Design of Key-Value Stores for IoT Edge Storage

Dr. Rajiv Misra
Associate Professor
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Content of this Lecture:

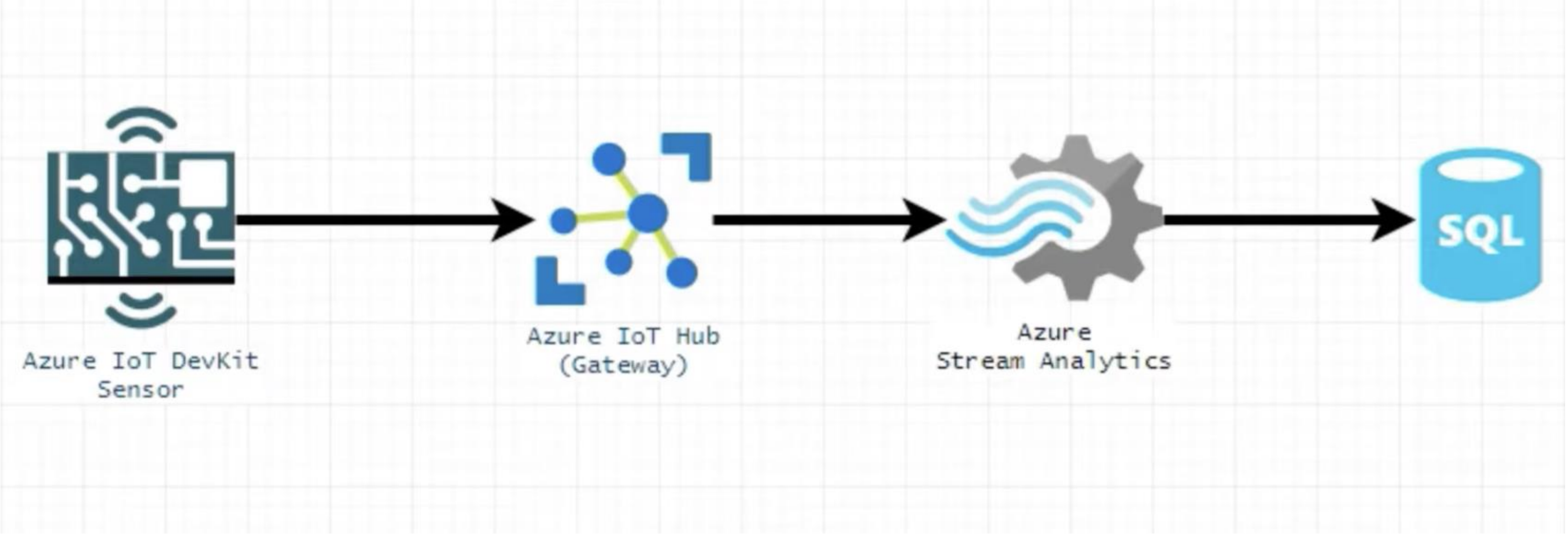
- Design of datastore for IoT applications.

- In this lecture, we will discuss the design and insight of **Key-value/NoSQL stores** for today's Edge storage systems.

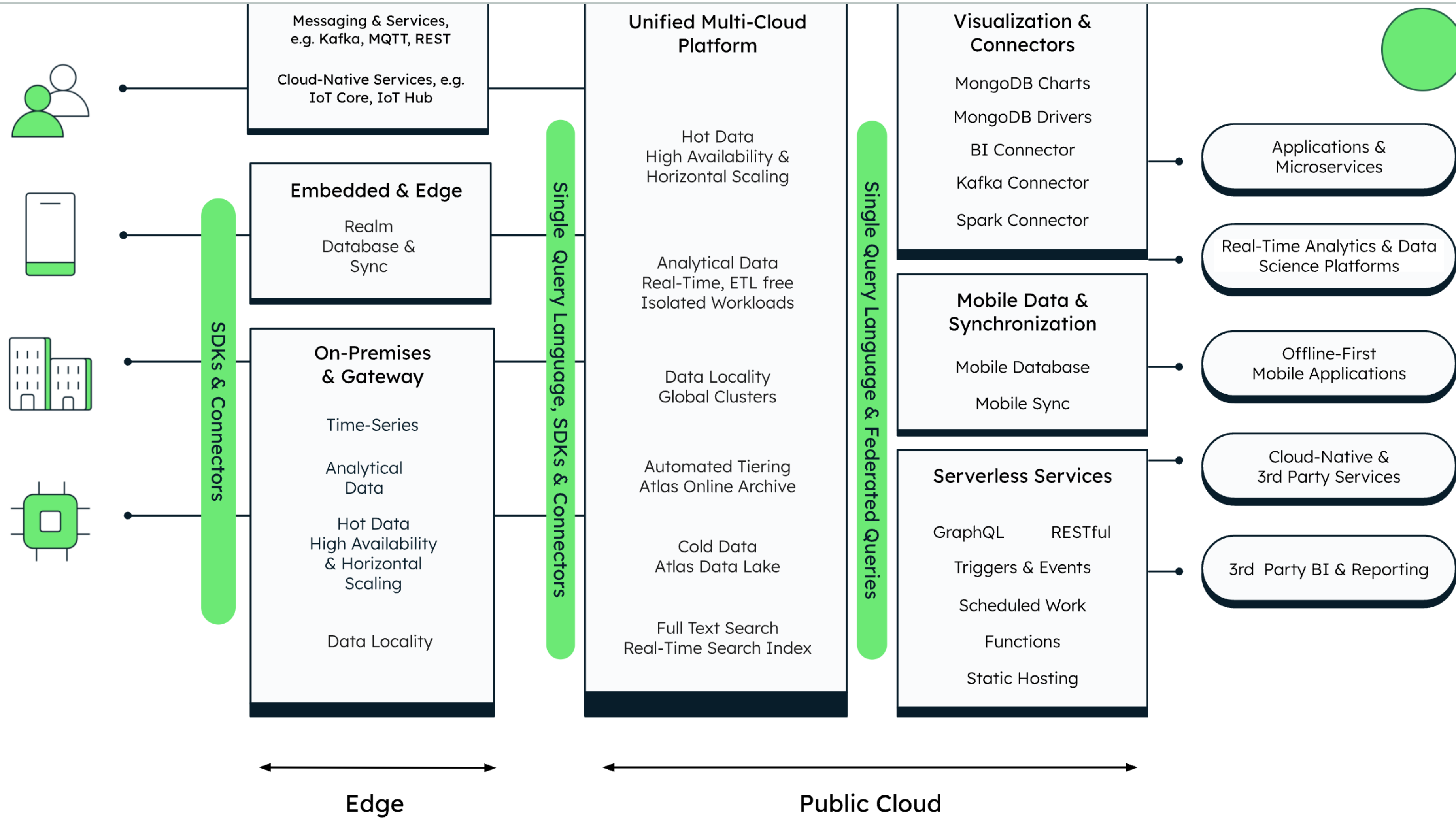
- We will also discuss one of the most popular cloud storage system i.e. **Apache Cassandra** and different consistency solutions.

