Stream Processing

Lecture 11

2018/2019

Table of Contents

Context

- Big data systems need to store huge amounts of data
- Cloud platforms need to be elastic and fault tolerant, supporting the addition and removal of nodes
  - Storage systems must support the same features
- Traditional storage systems are not adequate for such settings

Base storage technologies

- Direct Attached Storage
  - Disks directly attached to servers
  - Hard to upgrade capacity

Existing storage technologies (2)

- Network Attached Storage
  - Storage is connected to the network directly; nodes access storage through the network
- Storage Area Network
  - Typically use optical fiber connection, allowing multipath data switching among any internal nodes

Distributed storage systems

- Distributed storage systems store data at multiple nodes. Often, nodes only have directly attached storage, because they are cheaper.
- For addressing fault-tolerance, data needs to be stored at multiple nodes.

Table of Contents

File systems

- Google File System (GFS) is a CP file system that had a large influence on other systems
  - Built for commodity servers.
  - Designed for supporting more reads than writes; sequential reads.
- HDFS derives from GFS design.

Requirements

- Large files
  - Append in the end, append may be concurrent.
  - Read sequentially.
  - Few deletes.
- Fault tolerance
  - Should tolerate machine failures;
  - Should support easy machine addition.
- Scalability
  - Should support thousands of machines.

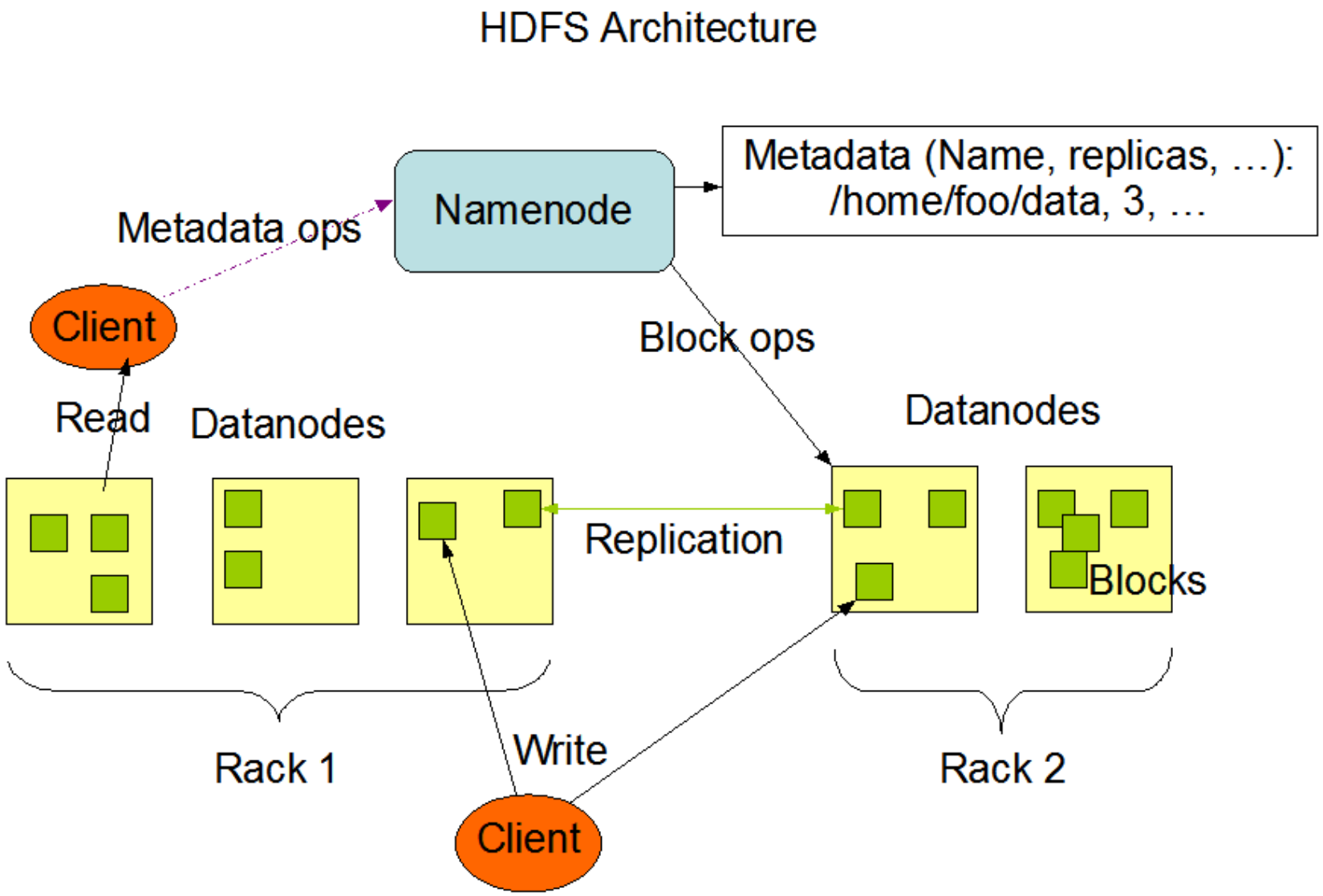
Goals of HDFS

- Very Large Distributed File System
  - 10K nodes, 100 million files, 10PB
- Assumes Commodity Hardware
  - Files are replicated to handle hardware failure
  - Detect failures and recover from them
- Optimized for Batch Processing
  - Data locations exposed so that computations can move to where data resides
  - Provides very high aggregate bandwidth

