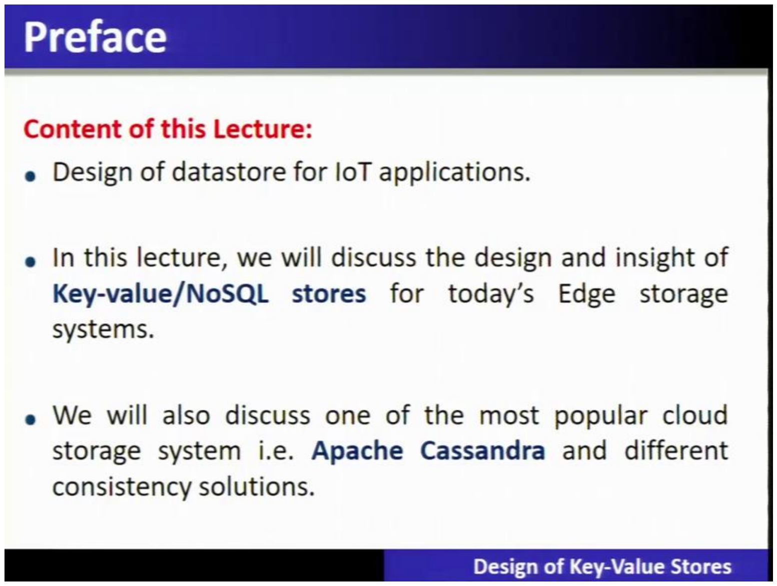
Foundation of Cloud IoT Edge ML

Professor Rajiv Misra
Department of Computer Science and Engineering
Indian Institute of Technology, Patna
Lecture 19
Design of Key Value Stores for IoT Edge Storage

Myself Doctor Rajiv Misra. I am from IIT Patna. the topic of this lecture is Design of Key-Value Stores for IoT Edge Storage.

(Refer Slide Time: 00:25)



Preface

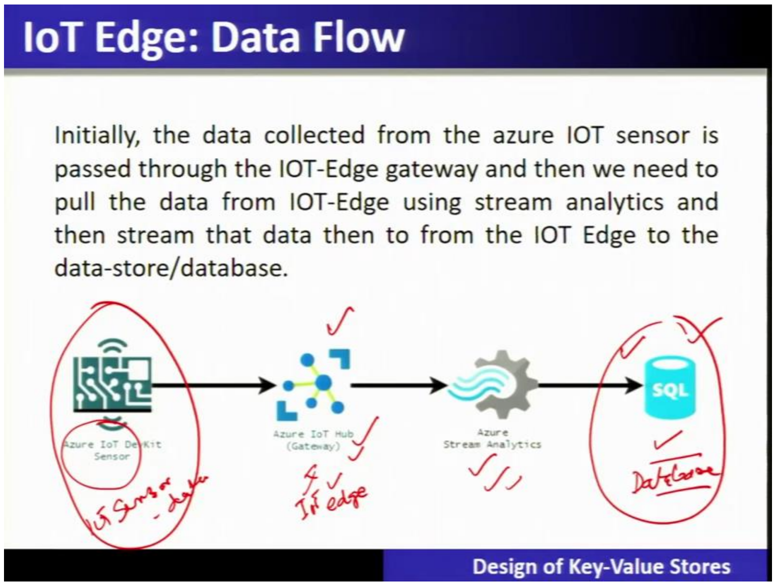
Content of this Lecture:

- 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.

Design of Key-Value Stores

Content of this lecture. In this lecture we will look up the need for design of a data store, that is databases for IoT applications. In this lecture, then we will discuss the design and insight of these key value or NoSQL stores that is used for today's edge storage system. then we will also discuss one of the most popular cloud storage system, that is Apache Cassandra, and the different Consistency solutions which may be required for different IoT applications.

(Refer Slide Time: 01:10)



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.

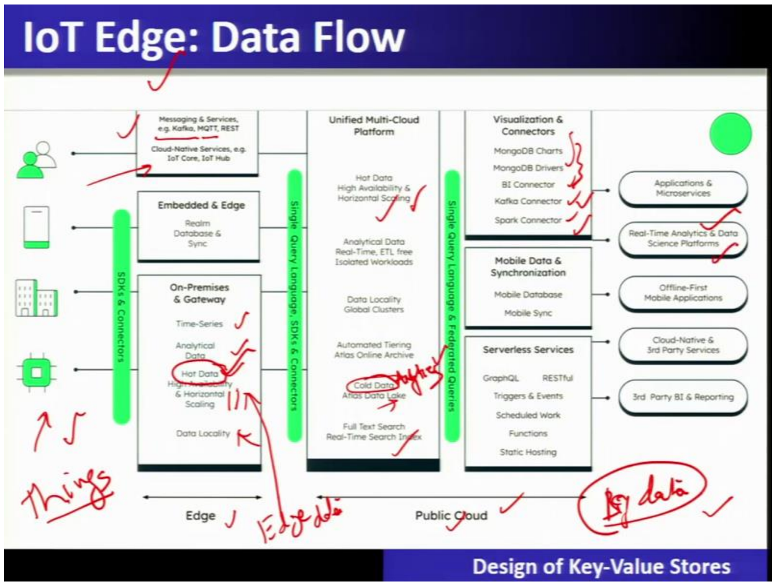
Design of Key-Value Stores

So, let us get started with the simple illustration of a data flow in an IoT system. So, initially, the data is collected from Azure IoT sensor, that is shown over here as a representative figure of all the IoT sensors. these IoT sensors used to pass through the IoT edge gateway and then we need to pull the data from IoT edge using stream analytics and then stream that data to and from IoT edge to the data store or the databases.

So, this particular IoT sensor is the source of this particular data in an IoT system. This particular data will be now go through the IoT hub, that is the gateway or you can also say that it is an IoT edge. After that it will be used for the stream analytics, and then this particular data will be stored in the database for further actions on it. So, therefore for every IoT system, you may require an appropriate data store or a database. And we have shown you that IoT sensor through the edge, that is IoT edge, in this example we have taken Azure IoT hub, so Azure IIT hub is having the edge offering also.

So, cloud and edge, both the offerings are there, and then there will be using that in stream analytics. And finally, this particular data will be stored into the database whether this database will be at the edge or at the cloud depending upon the situation. We will discuss this in more detail how different technologies will provide this kind of sensor data storage.