IoT Edge: Data Flow

Initially the data collected from the azure IOT sensor is passed through the IOT-Edge gateway and then we need to pull the data from IOT-Edge using stream analytics and then stream that data then to from the IOT Edge to the data-store/database



IoT Edge: Data Flow



IoT Edge: Databases

The most popular databases for IoT apps are InfluxDB, CrateDB, Riak TS, MongoDB, RethinkDB, SQLite, Apache Cassandra.

To select the right storage for Time Series and IoT domain use case, it depends upon the data-access methods, you may require the following database:

- **Hot database:**

These are typically used for data that is frequently being queried or updated. They are often a good choice for storing data as they provide read and write capabilities with little latency at the lowest cost. When choosing a hot database you can consider the following features — flexibility in data formats, querying abilities, messaging/ queueing capability, and tiered memory models.

- **Cold Database:**

They store information in their original state with little to no changes made thereafter. In contrast with real-time data collection, storing huge volumes of historical data can be a difficult task on cold databases.

IoT Edge: Databases

NoSQL with Built-In Sorting

BigTable, HBase, Cassandra, DynamoDB, Accumulo are often used to store time-series data.

> ***Strong Sides:*** *Extremely well scaled for writes. Performing the basic level of analytics extremely efficient.*
>
> ***Weak Sides:*** *All other kinds of analytics are not supported and not efficient*

NoSQL Purpose-Built Time Series DB

There are engines that have been designed from the ground up as Time Series databases. In the majority of cases, they are NoSQL.

NewSQL In-Memory Databases

The in-memory nature of SQL databases increases their ability to handle fast data ingestion. SQL interface enriched by the time buckets normalization support

> ***Strong Sides:*** *Provide the reach analytics capabilities.*
>
> ***Weak Sides:*** *The scalability for writes and reads are usually limited or is very expensive*

Cloud Time-Series Platforms

Azure and AWS released recently their time series data services/platforms:

> **Azure Time Series Insights**
>
> **Amazon Timestream**

The platforms cover many aspects of the time series data storing, visualizing, and really reach capabilities in querying. They have built-in separation of data between hot, warm, and cold storage to make the data storing and retrieval well balances from the cost of ownership perspective.

IoT Edge Database: Example

As a continuation of the series of lecture about IoT Data Analytics, let's use the Fitness Tracker use case which represents well a typical IoT use case. A dataset (as it is also described here and here) consists of a set of observation, and each observation contains:

- A *metric name* generating by a sensor/edge, i.e.: heart rate, elevation, steps
- A *metric value* generated by the sensor bound to the point in time, i.e.: (2020–11–12 17:14:07, 71bpm), (2020–11–12 17:14:32, 93bpm), etc
- *Tags or Context description* in which a given sensor is generating data, i.e.: device model, geography location, user, activity type, etc.

IoT Edge Database: Functional Requirement

Basic Level: Simple Data Retrieval

- **Random data access:** for the particular point in time return the proper metric value
- **Small range scans:** for the particular time range (reasonably small, within minutes or hours depending on the frequency of data generation) return the sequential metric values (i.e.: to draw a standard chart on it)

IoT Edge Database: Functional Requirement

Middle Level: Time Window Normalization

The measurement events usually supposed to be triggered on a predefined recurrence basis, but there are always deviations in data points timing. That is why it is highly desirable to have capabilities around building predefined time windows to normalize the time series data.

To the mid-level capacities it is worth to add more sophisticated diagnostic analytics/ad-hoc queries:

- **Flexible Filtering:** filter data points based on predicate on tags/context attributes, *i.e.: filtering data points by some region, user, or activity type*
- **Flexible Aggregations:** grouping and aggregations on tags/context attributes or their combinations, *i.e.: max hearth rate by region by activity type.*

IoT Edge Database: Functional Requirement

Advance Level: Sequential Row Pattern Matching

The most advanced level would include checking if the **sequence of events matches** the particular **pattern** to perform introspection and advanced diagnosis:

- Did similar patterns of measurements precede specific events?;
- What measurements might indicate the cause of some event, such as a failure?

IoT Edge Database: Non Functional Requirement

Besides the functional requirements, it's really crucial to consider non-functional requirements which often are the main drivers for the selection:

- **Scalable storage:** ability to handle big data volumes
- **Scalable writes:** the ability to handle a big amount of simultaneous writes. This is closely related to the real-time data access — the ability to have the minimum possible lag between when the data point is generated and when it's available for reading.
- **Scalable reads:** the ability to handle a big amount of simultaneous reads
- **High Maturity:** presence on the market and community support.

The Key-value Abstraction

- (Business) Key → Value
- (flipkart.com) item number → information about it
- (easemytrip.com) Flight number → information about flight, e.g., availability
- (twitter.com) tweet id → information about tweet
- (mybank.com) Account number → information about it

The Key-value Abstraction (2)

- **It's a dictionary datastructure.**
  - Insert, lookup, and delete by key
  - Example: hash table, binary tree
- But distributed.
- Seems familiar? Remember **Distributed Hash tables (DHT) in P2P systems?**
- Key-value stores reuse many techniques from DHTs.

Is it a kind of database ?