Distributed File System

- Single Namespace for entire cluster
- Data Coherency
  - Write-once-read-many access model
  - Client can only append to existing files
- Files are broken up into blocks
  - Typically 64MB block size
  - Each block replicated on multiple DataNodes

HDFS Architecture



Functions of a NameNode

- Manages File System Namespace
  - Maps a file name to a set of blocks
  - Maps a block to the DataNodes where it resides
- Cluster Configuration Management
- Replication Engine for Blocks

NameNode Metadata

- Metadata in Memory
  - The entire metadata is in main memory
  - No demand paging of metadata
- Types of metadata
  - List of files
  - List of Blocks for each file
  - List of DataNodes for each block
  - File attributes, e.g. creation time, replication factor
- A Transaction Log
  - Records file creations, file deletions etc

DataNode

- A Block Server
  - Stores data in the local file system (e.g. ext3)
  - Stores metadata of a block (e.g. CRC)
  - Serves data and metadata to Clients
- Block Report
  - Periodically sends a report of all existing blocks to the NameNode

How HDFS works

- Files are divided into blocks
  - 64 MB
- The mapping between filename and blocks goes to the NameNode
- Each block is replicated and sent off to DataNodes
  - By default, 3
  - The NameNode determines which dataNodes

Creation of a file

- On file creation creates a separate file on NameNode
- Files in HDFS are write-once and have one writer at any time.
- Avoid bothering the NameNode too often
  - Cache file data
  - When a Client has 1 chunk's worth of data, contact the NameNode to check where to send data to
  - Only sends data to the 1st replica.

Data Pieplining

- Client retrieves a list of DataNodes on which to place replicas of a block
- Client writes block to the first DataNode
- The first DataNode forwards the data to the next node in the Pipeline
- When all replicas are written, the Client moves on to write the next block in file

Block Placement

- Default Strategy
  - One replica on local node
  - Second replica on a remote rack
  - Third replica on same remote rack
- Clients read from nearest replicas

Heartbeats

- DataNodes send hearbeat to the NameNode
  - Once every 3 seconds
- NameNode uses heartbeats to detect DataNode failure

Replication Engine

- NameNode detects DataNode failures
  - Chooses new DataNodes for new replicas
  - Balances disk usage
  - Balances communication traffic to DataNodes

NameNode Failure

- A single point of failure
- Transaction Log stored in multiple directories
  - A directory on the local file system
  - A directory on a remote file system (NFS/CIFS)

- Several solution for high availability of the NameNode have been proposed.

Rebalancer

- Goal: % disk full on DataNodes should be similar
  - Usually run when new DataNodes are added
  - Cluster is online when Rebalancer is active
  - Rebalancer is throttled to avoid network congestion
  - Command line tool

Amazon S3

- Object store, with flat namespace
  - Can emulate hierarchical namespace by using names with structure.
- Used as a replacement of file systems
  - E.g. storing static objects in a web site.
- Provides high availability, by storing object at multiple replicas (in multiple devices and facilities in a given region)
  - Supports for inter-region replication.

Table of Contents

Key-value databases

- **Data model**: data is stored as key-value pairs.
- **API** (variants exist):
  - get(key) -> value
  - put(key, value)
- Some systems provide secondary indexes for faster retrieval of data.

- Simple model simplifies scalable design.