(Refer Slide Time: 03:46)



So, this is an overall data flow diagram. So, you can see there is an edge, there is a public cloud, two entity. And here we can say it is a things. So, you can see that the IoT sensor which is embedded across all the things, will be the source of the data generation. So, the data will be now either using the data ingestion technologies like Kafka or using telemetry that is MQTT protocol and rest APIs will be used to pull the data out from the sensor source, and then it will be taken up into an edge oblique public cloud system.

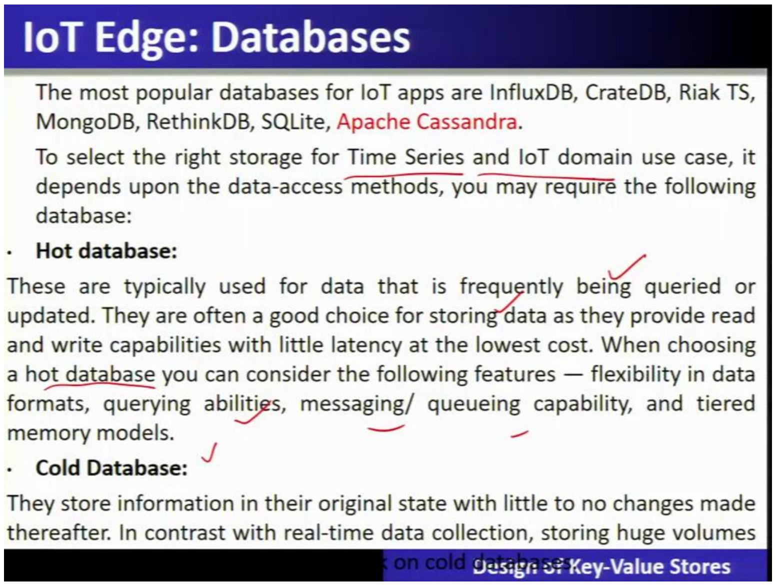
Now, you can see that this particular data often is called a time-series data and this particular data sometimes require as the analytical data for the different purposes. So, this data, where once it is inside the system, immediately if it requires the processing, they are called the hot data. And for this hot data, there is a requirement that storage has to be scalable horizontally. So, therefore, this particular type of situation is called the edge datacenter will provide the elastic data store at the edge also and support the data locality.

Now, crossing this edge when the data moves into the public cloud system, this particular data will be stored somewhere into the data store. And that is also using the technologies which is called data lake. So, once the data is stored somewhere, it becomes the issue for cold data analytics. So, cold data analytics also requires sometimes these Availability and horizontal scaling. So, that particular data is stored also requires this kind of situations where you require the enormous amount of resources somehow, because you can see that hundreds and thousands of IoT devices are sending the data through the edge to the cloud. So, you require both at the edge, a horizontal scale based the data store.

Similarly, at the cloud also you require that particular system. So, these kind of system which supports is horizontally scaling or scalability is called the big data platform. So, big data technologies are supporting this kind of data store which is used for an IoT edge system. Now, then we can see different kind of connectors are very much needed. One is the connectors for the public cloud store and the big data store. Similarly, for data ingestion also you require a connector. Similarly, for big data store, you require a connector. So, all these are specified here into the public cloud system.

Finally, these particular data which is stored can further be used for model training and performing the analytics whether it is real time analytics if it is a hot data analytics or it is the cold data analytics, where you can get the insights at later stage. And then after the inside actions are to be derived for business intelligence. So, therefore let us get started into more detail about this.

(Refer Slide Time: 08:00)



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

[redacted] on cold databases

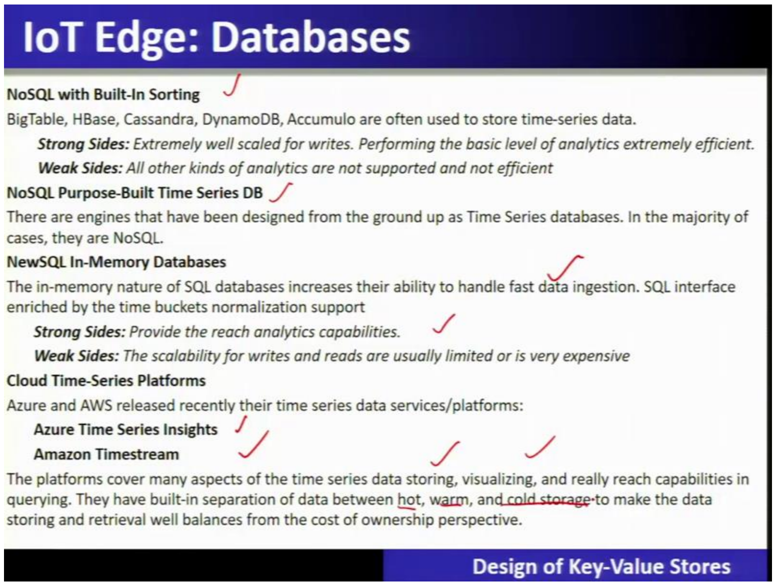
Design of Key-Value Stores

So, the most popular databases for IoT applications are InfluxDB, CrateDB, Riak TS, MongoDB, then RethinkDB, SQLite, Apache Cassandra. To select the right storage for the Time Series data and in the IoT domain use case, it depends upon the data access methods which is given as follows. So, hot database means typically used for the data that is frequently being queried and updated. And as the data enters into this system of edge and cloud these kind of queries are analysis is to be supported instantly.

So, they are also good choice for storing the data as they provide read and write capabilities with little latency at a low cost. So, when choosing the hot database, we have to consider several important features such as querying ability, messaging/queuing capabilities tiered memory models and so on. Now, the next important store is called cold databases.

So, they store the information in their original state with little to no changes made thereafter. So, in contrast to the real time data store these kind of data which is called the cold storage requires the huge storage volumes and often being used for model training of a machine learning models at a later point of time.

(Refer Slide Time: 09:42)



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.