- Yes, kind of
- **Relational Database Management Systems (RDBMSs)** have been around for ages
- **MySQL** is the most popular among them
- Data stored in tables
- Schema-based, i.e., structured tables
- Each row (data item) in a table has a primary key that is unique within that table
- Queried using **SQL (Structured Query Language)**
- Supports joins

Relational Database Example

users table

user_id	name	zipcode	blog_url	blog_id
110	Smith	98765	smith.com	11
331	Antony	54321	antony.in	12
767	John	75676	john.net	13

Primary keys

Foreign keys

blog table

Id	url	last_updated	num_posts
11	smith.com	9/7/17	991
13	john.net	4/2/18	57
12	antony.in	15/6/16	1090

Example SQL queries

1. SELECT zipcode
   FROM users
   WHERE name = "John"

2. SELECT url
   FROM blog
   WHERE id = 11

3. SELECT users.zipcode, blog.num_posts
   FROM users JOIN blog
   ON users.blog_url = blog.url

Mismatch with today's workloads

- **Data: Large and unstructured:** Difficult to come out with schemas where the data can fit
- **Lots of random reads and writes:** Coming from millions of clients.
- **Sometimes write-heavy:** Lot more writes compare to read
- **Foreign keys rarely needed**
- **Joins infrequent**

Needs of Today's Workloads

- Speed
- Avoid Single point of Failure (SPoF)
- Low TCO (Total cost of operation and Total cost of ownership)
- Fewer system administrators
- Incremental Scalability
- Scale out, not scale up

Scale out, not Scale up

- **Scale up = grow your cluster capacity by replacing with more powerful machines**
    - Traditional approach
    - Not cost-effective, as you're buying above the sweet spot on the price curve
    - And you need to replace machines often

- **Scale out = incrementally grow your cluster capacity by adding more COTS machines (Components Off the Shelf)**
    - Cheaper
    - Over a long duration, phase in a few newer (faster) machines as you phase out a few older machines
    - Used by most companies who run datacenters and clouds today

Key-value/NoSQL Data Model

- NoSQL = "Not Only SQL"
- Necessary API operations: **get(key) and put(key, value)**
  - And some extended operations, e.g., "CQL" in Cassandra key-value store

- **Tables**
  - "Column families" in Cassandra, "Table" in HBase, "Collection" in MongoDB
  - Like RDBMS tables, but …
  - May be unstructured: May not have schemas
    - Some columns may be missing from some rows
  - Don't always support joins or have foreign keys
  - Can have index tables, just like RDBMSs

Key-value/NoSQL Data Model

- Unstructured
- No schema imposed
- Columns missing from some Rows
- No foreign keys, joins may not be supported

Key

Value

users table

user_id	name	zipcode	blog_url
110	Smith	98765	smith.com
331	Antony		antony.in
767		75676	john.net

Value

blog table

Id	url	last_updated	num_posts
11	smith.com	9/7/17	991
13	john.net		57
12	antony.in	15/6/16	

Column-Oriented Storage

NoSQL systems often use column-oriented storage

- RDBMSs store an entire row together (on disk or at a server)
- NoSQL systems typically store a column together (or a group of columns).
    - Entries within a column are indexed and easy to locate, given a key (and vice-versa)
- **Why useful?**
    - Range searches within a column are fast since you don't need to fetch the entire database
    - E.g., Get me all the blog_ids from the blog table that were updated within the past month
        - Search in the the last_updated column, fetch corresponding blog_id column
        - Don't need to fetch the other columns

Design of Apache Cassandra

Cassandra

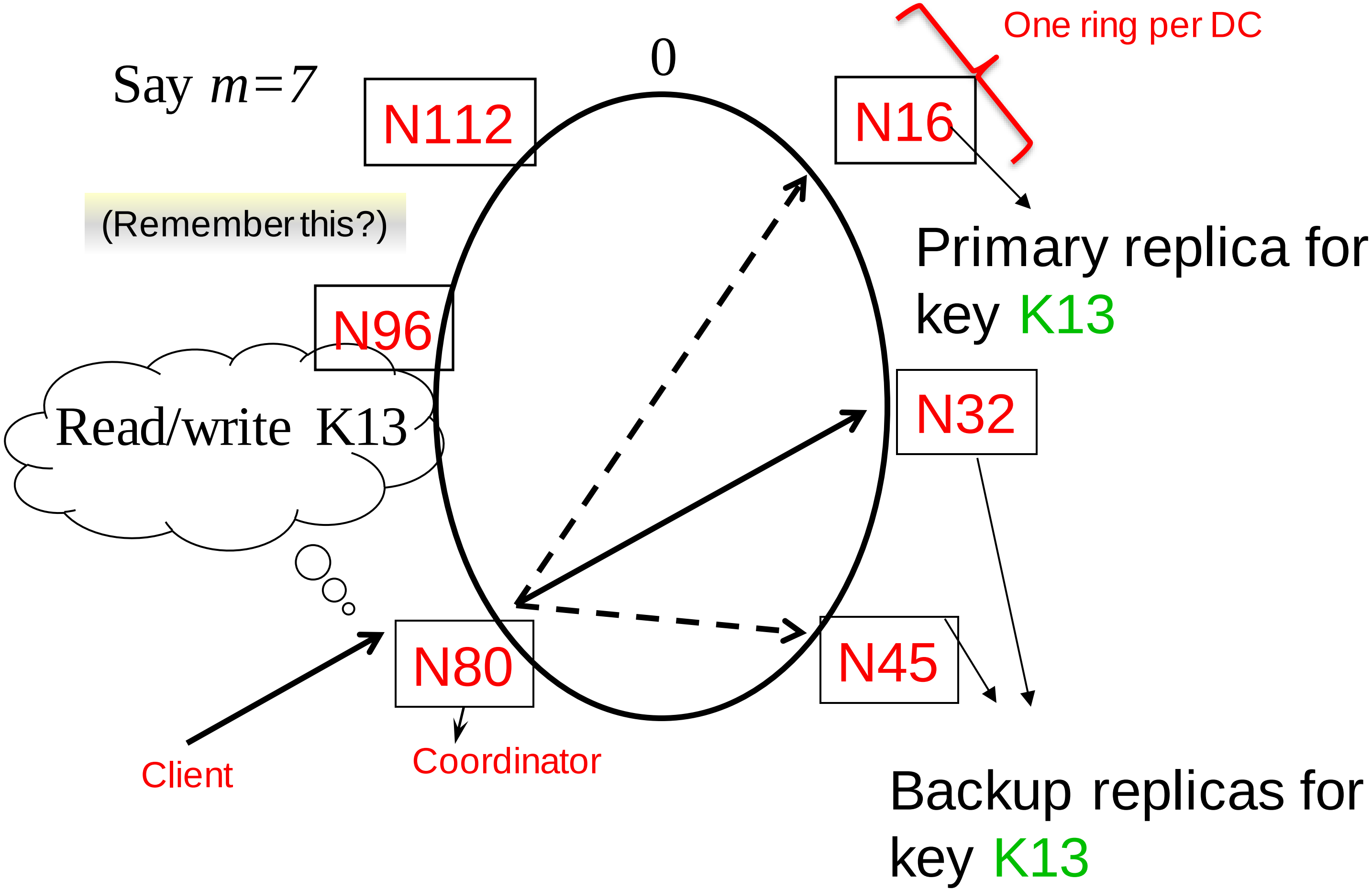
- **A distributed key-value store**
- Intended to run in a datacenter (and also across DCs)
- Originally designed at Facebook
- Open-sourced later, today an Apache project
- Some of the companies that use Cassandra in their production clusters
  - Blue chip companies: IBM, Adobe, HP, eBay, Ericsson
  - Newer companies: Twitter
  - Nonprofit companies: PBS Kids
  - Netflix: uses Cassandra to keep track of positions in the video.