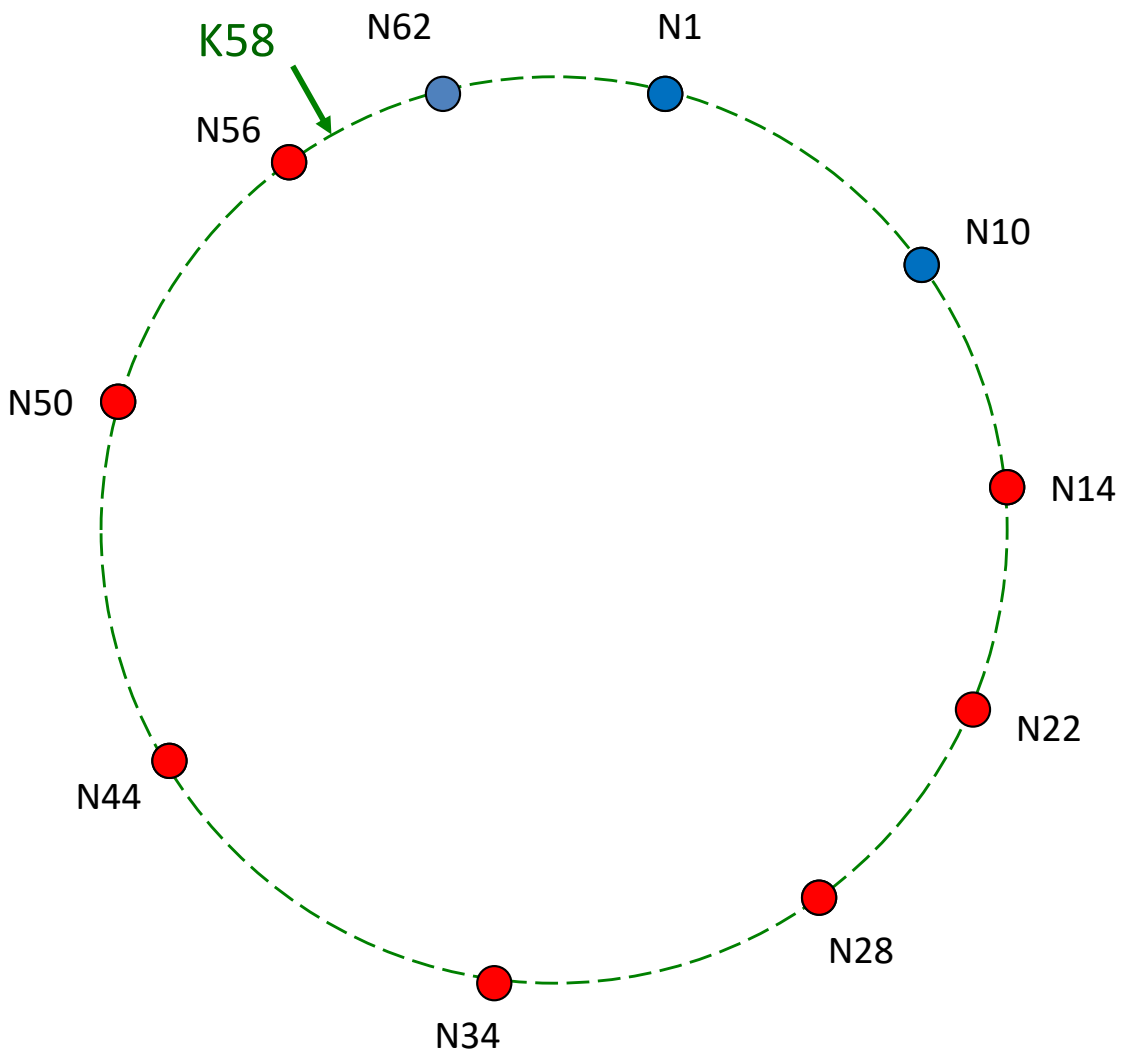
Dynamo

- Database from Amazon
  - Note: DynamoDB available at AWS provides additional features, such as strongly consistent updates.
  - Note: Cassandra is an open-source database inspired in Dynamo design.
- Data model: key-value pairs, with get/put API.

- Dynamo is an AP system: it continues working in the presence of failures
  - Gets may return old values.
  - Gets may return multiple values (in the presence of concurrent updates).
- Data partitioned using “consistent hashing”
  - Machines organized in a ring, where each node as a key.

Consistent hashing

- Each node has an id from 0 to 2^m
- Each key has an id computed by applying an hash function to the key name
- Key with id k is replicated in the n replicas that have ids greater or equal to k
- Example (n=3)



Why is this scalable?

- Good distribution of data across the nodes.
  - Hash function distributes keys uniformly.
- Easy to know which replica stores each key.
- When a node fails or a new node enters, limited amount of data needs to be moved.

Table of Contents

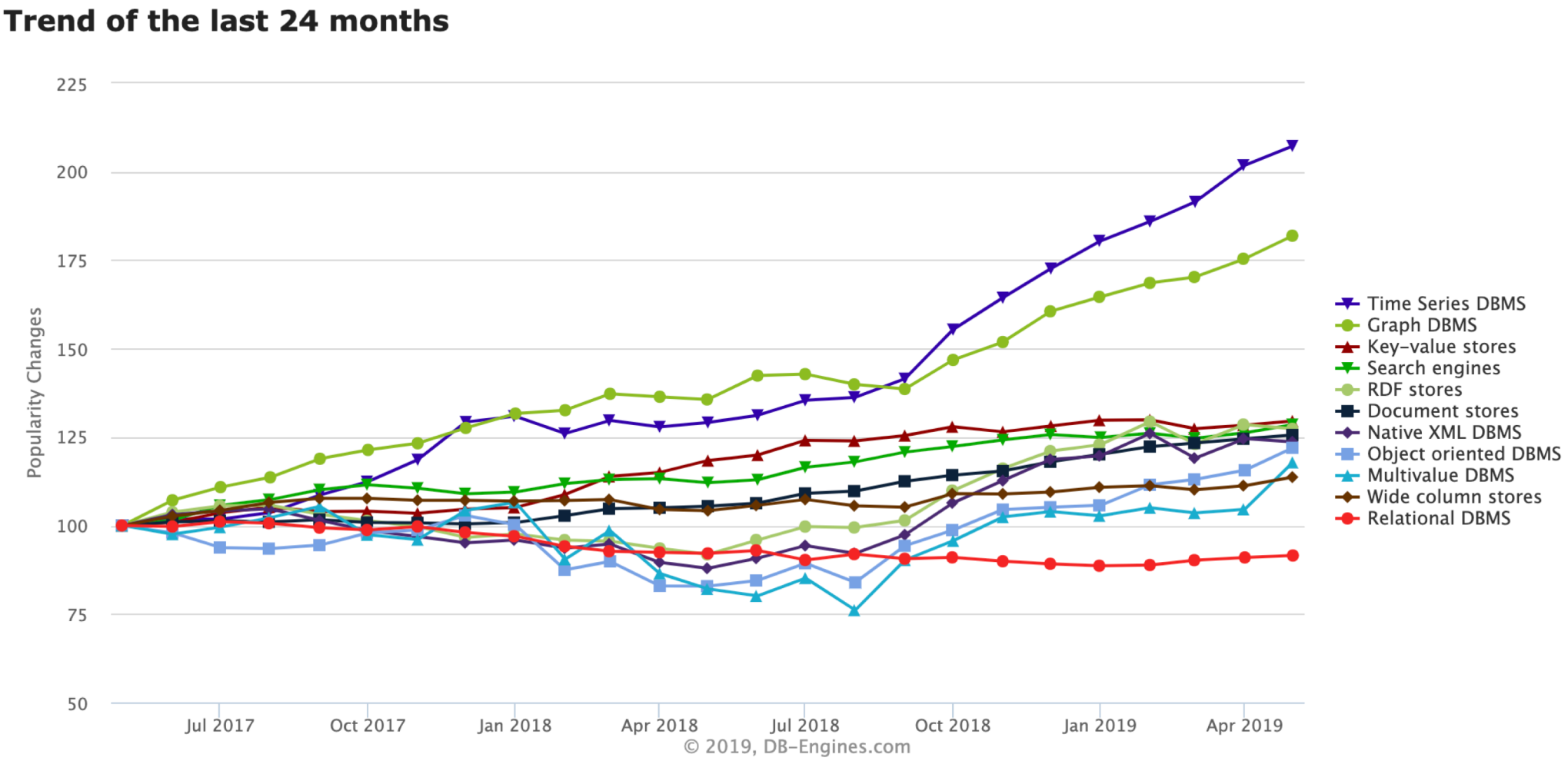
What are time-series?