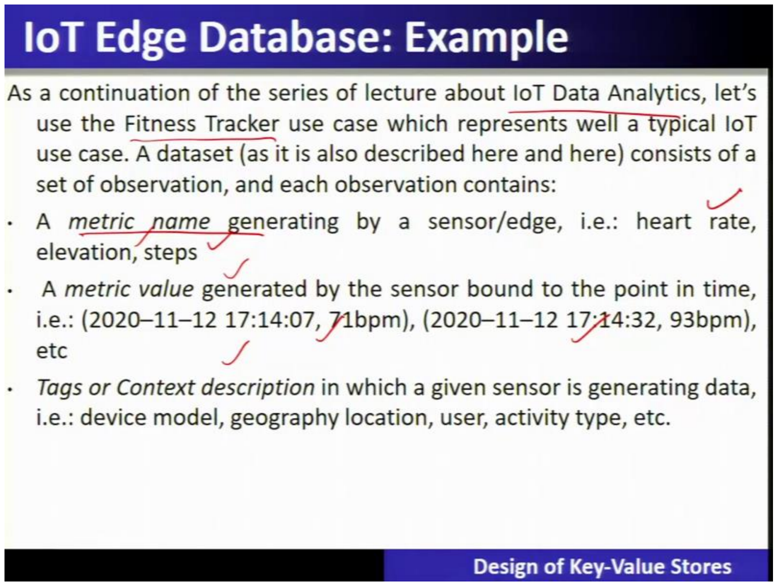
Design of Key-Value Stores

So, now let us go into more detail about cold storage data. Cold databases will use the technologies such as NoSQL with a built-in sorting. So, the examples of such technologies are BigTable, HBase, Cassandra, DynamoDB, and they are often used to store the time series data with a huge volume involved within it. So, this particular kind of storage which is required for IoT system, they are extremely and well scaled for the writes. And these particular different kinds of analytics are not supported and not efficient sometimes, but nevertheless it gives a way for scalable writes in very efficient manner.

So, NoSQL purpose-build time series databases are also available. So, there are several engines which supports for time series databases, and most of them are NoSQL. So, we are going to discuss them in more detail. NewSQL in-memory databases. So, in-memory nature of SQL databases increases their ability to handle fast data ingestion, SQL interface enriched by the time bucket normalization support and so on. So, these are very much required for to supporting up as the infrastructure for analytics capabilities.

So, the cloud time series platforms which different cloud vendors they support, such as Azure and AWS, they have recently released their similar databases that is called Azure Time Series Insights and Amazon Timestream. All these platforms supports time series data storing, visualization and other capabilities for querying. Now, they have been built in separation of the data between hot warm and cold storage to make the data store and retrieval well balanced from the cost of ownership perspective.

(Refer Slide Time: 11:52)



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:

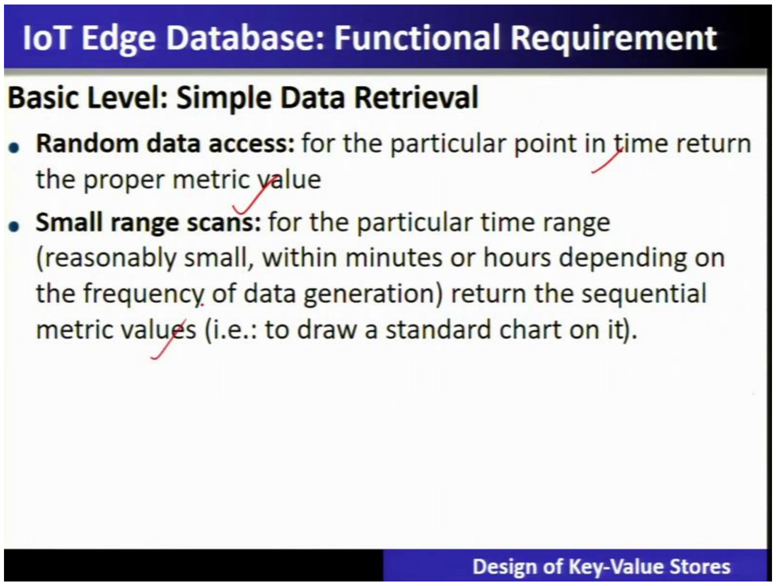
- A *metric name* generating by a sensor/edge, i.e.: heart rate, elevation, steps
- A *metric value* generated by the sensor bound to the point in time, i.e.: (2020–11–12 17:14:07, 71bpm), (2020–11–12 17:14:32, 93bpm), etc
- *Tags or Context description* in which a given sensor is generating data, i.e.: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

Design of Key-Value Stores

Let us see this kind of example of such situations. So, in this continuation, we are considering let us say a use case of IoT data analytics. And let us use a fitness tracker. Normally, it comes as an IoT gadget which people can wear and then these particular sensors within it can send the data which is now further processed as the IoT data analytics.

So, the data set consist of a set of observations such as this particular data will contain the metric name generated by the sensor or edge such as heart rate, elevation, steps and so on. These metric values which are generated by this sensor bound to the point in time. So, therefore timestamps are also to be attached with these kind of fitness tracker generated data set. Now, the tags and the context description in which a given sensor is generating the data such as device model, geographic location, user, activity type is also to be included by these IoT devices.

(Refer Slide Time: 13:10)



IoT Edge Database: Functional Requirement

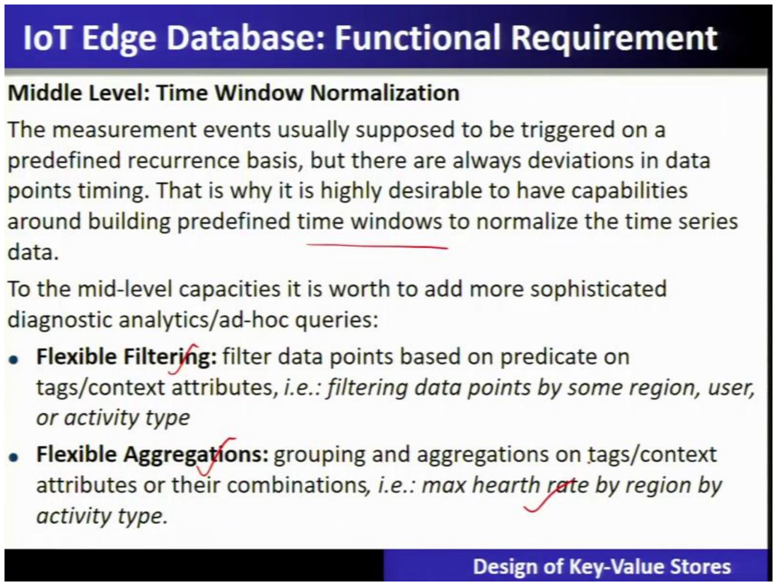
Basic Level: Simple Data Retrieval

- **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).

Design of Key-Value Stores

Now, then, this particular data required some functional needs for data retrieval. So, let us see a very basic level of data retrieval includes the random data access for a particular point in time and return the proper metric value. Similarly, a small range also sometimes based query is needed and return the sequential metric values.