Inside Cassandra: Key -> Server Mapping

- How do you decide which server(s) a key-value resides on?

Say $m=7$

(Remember this?)



Cassandra uses a Ring-based DHT but without finger tables or routing

Key →server mapping is the "Partitioner"

Data Placement Strategies

- **Replication Strategy:**
  1. *SimpleStrategy*
  2. *NetworkTopologyStrategy*

1. **SimpleStrategy:** uses the Partitioner, of which there are two kinds
   1. ***RandomPartitioner***: Chord-like hash partitioning
   2. ***ByteOrderedPartitioner***: Assigns ranges of keys to servers.
      - Easier for ***range queries*** (e.g., Get me all twitter users starting with [a-b])
2. **NetworkTopologyStrategy:** for multi-DC deployments
   - Two replicas per DC
   - Three replicas per DC
   - Per DC
     - First replica placed according to Partitioner
     - Then go clockwise around ring until you hit a different rack

Snitches

- **Maps:** IPs to racks and DCs. Configured in cassandra.yaml config file
- **Some options:**
  - **SimpleSnitch:** Unaware of Topology (Rack-unaware)
  - **RackInferring:** Assumes topology of network by octet of server's IP address
    - 101.102.103.104 = x.<DC octet>.<rack octet>.<node octet>
  - **PropertyFileSnitch:** uses a config file
  - **EC2Snitch:** uses EC2.
    - EC2 Region = DC
    - Availability zone = rack
- Other snitch options available

Writes

- Need to be lock-free and fast (no reads or disk seeks)
- Client sends write to one coordinator node in Cassandra cluster
  - Coordinator may be per-key, or per-client, or per-query
  - Per-key Coordinator ensures writes for the key are serialized
- Coordinator uses Partitioner to send query to all replica nodes responsible for key
- When X replicas respond, coordinator returns an acknowledgement to the client
  - X?

Writes (2)

- **Always writable: Hinted Handoff mechanism**
  - If any replica is down, the coordinator writes to all other replicas, and keeps the write locally until down replica comes back up.
  - When all replicas are down, the Coordinator (front end) buffers writes (for up to a few hours).

- **One ring per datacenter**
  - Per-DC coordinator elected to coordinate with other DCs
  - Election done via Zookeeper, which runs a Paxos (consensus) variant

Writes at a replica node

On receiving a write

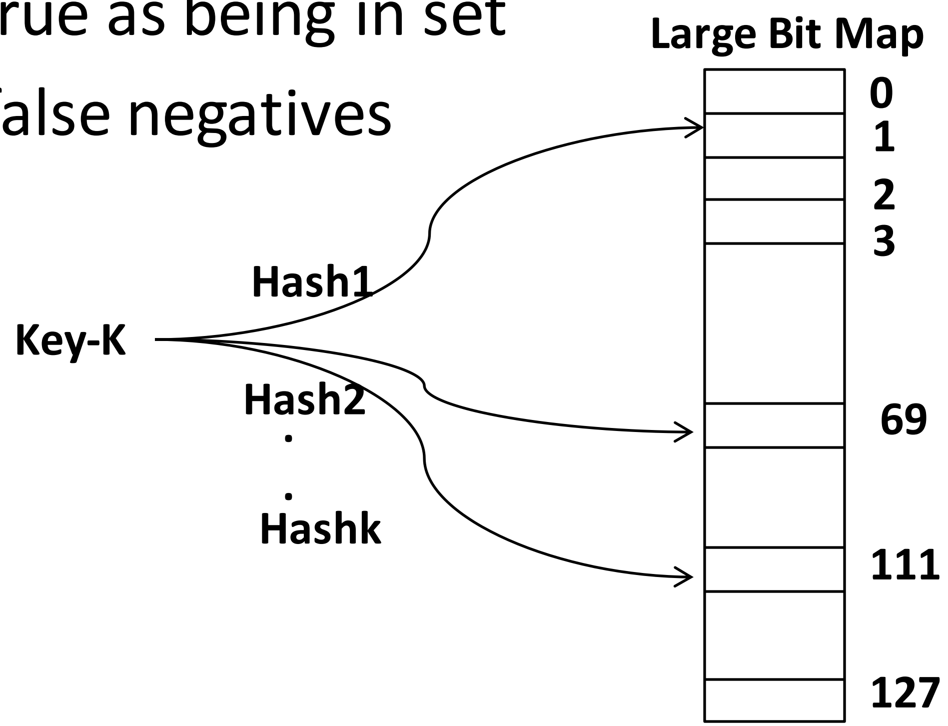
1. Log it in disk commit log (for failure recovery)
2. Make changes to appropriate memtables
   - **Memtable** = In-memory representation of multiple key-value pairs
   - *Typically append-only datastructure (fast)*
   - Cache that can be searched by key
   - Write-back aas opposed to write-through

Later, when memtable is full or old, flush to disk

- Data File: An **SSTable** (Sorted String Table) – list of key-value pairs, sorted by key
- *SSTables are immutable (once created, they don't change)*
- Index file: An SSTable of (key, position in data sstable) pairs
- And a Bloom filter (for efficient search)

Bloom Filter

- Compact way of representing a set of items
- Checking for existence in set is cheap
- Some probability of false positives: an item not in set may check true as being in set
- Never false negatives



On insert, set all hashed bits.

On check-if-present, return true if all hashed bits set.