- A “Time Series is an ordered sequence of values of a variable (e.g. temperature) with an associated timestamp”.
  - Time series can be obtained at equally spaced time intervals or not.
- “Sequence of discrete-time data, ordered on a timeline.”
- “Time series data are simply measurements or events that are tracked, monitored, downsampled, and aggregated over time”.

Why are time series important?

- First-generation time series focused mainly on financial markets.
- Current drivers:
  - Monitoring of computing infrastructures in a cluster: performance monitoring, network data;
  - Monitoring of physical world – IoT, sensor networks, etc.
- Emergence of Time-series Databases (TSDB)

Time-series databases popularity



Requirements: write dominate

- It should always be possible to execute writes.
- Write scale is huge - example from server monitoring

  2,000 servers, VMs, containers, or sensor units

  1,000 measurements per server/unit
  every 10 seconds

  = 17,280,000,000 distinct points per day
- Read scale is smaller
  - E.g. Facebook Gorilla reports “couple orders of magnitude lower”
  - Automated systems watching ’important’ time series
  - Dashboards for humans
  - Human operators wishing to diagnose an observed problem

Requirements: state transitions

- Identify issues that occur on monitored data.
- TSDB should support fine-grained aggregations over short-time windows.
- TSDB should have the ability to identify state transitions within tens of seconds.

Requirements: high availability and fault tolerance

- TSDB should support write and reads even in the presence of network partitions.
- TSDB should replicate data to survice server failure.

Other requirements

- ACID guarantees are not a requirement, but...
- …high percentage of writes must succeed at all times (some may fail… typically not a problem under high load). Why?
- … recent data is of higher value than older data.