(Refer Slide Time: 13:36)



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:

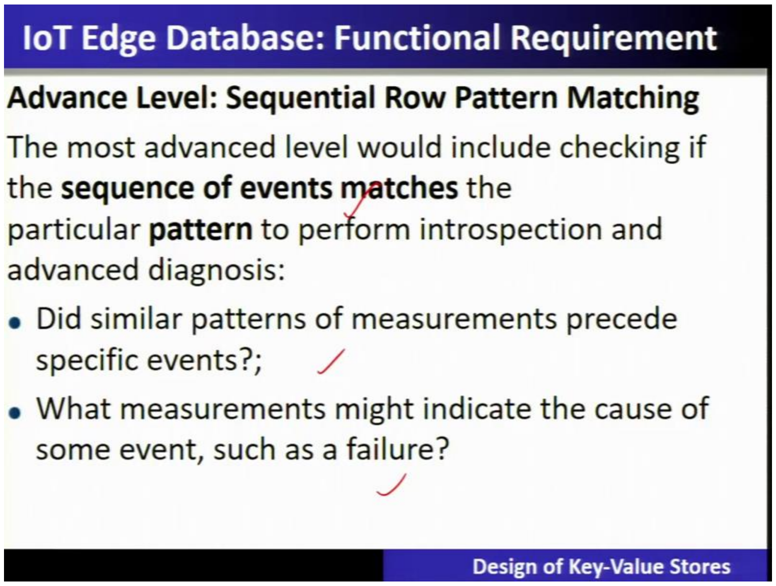
- **Flexible Filtering:** filter data points based on predicate on tags/context attributes, *i.e.: filtering data points by some region, user, or activity type*
- **Flexible Aggregations:** grouping and aggregations on tags/context attributes or their combinations, *i.e.: max hearth rate by region by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

Design of Key-Value Stores

Middle level, that is the time window normalization applied technologies or techniques to be supported for the measurement events usually supported, supposed to be triggered on predefined recurrence basis but they are always deviation in the data points timings. That is why it is highly desirable to have these capabilities around and predefined time windows to normalize this time series data.

So, mid level database capacities such as a flexible filtering and flexible aggregation, this is sometimes needed. For example, when you say flexible filtering we mean that filter the data point based on predicate on tags and context attributes. That is filtering data points by some region, user or activity type flexible aggregation we mean that grouping and aggregation on tags and context or their combinations. That is, maximum heart rate by region, activity type and so on.

(Refer Slide Time: 14:47)



IoT Edge Database: Functional Requirement

Advance Level: Sequential Row Pattern Matching

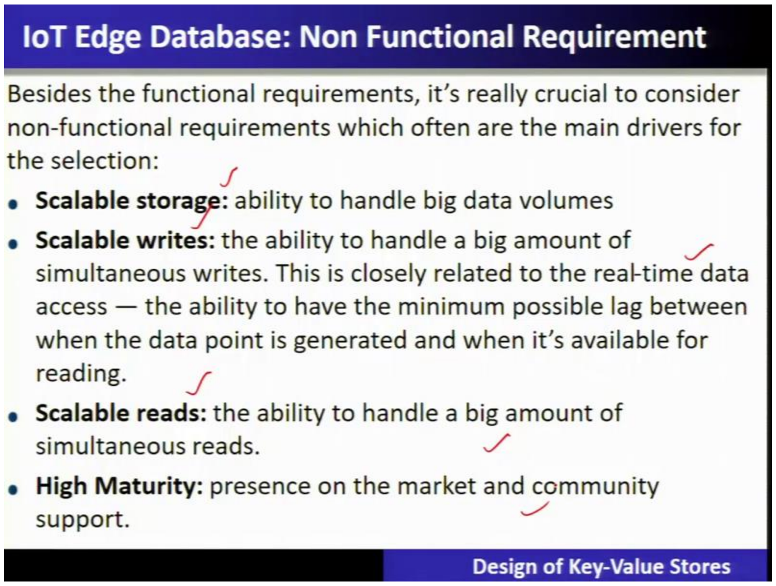
The most advanced level would include checking if the **sequence of events matches** the particular **pattern** to perform introspection and advanced diagnosis:

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

Design of Key-Value Stores

Whereas the advanced level requirement comes in the form of sequential row pattern matching and checking if the sequence of events matches the particular pattern to perform the introspection and advanced diagnosis. So, diagnosis includes some of these questions which require to be answered during the analysis stage. Did some pattern of measurement proceeds specific event, what measurements might indicate the cause of some events such as failure? So, all this is required and analysis into the time series data.

(Refer Slide Time: 15:26)



IoT Edge Database: Non Functional Requirement

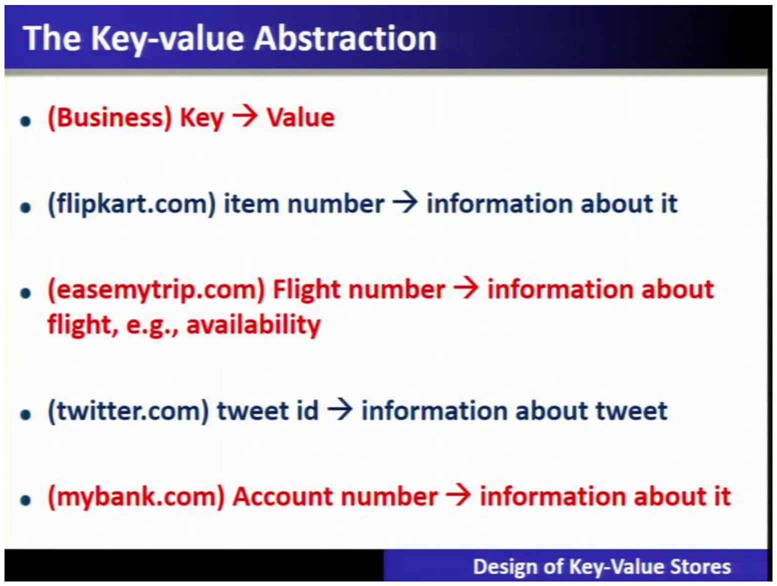
Besides the functional requirements, it's really crucial to consider non-functional requirements which often are the main drivers for the selection:

- **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.

Design of Key-Value Stores

And that requires the storage, which has this kind of querying very efficiently or retrieving the data efficiently. So, therefore it requires stable storage ability to handle big volumes, it requires a scalable writes, that is ability to handle the big amount of simultaneous writes. So, this is closely related to the real time data access and then it is also to be supporting the scalable reads, that is the ability to handle big amount of simultaneous reads, and high maturity that is in the presence of market and community.