- **False positives**

False positive rate low

- **m=4 hash functions**
- **100 items**
- **3200 bits**
- **FP rate = 0.02%**

Compaction

Data updates accumulate over time and SStables and logs need to be compacted

- The process of compaction merges SSTables, i.e., by merging updates for a key
- Run periodically and locally at each server

Deletes

Delete: don't delete item right away

- Add a **tombstone** to the log
- Eventually, when compaction encounters tombstone it will delete item

Reads

Read: Similar to writes, except

- **Coordinator can contact X replicas (e.g., in same rack)**
  - Coordinator sends read to replicas that have responded quickest in past
  - When X replicas respond, coordinator returns the latest-timestamped value from among those X
  - (X? We will check it later.)
- **Coordinator also fetches value from other replicas**
  - Checks consistency in the background, initiating a **read repair** if any two values are different
  - This mechanism seeks to eventually bring all replicas up to date
- **At a replica**
  - A row may be split across multiple SSTables => reads need to touch multiple SSTables => reads slower than writes (but still fast)

Membership

- Any server in cluster could be the coordinator
- So every server needs to maintain a list of all the other servers that are currently in the server
- List needs to be updated automatically as servers join, leave, and fail

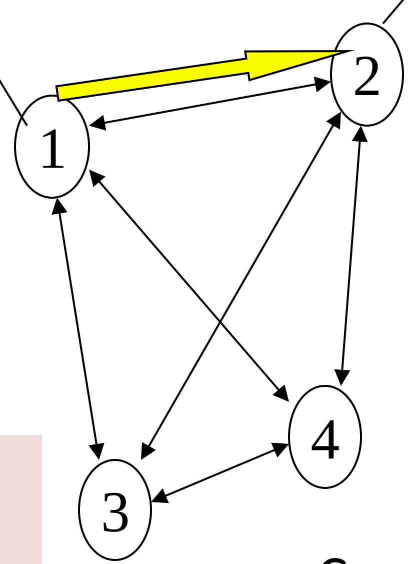
Cluster Membership – Gossip-Style

Cassandra uses gossip-based cluster membership

1	10120	66
2	10103	62
3	10098	63
4	10111	65

Address **Time (local)**

Heartbeat Counter



1	10118	64
2	10110	64
3	10090	58
4	10111	65

1	10120	70
2	10110	64
3	10098	70
4	10111	65

Protocol:

- Nodes periodically gossip their membership list
- On receipt, the local membership list is updated, as shown
- If any heartbeat older than Tfail, node is marked as failed

Current time : 70 at node 2

(asynchronous clocks)

(Remember this?)

Suspicion Mechanisms in Cassandra

- Suspicion mechanisms to adaptively set the timeout based on underlying network and failure behavior
- **Accrual detector:** Failure Detector outputs a value (PHI) representing suspicion
- Applications set an appropriate threshold
- **PHI calculation for a member**
  - Inter-arrival times for gossip messages
  - PHI(t) =

    – log(CDF or Probability(t_now – t_last))/log 10
  - PHI basically determines the detection timeout, but takes into account historical inter-arrival time variations for gossiped heartbeats
- In practice, PHI = 5 => 10-15 sec detection time

Cassandra Vs. RDBMS

- MySQL is one of the most popular (and has been for a while)
- On > 50 GB data
- **MySQL**
  - Writes 300 ms avg
  - Reads 350 ms avg
- **Cassandra**
  - Writes 0.12 ms avg
  - Reads 15 ms avg
- Orders of magnitude faster
- What's the catch? What did we lose?

CAP Theorem

CAP Theorem

- **Proposed by Eric Brewer (Berkeley)**
- Subsequently proved by Gilbert and Lynch (NUS and MIT)
- In a distributed system you can satisfy atmost 2 out of the 3 guarantees:

1. **Consistency:** all nodes see same data at any time, or reads return latest written value by any client
2. **Availability:** the system allows operations all the time, and operations return quickly
3. **Partition-tolerance:** the system continues to work in spite of network partitions

Why is Availability Important?

- **Availability =** Reads/writes complete reliably and quickly.
- Measurements have shown that a 500 ms increase in latency for operations at Amazon.com or at Google.com can cause a 20% drop in revenue.
- At Amazon, each added millisecond of latency implies a $6M yearly loss.
- **User cognitive drift:** If more than a second elapses between clicking and material appearing, the user's mind is already somewhere else
- SLAs (Service Level Agreements) written by providers predominantly deal with latencies faced by clients.

Why is Consistency Important?

- **Consistency =** all nodes see same data at any time, or reads return latest written value by any client.

- When you access your bank or investment account via multiple clients (laptop, workstation, phone, tablet), you want the updates done from one client to be visible to other clients.

- When thousands of customers are looking to book a flight, all updates from any client (e.g., book a flight) should be accessible by other clients.

Why is Partition-Tolerance Important?

- **Partitions** can happen across datacenters when the Internet gets disconnected
  - Internet router outages
  - Under-sea cables cut
  - DNS not working

- Partitions can also occur within a datacenter, e.g., a rack switch outage

- Still desire system to continue functioning normally under this scenario

CAP Theorem Fallout

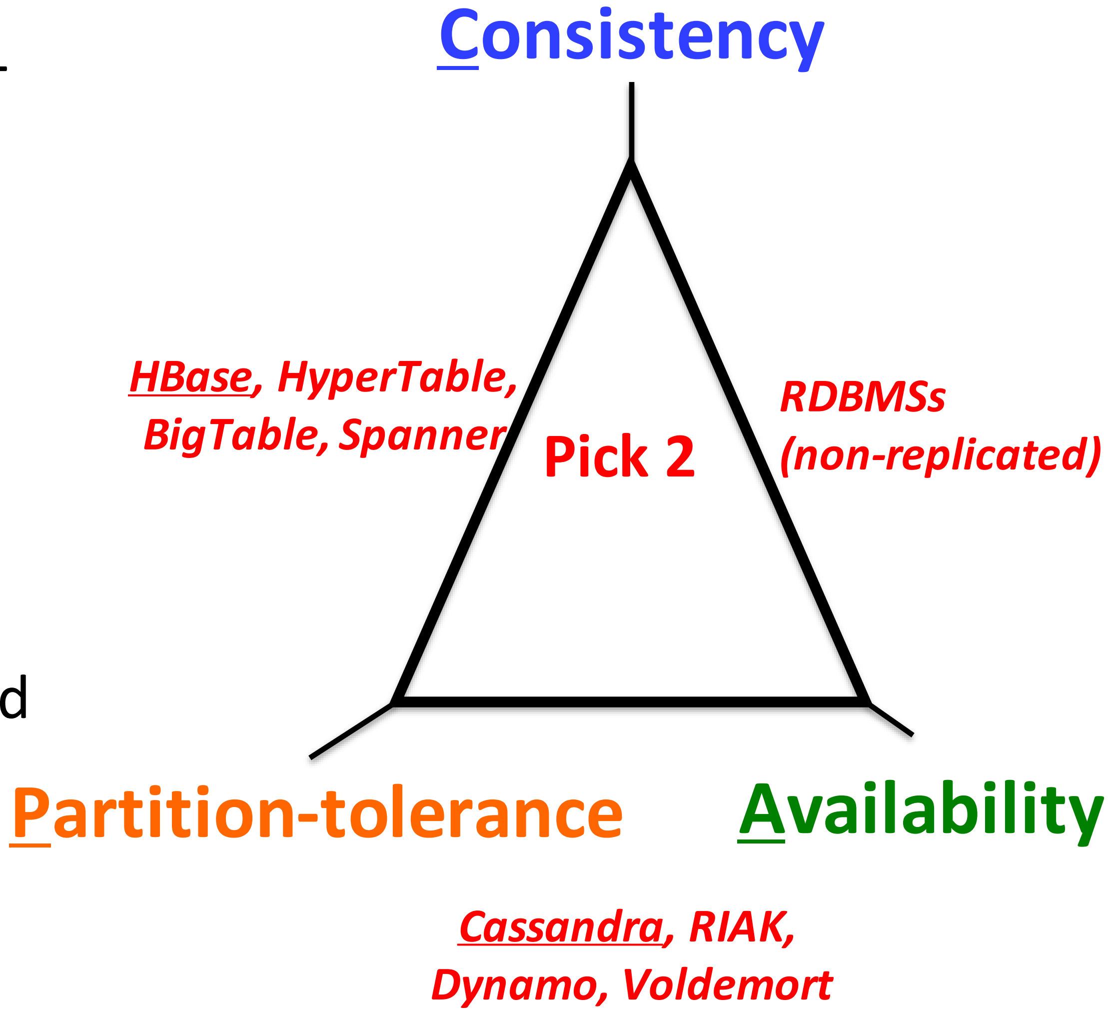
- Since partition-tolerance is essential in today's cloud computing systems, CAP theorem implies that a system has to choose between consistency and availability