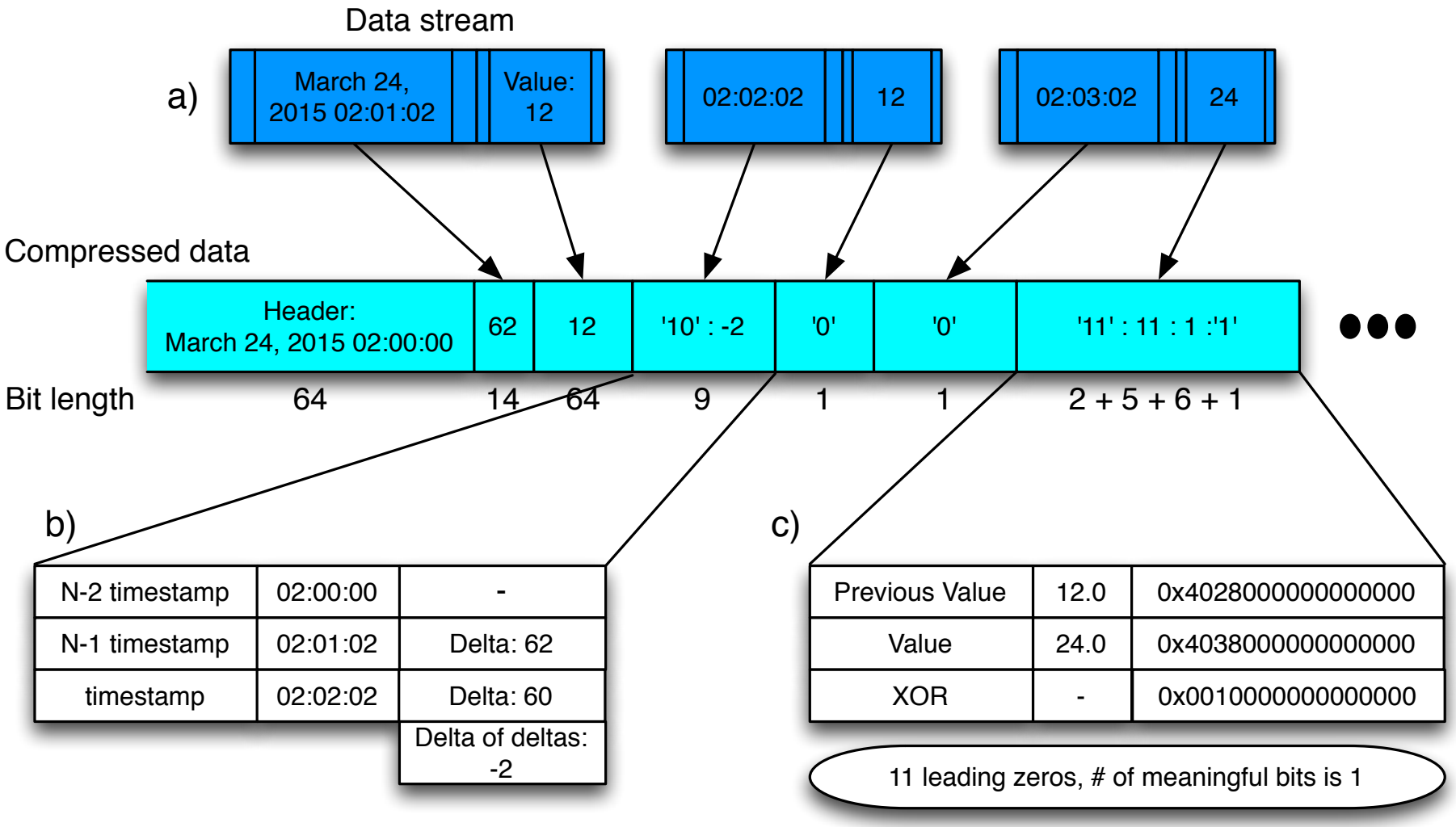
Design of a TSDB

- Problem: scale of data is enormous
- Solution: compression of the data

Time series compression

- Compresses data points within a time series.
- E.g.: Facebook Gorilla
- Each data point is a pair of 64 bit values representing the time stamp and value at that time.
- Timestamps and values are compressed separately using information about previous values – storing deltas is cheaper.

Time series compression



Design of a TSDB

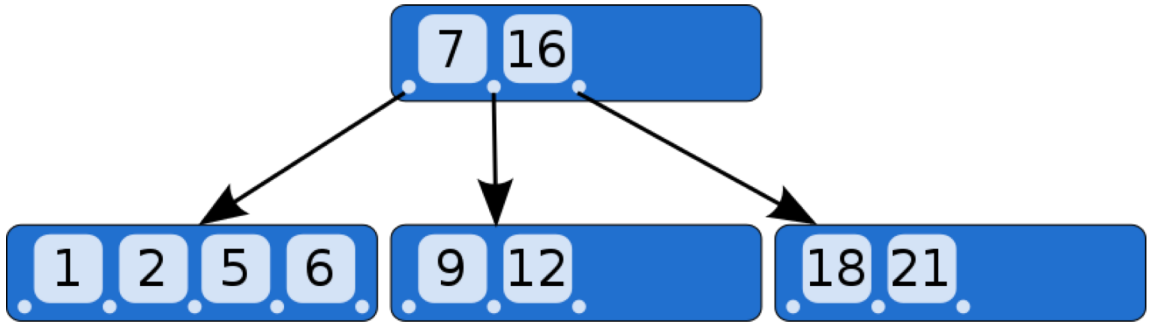
- Problem: need to write fast, read fast
- Solution: new storage designs, keep indices in memory

Indexing time series

- Need to support fast writes…
- … and fast reads

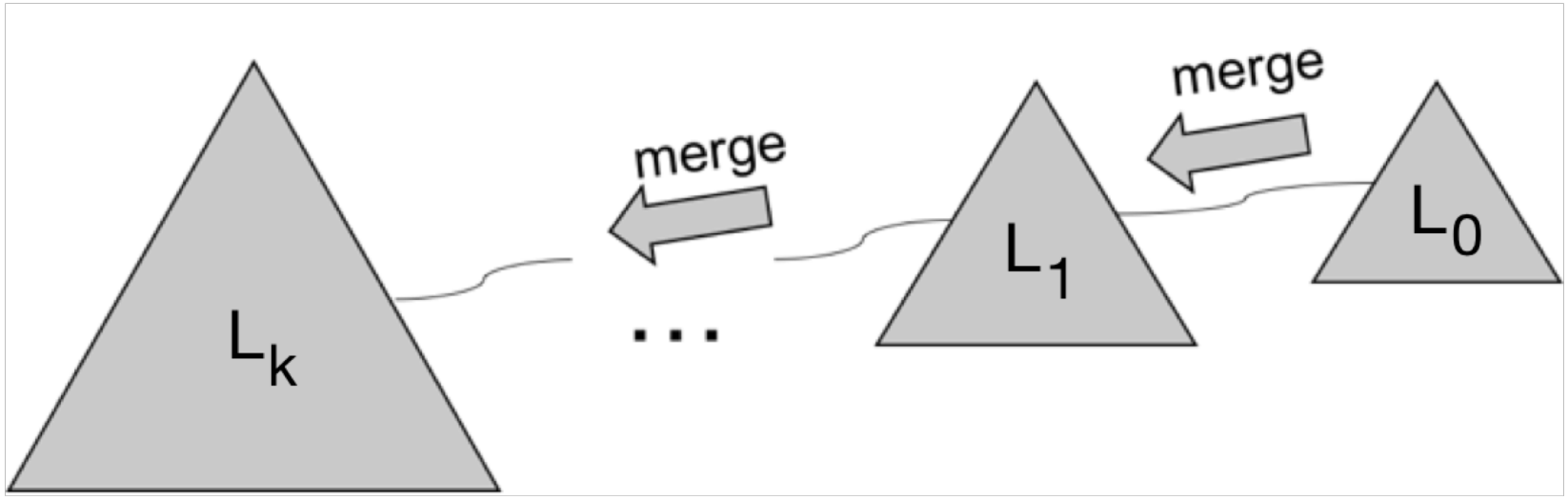
Indexing in databases: B-trees

- B-trees are the most commonly used data structure for indexing in databases
- Writes access O(log n) tree nodes
  - And are disperse over the tree
- Point read access O(log n) tree nodes
- Range reads further access the needed tree nodes



Log-structured merge tree (LSM-tree)

- An LSM-tree consists of a hierarchy of storage levels that increase in size.
- The first level, L0, is stored in memory – used to buffer updates.
- The other levels are stored on disk.



