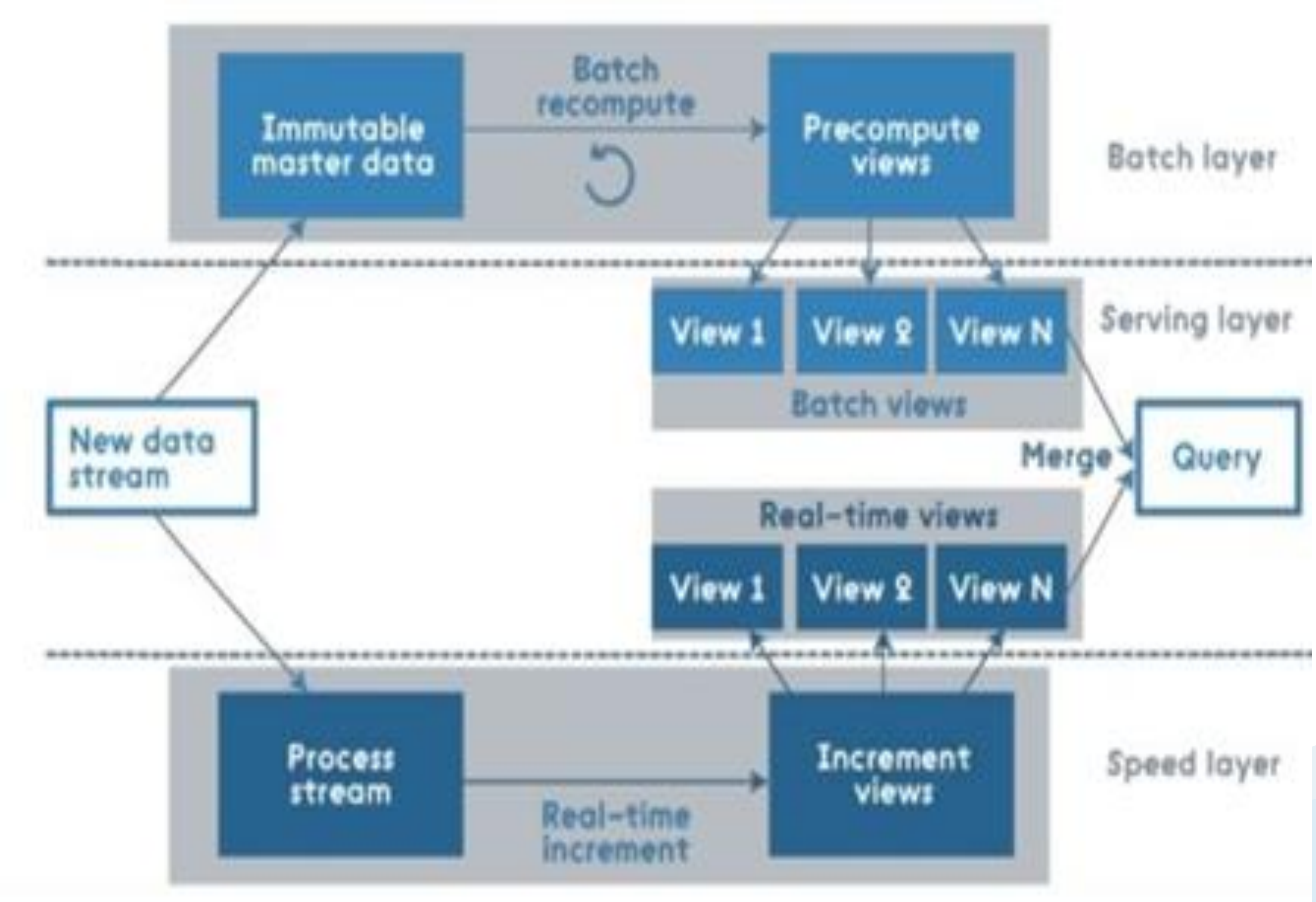
After the first batch computation run, the serving layer remains empty but the speed layer has processed recent data.