- **Cassandra**
  - Eventual (weak) consistency, Availability, Partition-tolerance

- **Traditional RDBMSs**
  - Strong consistency over availability under a partition

CAP Tradeoff

- Starting point for NoSQL Revolution
- A distributed storage system can achieve **at most two of C, A, and P.**
- When partition-tolerance is important, you have to choose between consistency and availability



Eventual Consistency

- If all writes stop (to a key), then all its values (replicas) will converge eventually.
- If writes continue, then system always tries to keep converging.
    - **Moving "wave" of updated values lagging behind the latest values sent by clients, but always trying to catch up.**
- May still return stale values to clients (e.g., if many back-to-back writes).
- But works well when there a few periods of low writes – system converges quickly.

RDBMS vs. Key-value stores

- While RDBMS provide **ACID**
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

- Key-value stores like Cassandra provide **BASE**
  - **B**asically **A**vailable **S**oft-state **E**ventual Consistency
  - Prefers Availability over Consistency

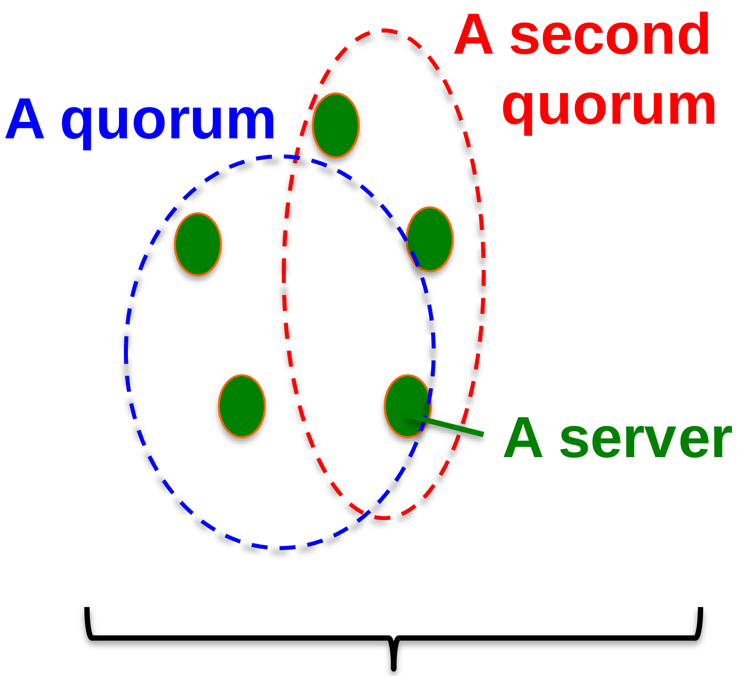
Consistency in Cassandra

- Cassandra has **consistency levels**
- Client is allowed to choose a consistency level for each operation (read/write)
  - **ANY:** any server (may not be replica)
    - Fastest: coordinator caches write and replies quickly to client
  - **ALL:** all replicas
    - Ensures strong consistency, but slowest
  - **ONE:** at least one replica
    - Faster than ALL, but cannot tolerate a failure
  - **QUORUM:** quorum across all replicas in all datacenters (DCs)
    - What?

Quorums for Consistency

In a nutshell:

- Quorum = majority
  - > 50%
- Any two quorums intersect
  - Client 1 does a write in red quorum
  - Then client 2 does read in blue quorum
- At least one server in blue quorum returns latest write
- Quorums faster than ALL, but still ensure strong consistency



Five replicas of a key-value pair

Quorums in Detail

- Several key-value/NoSQL stores (e.g., Riak and Cassandra) use quorums.
- **Reads**
  - Client specifies value of **R** ($\leq$ N = total number of replicas of that key).
  - R = read consistency level.
  - Coordinator waits for R replicas to respond before sending result to client.
  - In background, coordinator checks for consistency of remaining (N-R) replicas, and initiates read repair if needed.

Quorums in Detail (Contd..)

- Writes come in two flavors
  - Client specifies W (≤ N)
  - W = write consistency level.
  - Client writes new value to W replicas and returns. Two flavors:
    - Coordinator blocks until quorum is reached.
    - Asynchronous: Just write and return.

Quorums in Detail (Contd.)

- R = read replica count, W = write replica count
- Two necessary conditions:
  1. **W+R > N**
  2. **W > N/2**
- Select values based on application
  - **(W=1, R=1):** very few writes and reads
  - **(W=N, R=1):** great for read-heavy workloads
  - **(W=N/2+1, R=N/2+1):** great for write-heavy workloads
  - **(W=1, R=N):** great for write-heavy workloads with mostly one client writing per key

Cassandra Consistency Levels (Contd.)