LSM-tree

- When a level gets full, its contents are merged into the following level.
- A level consists of a collection of one or more sorted files with non-overlapping key ranges.
  - The maximum size/number of the files grows with the depth of the tree

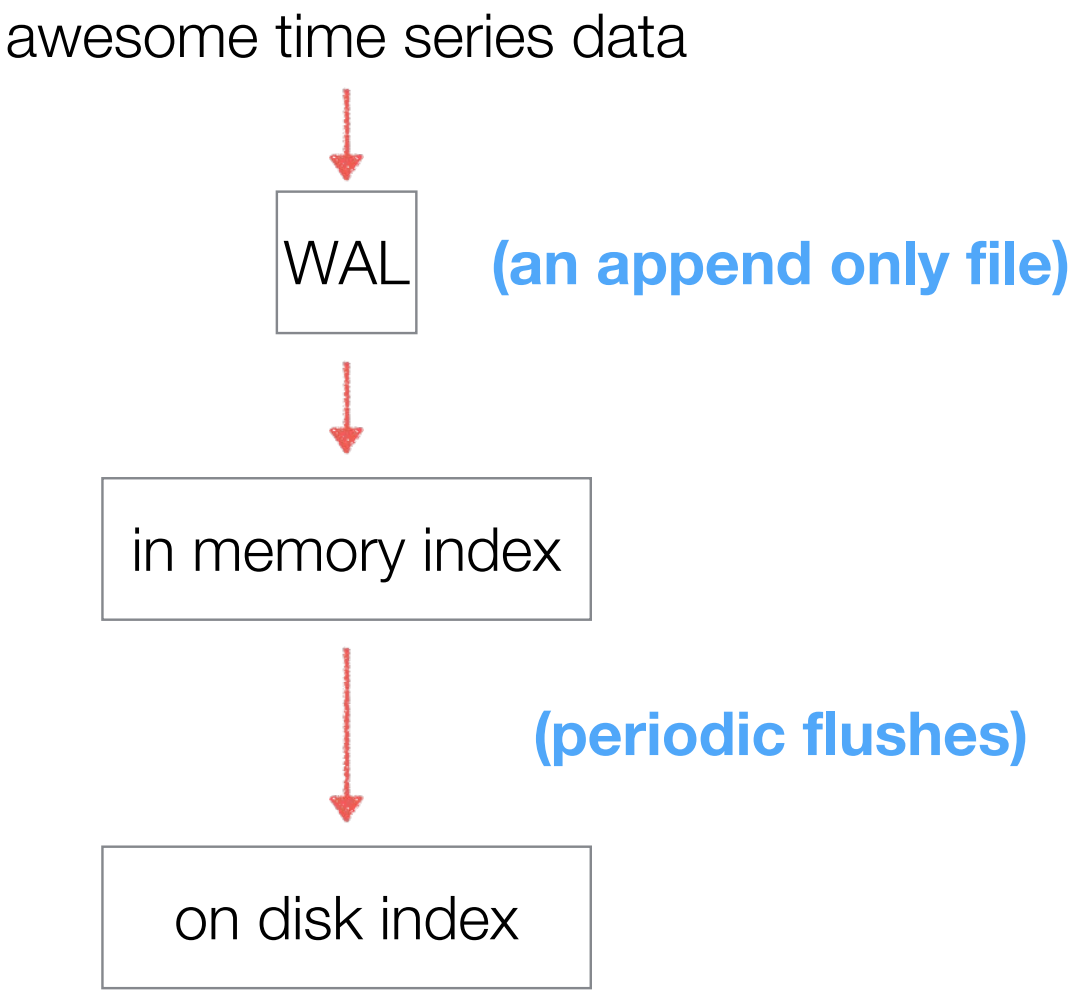
LSM-tree: operations

- Insert is fast
  - Basically adds the new (key,value) pair to L0
    - Delete consists in inserting a special marker
- Merge consists in merging a set of sorted files into a smaller number of sorted files
  - Merge of levels is done asynchronously and in batch

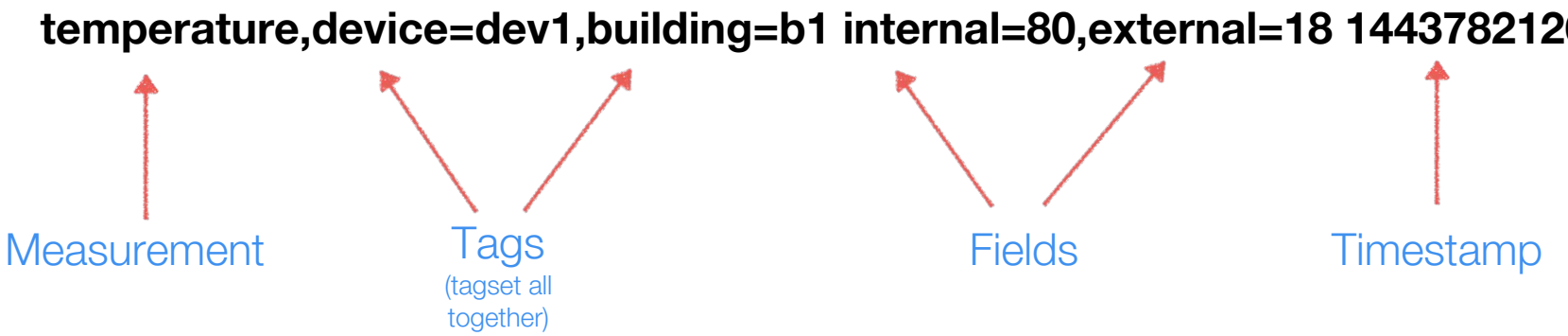
LSM-tree: operations (cont.)