(Refer Slide Time: 16:05)



The Key-value Abstraction

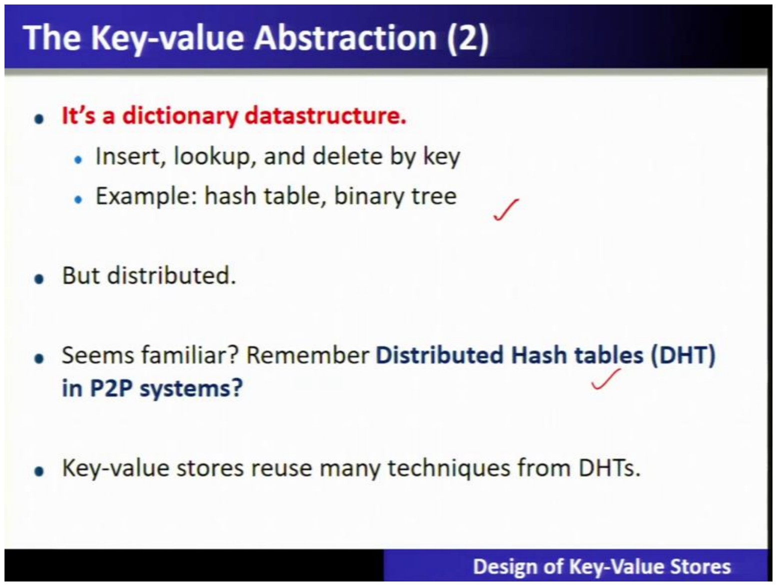
- (Business) Key → Value
- (flipkart.com) item number → information about it
- (easemytrip.com) Flight number → information about flight, e.g., availability
- (twitter.com) tweet id → information about tweet
- (mybank.com) Account number → information about it

Design of Key-Value Stores

So, therefore let us discuss a particular design of databases which can support all these functionalities, and that is called Key-value abstraction. So, what we mean by the key-value abstraction? The key-value abstraction means that from the, there is a key associated with the value that is the data stored is organized in this particular form, that is key-value store.

So, what do you mean by the key for a particular business? We will see that if it is a flipkart.com, key means the item number and value means the information about those items. Similarly, if it is a easemytrip.com, the key means that flight number and the value means the information about the flight, availability and all other details. Similarly, twitter.com, the key will become the tweet id and the information about the tweet. mybank.com is an account number and the information about it.

(Refer Slide Time: 17:03)


The Key-value Abstraction (2)

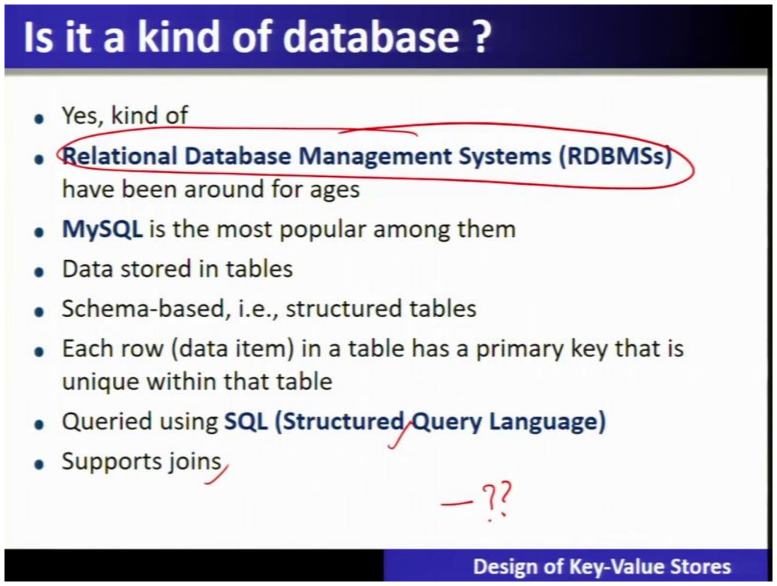
- **It's a dictionary datastructure.**
  - Insert, lookup, and delete by key
  - Example: hash table, binary tree ✓
- But distributed.
- Seems familiar? Remember **Distributed Hash tables (DHT) in P2P systems?** ✓
- Key-value stores reuse many techniques from DHTs.

Design of Key-Value Stores

So, this key-value store becomes a dictionary data structure which supports the operations like insert, lookup and delete by key. And these typical examples are the hash table or the binary tree. But this particular dictionary data structures, what is needed here in this particular situation of supporting the IoT data store, and also of a big size that is a big data store, using supported by the big data technologies. So, they are required to be distributed. So, that kind of data, dictionary data structure which is centralized is not going to be used in the IoT data store. It has to be a distributed one.

So, the distributed data store which supports this distributed technologies, is that the distributed hash table implementation, but that is on the peer-to-peer system design. So, the design of these data stores are quite different and they are called distributed hash tables. So, key-value store reuse many technologies from the distributed hash table. So, that we are going to see.

(Refer Slide Time: 18:20)



Is it a kind of database ?

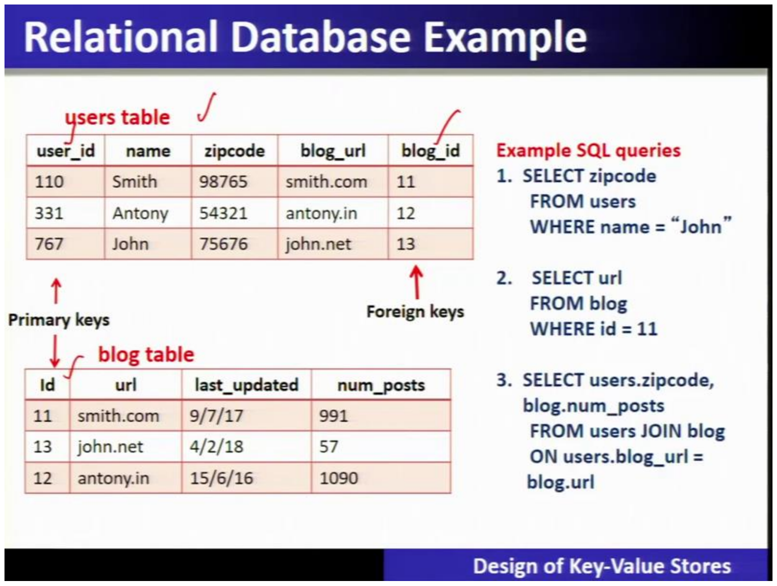
- Yes, kind of
- **Relational Database Management Systems (RDBMSs)** have been around for ages
- **MySQL** is the most popular among them
- Data stored in tables
- Schema-based, i.e., structured tables
- Each row (data item) in a table has a primary key that is unique within that table
- Queried using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

— ??