- Client is allowed to choose a consistency level for each operation (read/write)
  - ANY: any server (may not be replica)
    - Fastest: coordinator may cache write and reply quickly to client
  - ALL: all replicas
    - Slowest, but ensures strong consistency
  - ONE: at least one replica
    - Faster than ALL, and ensures durability without failures
  - **QUORUM:** quorum across all replicas in all datacenters (DCs)
    - Global consistency, but still fast
  - **LOCAL_QUORUM:** quorum in coordinator's DC
    - Faster: only waits for quorum in first DC client contacts
  - **EACH_QUORUM:** quorum in every DC
    - Lets each DC do its own quorum: supports hierarchical replies

Types of Consistency

- Cassandra offers **Eventual Consistency**
- Are there other types of weak consistency models?

Consistency Solutions

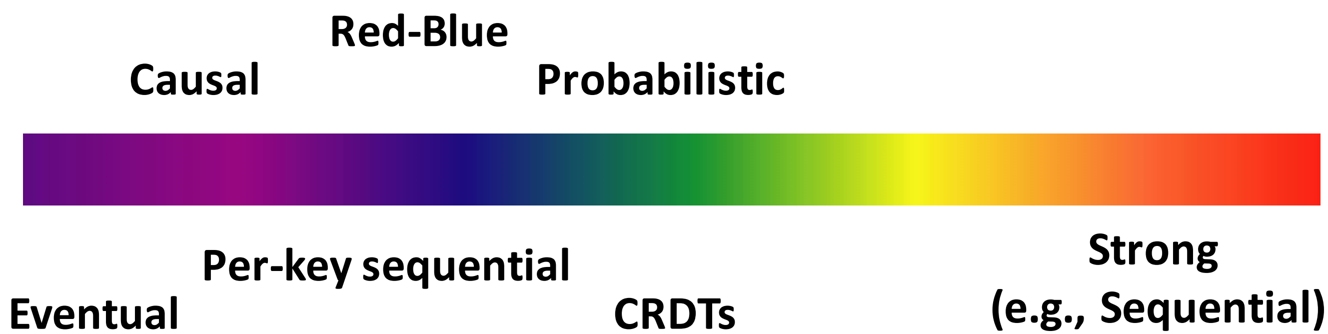
Consistency Solutions

Eventual Consistency

- Cassandra offers **Eventual Consistency**
  - If writes to a key stop, all replicas of key will converge
  - Originally from Amazon's Dynamo and LinkedIn's Voldemort systems

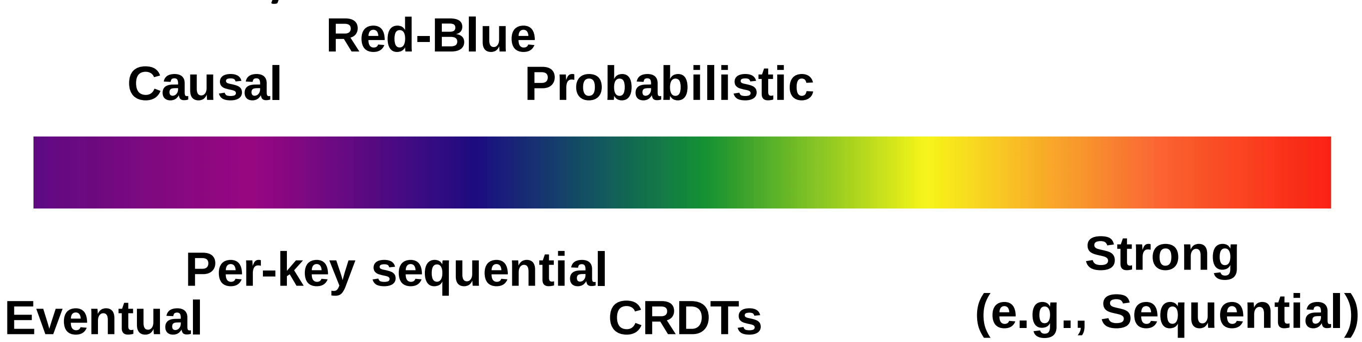
Newer Consistency Models

- Striving towards strong consistency
- While still trying to maintain high availability and partition-tolerance



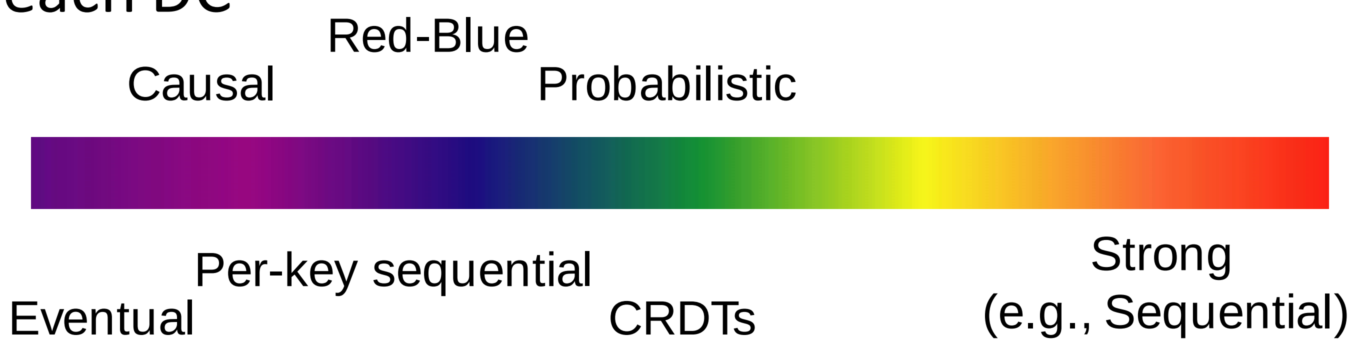
Newer Consistency Models (Contd.)