- A simple lookup consists in:
  - Searching the value in L0
  - If not found, continue searching in the following levels
    - For efficiency, each level records a summary of the elements present, as a Bloom filter
- Range lookups consist in:
  - Executing a range search in every level
  - Slow, but…
    - If searching for recent values, they will be in L0 (if large enough)
    - The way merging works makes values added at similar times to be in close levels

LSM-based storage in a TSDB (e.g. Influx DB)



How to explore this for indexing in TSDB – e.g. Influx DB



- Each field is a time series with an identifier

temperature,device=dev1,building=b1#internal → **1**

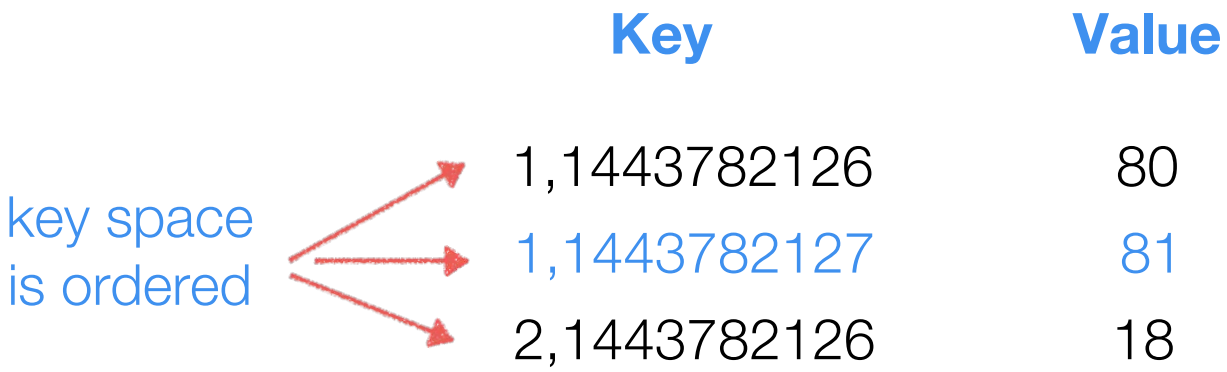
1 → **(1443782126,80)**

temperature,device=dev1,building=b1#external → **2**

2 → **(1443782126,18)**

How to explore this for indexing in TSDB – e.g. Influx DB

- Keys are the pair (identifier,timestamp)
- Data in each field is ordered by timestamp
  - Fast range queries per (id,timestamp)



From SQL schema to InfluxDB schema

park_id	planet	time	#_foodships
1	Earth	1429185600000000000	0
1	Earth	1429185601000000000	3
1	Earth	1429185602000000000	15
1	Earth	1429185603000000000	15
2	Saturn	1429185600000000000	5
2	Saturn	1429185601000000000	9
2	Saturn	1429185602000000000	10
2	Saturn	1429185603000000000	14
3	Jupiter	1429185600000000000	20
3	Jupiter	1429185601000000000	21
3	Jupiter	1429185602000000000	21
3	Jupiter	1429185603000000000	20
4	Saturn	1429185600000000000	5
4	Saturn	1429185601000000000	5
4	Saturn	1429185602000000000	6
4	Saturn	1429185603000000000	5

```
name: foodships
tags: park_id=1, planet=Earth
time                  #_foodships
----                  ------------
2015-04-16T12:00:00Z  0
2015-04-16T12:00:01Z  3
2015-04-16T12:00:02Z  15
2015-04-16T12:00:03Z  15

name: foodships
tags: park_id=2, planet=Saturn
time                  #_foodships
----                  ------------
2015-04-16T12:00:00Z  5
2015-04-16T12:00:01Z  9
2015-04-16T12:00:02Z  10
2015-04-16T12:00:03Z  14

name: foodships
tags: park_id=3, planet=Jupiter
time                  #_foodships
----                  ------------
2015-04-16T12:00:00Z  20
2015-04-16T12:00:01Z  21
2015-04-16T12:00:02Z  21
2015-04-16T12:00:03Z  20

name: foodships
tags: park_id=4, planet=Saturn
time                  #_foodships
----                  ------------
2015-04-16T12:00:00Z  5
2015-04-16T12:00:01Z  5
2015-04-16T12:00:02Z  6
2015-04-16T12:00:03Z  5
```

API (e.g. InfluxDB): continuous queries

Periodically execute a query and add the result to a table

```
CREATE CONTINUOUS QUERY <cq_name> ON <database_name>
BEGIN
    SELECT <function[s]> INTO <destination_measurement>
        FROM <measurement>
        [WHERE <stuff>]
        GROUP BY time(<interval>)[,<tag_key[s]>]
END
```