Design of Key-Value Stores

So, it is a kind of database such as relational database management system, was there around for the ages, and MySQL is a popular interface of accessing it due to the very close to all the programmers around the world, using this MySQL interface. So, data is stored in RDBMS in the form of table, schema-based it is, and each row in a table has a primary key that is unique within that particular table and is queried using structure query language and supports the join. So, this is the traditional RDBMS, we have talked about, but this is not going to be used or useful in this IoT data store which requires a big data technologies for this databases.

(Refer Slide Time: 19:10)



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

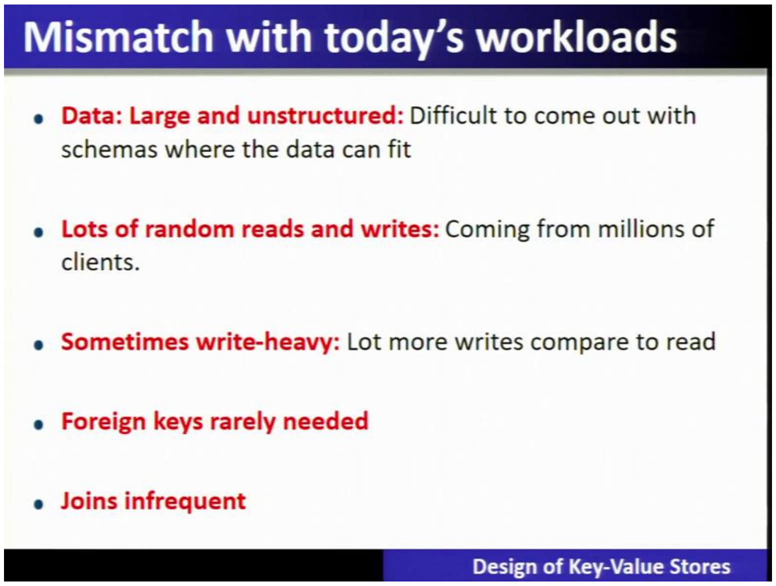
Example SQL queries

```
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
```

Design of Key-Value Stores

So, in the relational databases you know that the tables, the data is organized in the form of a tables and these tables are called, one of these attribute is called the primary key and one more attribute is called the foreign key which will become the primary key of another table and it supports the join operation.

(Refer Slide Time: 19:33)



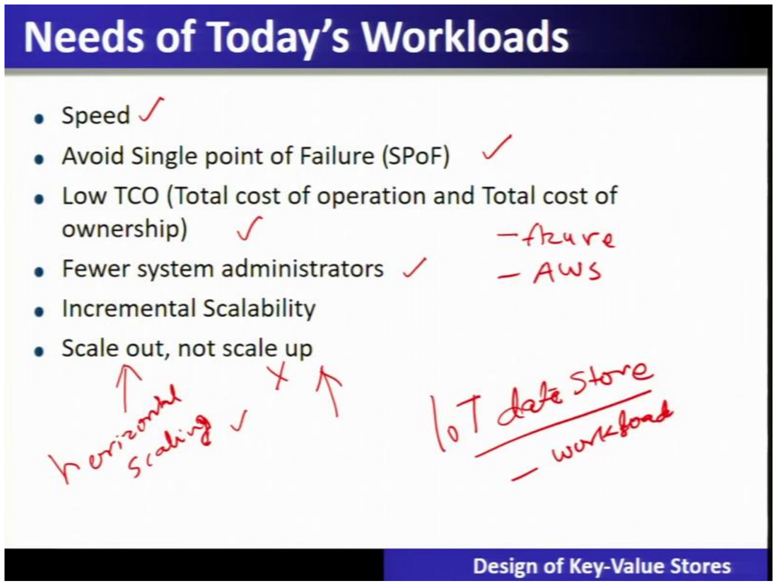
Mismatch with today's workloads

- **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**

Design of Key-Value Stores

But it will mismatch with the today's workload which is an IoT data store workload, which is categorized as the IoT data which is a large and unstructured. That is comes from, come out with the schemas where the data can fit. So, this particular type of workload has lot of random reads and writes coming from millions of IoT devices. Sometimes they are write-heavy and the foreign keys are rarely used and join is infrequent.

(Refer Slide Time: 20:12)



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

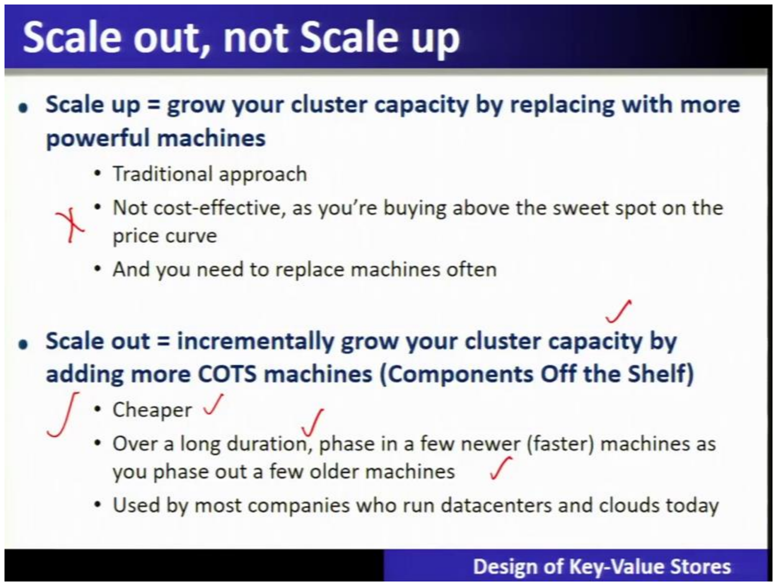
– Azure
– AWS
horizontal scaling ✓
IoT data store
– workload
Design of Key-Value Stores

Therefore, the today's workload which is an IoT data is required say speed and avoid a single point of failure and also it is a low cost of operation to the ownership, fewer administration and incremental scalability scale out and not scale up. So, when you say scale out, that means it is horizontal scaling. Horizontal scaling means as more resources are needed, a board system in the form of nodes can be added, and this kind of requirement-based scaling is called horizontal scaling.

Whereas scale-up means that if you have a system, let us say desktop, you require more resources, then you have to replace the desktop with, let us say that, a bigger machine. So, that is called scale-up technologies which is not required in support of today's workloads. Now, another thing is about incremental scalability that is being supported by horizontal scaling.