- **Per-key sequential:** Per key, all operations have a global order
- **CRDTs** (Commutative Replicated Data Types): Data structures for which commutated writes give same result [INRIA, France]
    - E.g., value == int, and only op allowed is +1
    - Effectively, servers don't need to worry about consistency



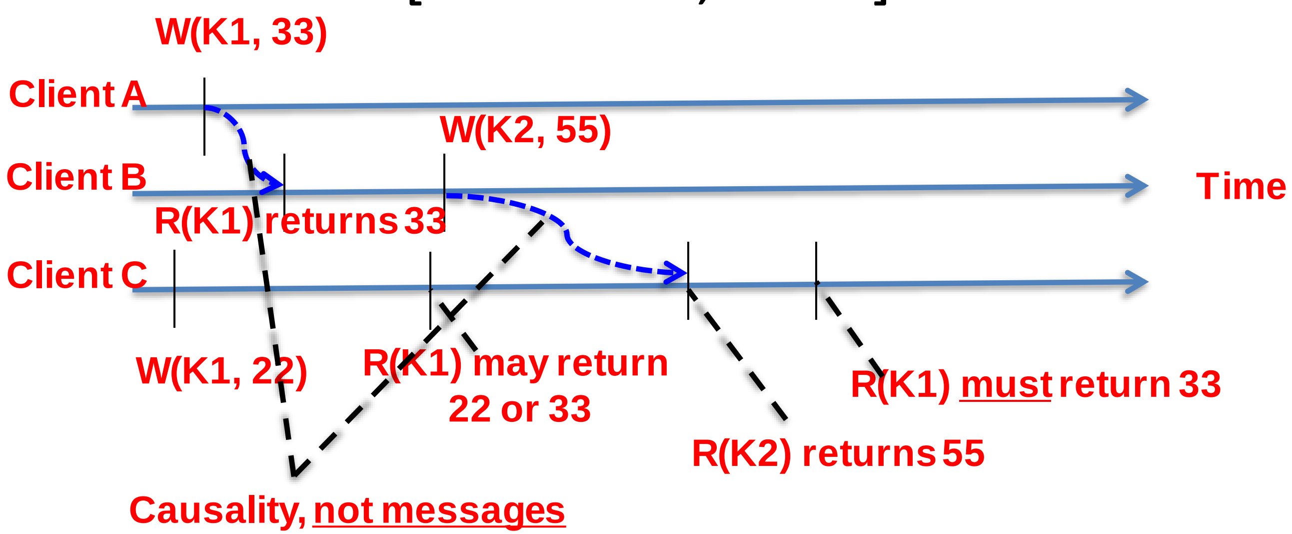
Newer Consistency Models (Contd.)

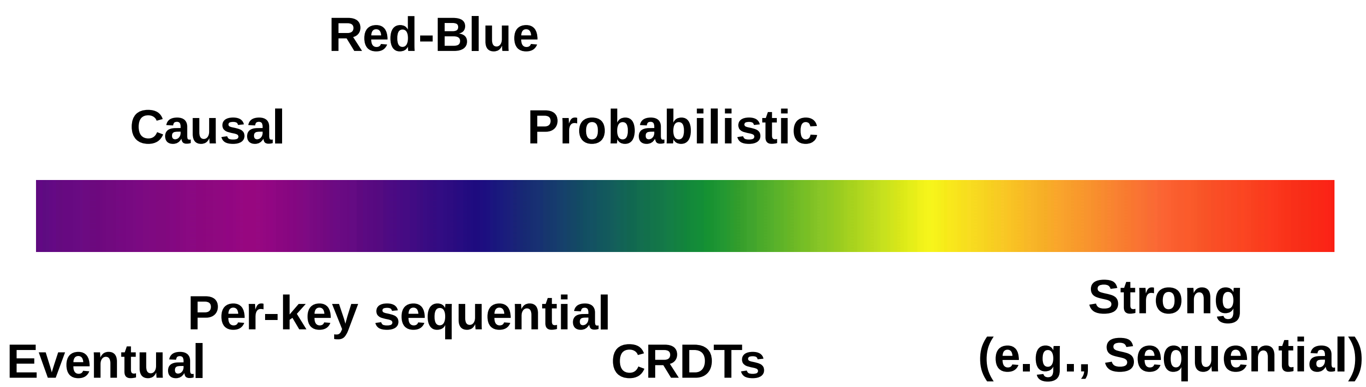
- **Red-blue Consistency**: Rewrite client transactions to separate operations into red operations vs. blue operations [MPI-SWS Germany]
    - Blue operations can be executed (commutated) in any order across DCs
    - Red operations need to be executed in the same order at each DC



Newer Consistency Models (Contd.)

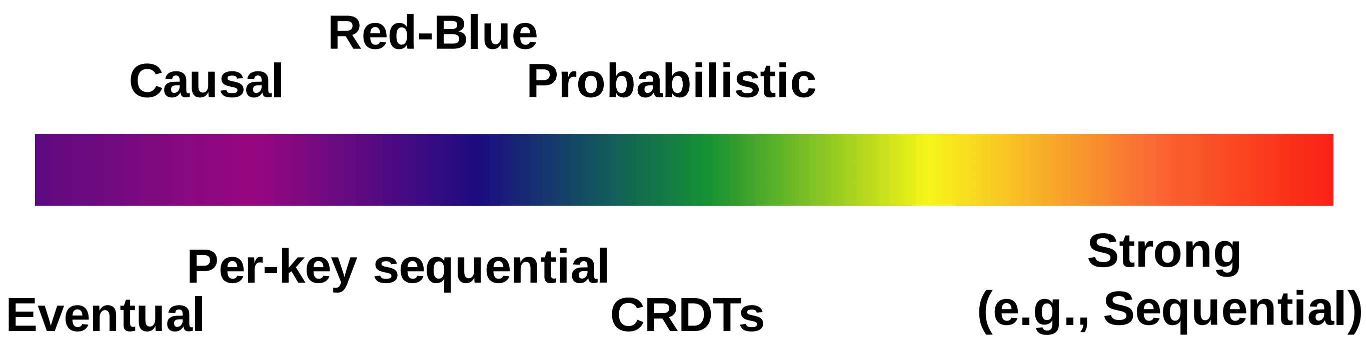
Causal Consistency: Reads must respect partial order based on information flow [Princeton, CMU]





Which Consistency Model should you use?

- Use the lowest consistency (to the left) consistency model that is “correct” for your application
  - Gets you fastest availability



Strong Consistency Models

- **Linearizability:** Each operation by a client is visible (or available) **instantaneously** to all other clients
  - Instantaneously in real time
- **Sequential Consistency** [Lamport]:
  - *"... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*
  - After the fact, find a "reasonable" ordering of the operations (can re-order operations) that obeys sanity (consistency) at all clients, and across clients.
- Transaction ACID properties, example: newer key-value/NoSQL stores (sometimes called **"NewSQL")**
  - Hyperdex [Cornell]
  - Spanner [Google]
  - Transaction chains [Microsoft Research]

Conclusion

- Traditional Databases (RDBMSs) work with strong consistency, and offer ACID
- Modern workloads don't need such strong guarantees, but do need fast response times (availability)
- Unfortunately, CAP theorem
- **Key-value/NoSQL systems offer BASE**
  [**B**asically **A**vailable **S**oft-state **E**ventual Consistency]
  - Eventual consistency, and a variety of other consistency models striving towards strong consistency
- We have also discussed the design of Cassandra and different consistency solutions.