API (e.g. Influx DB): continuous queries

Periodically execute a query and add the result to a table

```
CREATE CONTINUOUS QUERY "cq_basic" ON "transportation"
BEGIN
    SELECT mean("passengers") INTO "average_passengers”
    FROM "bus_data" GROUP BY time(1h)
END
```

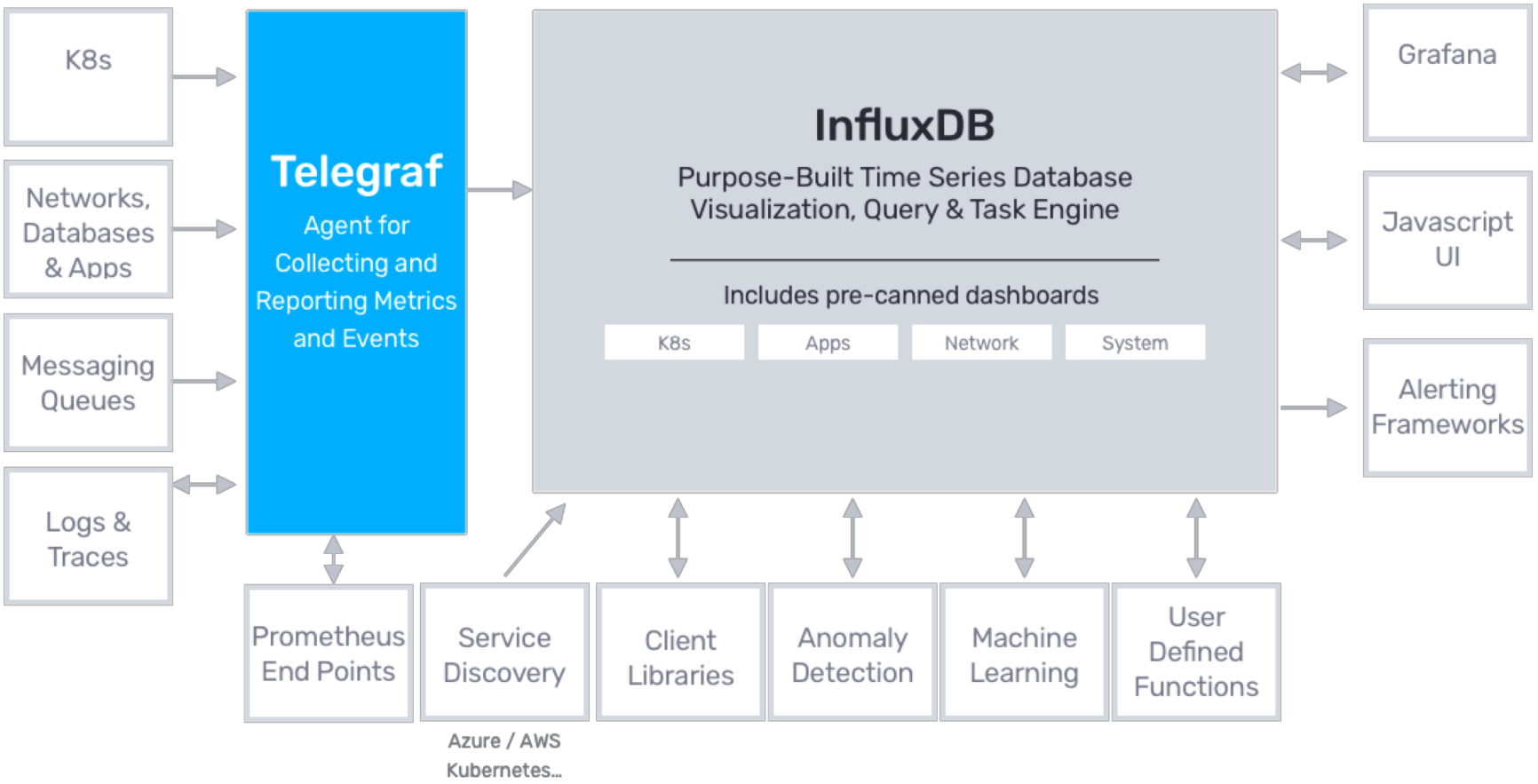
API (e.g. Influx DB): retention policy

Retention policy defines for how long data is kept

CREATE RETENTION POLICY <retention_policy_name>
ON <database_name> DURATION <duration>
REPLICATION <n> [SHARD DURATION <duration>] [DEFAULT]

CREATE RETENTION POLICY "one_day_only" ON ”transportation"
DURATION 1d REPLICATION 1

Influx DB ecosystem



Bibliography

- HDFS Architecture. Dhruba Borthakur.
  - http://svn.apache.org/repos/asf/hadoop/common/tags/release-0.19.2/docs/hdfs_design.pdf
- Gorilla: A Fast, Scalable, In-Memory Time Series
  - https://www.vldb.org/pvldb/vol8/p1816-teller.pdf
- Too detailed references:
  - Log-structured merge trees
    - https://www.cs.umb.edu/~poneil/lsmtree.pdf
  - https://docs.influxdata.com/influxdb/v1.7/concepts/storage_engine/

Acknowledgments

- Some images from:
  - Inside the InfluxDB Storage Engine. Gianluca Arbezzano
  - Tuomas Pelkonen, et. al. Gorilla: A Fast, Scalable, In-Memory Time Series Database. VLDB’15.