And fewer administrations means most of the operations are automatic and the total cost of operation, total cost of ownership is there and therefore lot of cloud providers, they provide this kind of solutions. Let us say Azure and AWS are some of the popular services by the cloud providers, avoid single point of failure and speed. So, IoT data store is this kind of workload which requires this kind of features to be supporting.

(Refer Slide Time: 22:06)



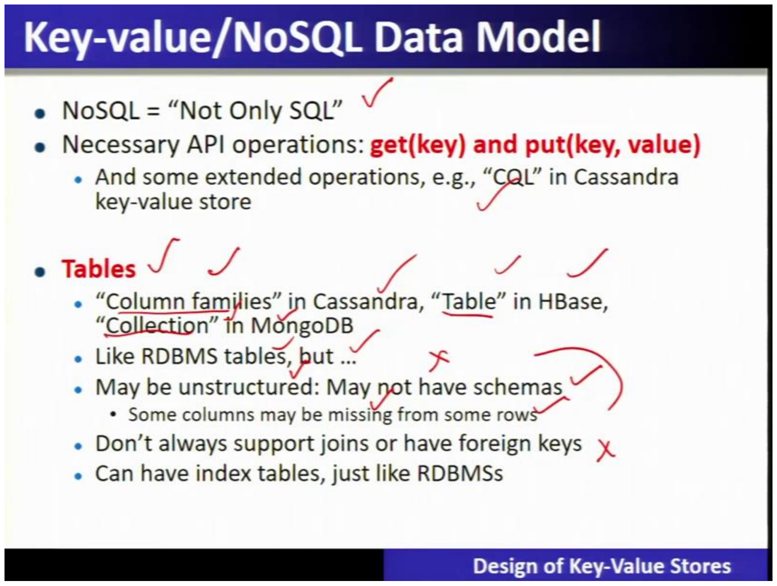
Scale out, not Scale up

- 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

Design of Key-Value Stores

So, scale up and scale out. So, scale up is not needed in these kind of workloads, which is an IoT workload. Scale out is there. Scale out means the incremental grow your cluster capacity by adding more COTS machine, that is Component off the Shelf machine. This is the cheaper way of providing more resources for storage, and also it is, can work over a long duration and phase in a fewer, that is the faster machines, as you phase out with the older machines. And most of the companies who runs the datacenter and cloud they use, they follow this scale out technologies.

(Refer Slide Time: 22:52)



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

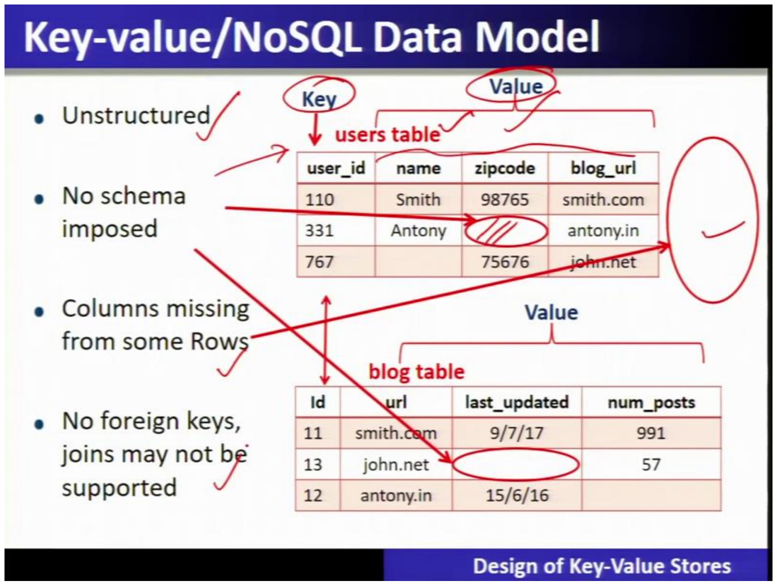
Design of Key-Value Stores

Now, this key value stores also supports a model which is called a NoSQL data model. Let us understand what you mean by NoSQL. So, the full form of NoSQL is called Not Only SQL, and this necessary API is to access this information is called get by key and put by key and value. So, some of the extended operations are for example, CQL in Cassandra key-value stores.

So, in NoSQL, the tables are called sometimes Column Families in Cassandra and Table in HBase and Collection in MongoDB. So, table has different names in NoSQL database systems or in a data model. So, the again I am repeating, Column Families, that is nothing but a table in a Cassandra, a Table as a table in HBase, a Collection as a table in MongoDB. So, all these are some form of a table but with a different name in these kind of NoSQL database.

RDBMS has a table but here, these tables are to be for unstructured data. So, maybe the data is unstructured, may not have the schemas and some columns may be missing and some rows may be missing. So, this is not fit for the RDBMS. So, RDBMS type of databases is not the requirement of supporting this IoT use case, and it requires this kind of data store which is called NoSQL data store. Similarly, this NoSQL data store also does not support require the join or a foreign keys and have the index tables like RDBMS.

(Refer Slide Time: 24:45)



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

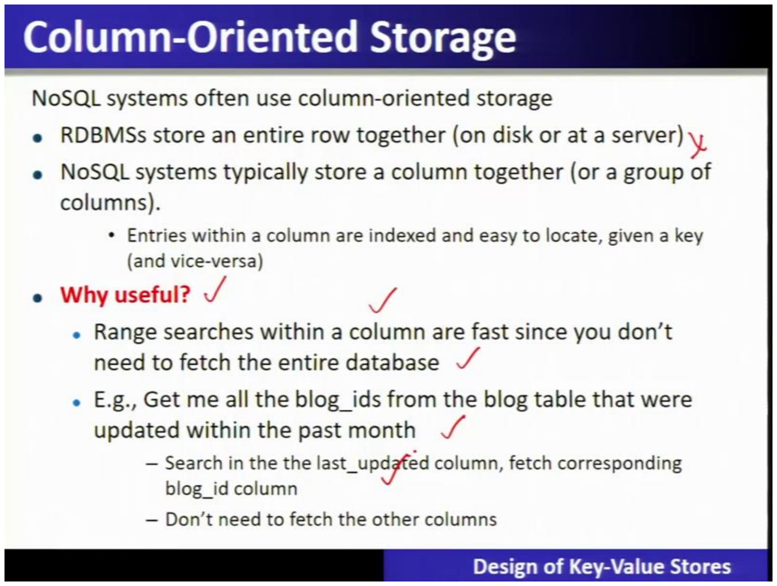
Id	url	last_updated	num_posts
11	smith.com	9/7/17	991
13	john.net		57
12	antony.in	15/6/16	

Design of Key-Value Stores

So, these particular details are explained here in this following illustration. For example, if the data is unstructured, you mean that no schema can be imposed and the data is organized in the form of a key-value store. But let us take this example of a table, let us say Cassandra Column

Families. So, user table will have this following attribute, user id will become a key and all other attributes will become the value. And here, since no schema is post, so some of the row entries or column entries may be missing. So, here you can see these are the column, from some rows, they can be missing and that is not a problem in NoSQL or unstructured data model.