- As time progresses, the second set of data arrives and is processed
- The second run of the batch layer immediately commences to process the 10 minutes of data that accumulated during the first run (i.e. absorbing data from speed into batch layer)
- Assuming this second run takes 15 minutes, eventually
  - Serving layer has views for the first 10 minutes after receiving from batch layer.
  - Speed layer has 10+15 minute views (first and second set)
  - It should be safe to expire the first 10 minute view from the speed layer once absorbed into batch/serving layer



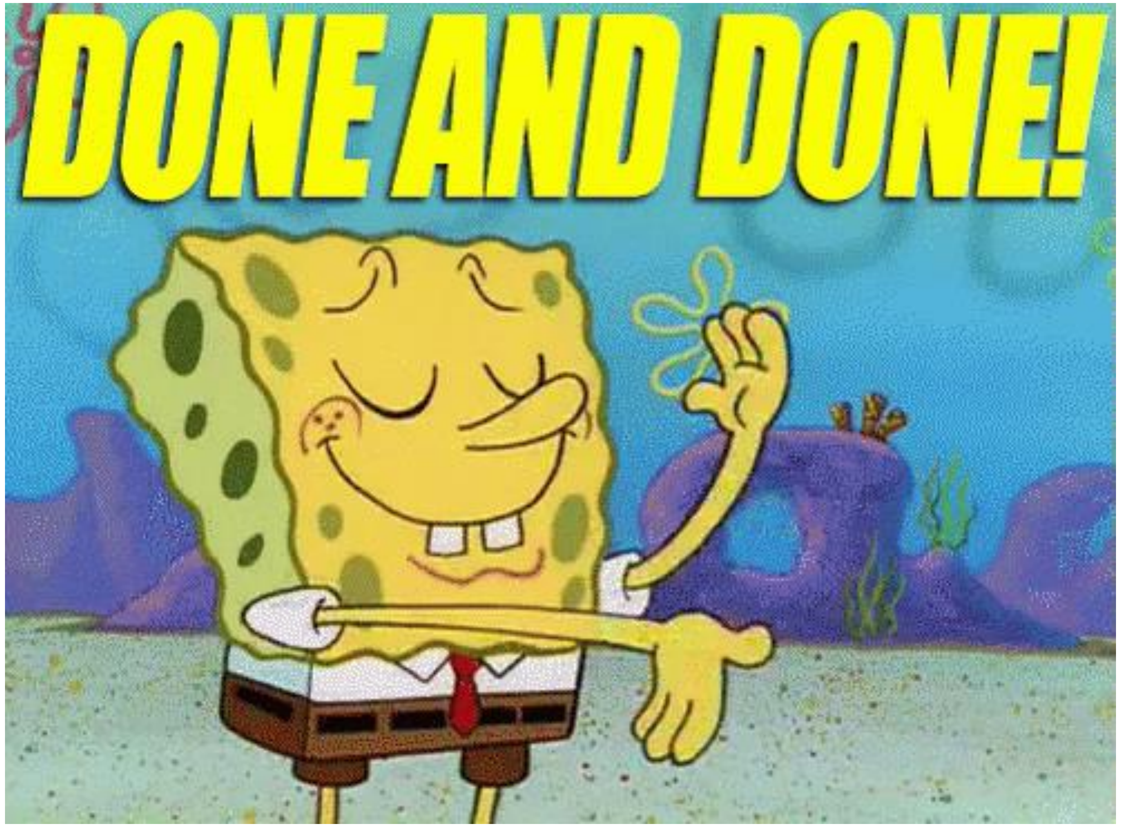
When the second batch computation run finishes, the first segment of data has been absorbed into the serving layer and can be expired from the speed layer views.

How does Lambda Architecture supports Data Science analysis ?



Done!

- That completes the theoretical concepts of Lambda Architecture



Thank you