(Refer Slide Time: 25:35)



Column-Oriented Storage

NoSQL systems often use column-oriented storage

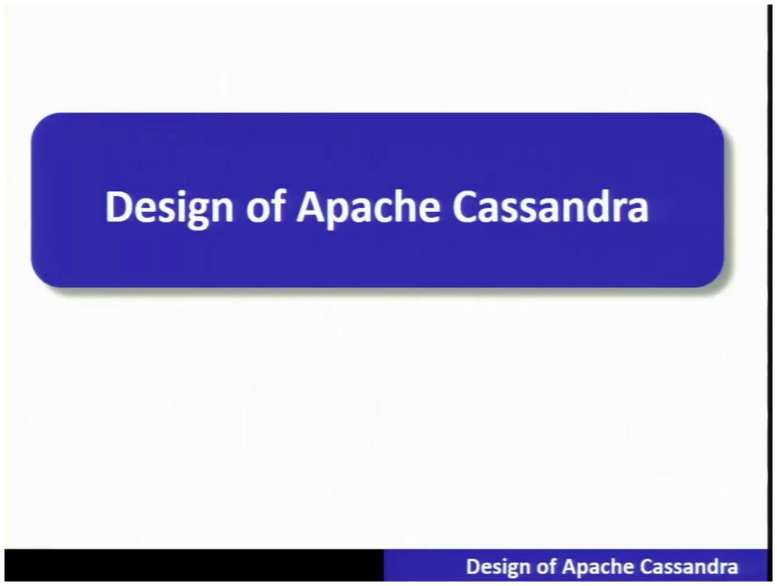
- RDBMSs store an entire row together (on disk or at a server)
- NoSQL systems typically store a column together (or a group of columns).
    - Entries within a column are indexed and easy to locate, given a key (and vice-versa)
- **Why useful?**
    - Range searches within a column are fast since you 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 Key-Value Stores

So, these are all called column-oriented storage, NoSQL system often use the column-oriented storage. RDBMS store the entire row together, that is row-oriented storage is RDBMS which is not used much for IoT scenarios. So, NoSQL system typically store the columns together or a group of columns together and entries within the columns are indexed and easy to locate given a key and vice versa.

So, why this column-oriented storage are so useful? It is useful for doing the range queries within the column or very fast, and you do not have to fetch the entire database. Only those column entries are need to be fetched. For example, if you say that get me all the blog ids from the block table that were updated within the past month. So, it will search in the last updated column and fetch the blog id and so on.

(Refer Slide Time: 26:35)

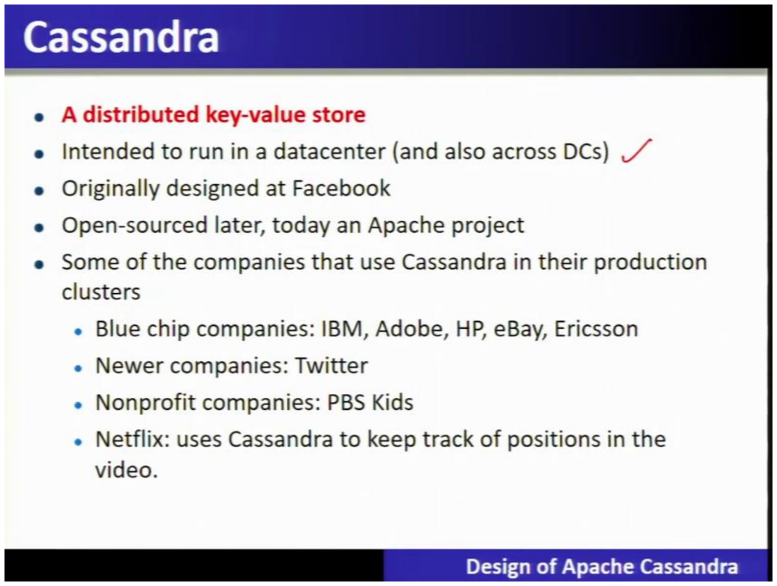


Design of Apache Cassandra

Design of Apache Cassandra

Let us go and see the design of Apache Cassandra.

(Refer Slide Time: 26:40)



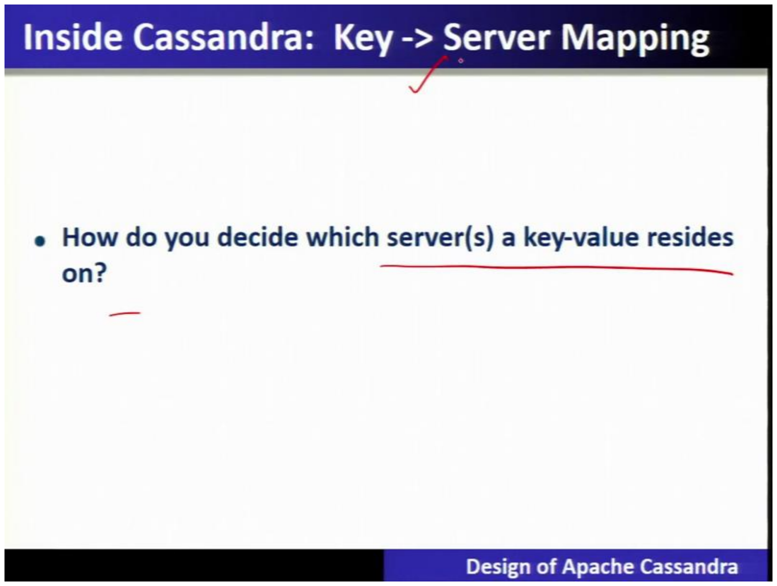
Cassandra

- 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.

Design of Apache Cassandra

So, Apache Cassandra is a distributed key-value store. It is intended to run in a datacenter, also called DCs and originally, this Cassandra is designed at the Facebook. It is open sourced later. Today, very important Apache project, some of the companies that use Cassandra in the production clusters are blue chip companies like IBM, Adobe, HP, eBay, Ericsson and newer companies like Twitter and non-profit companies like PBS Kids. Netflix uses Cassandra to keep track of positions in the videos.

(Refer Slide Time: 27:18)



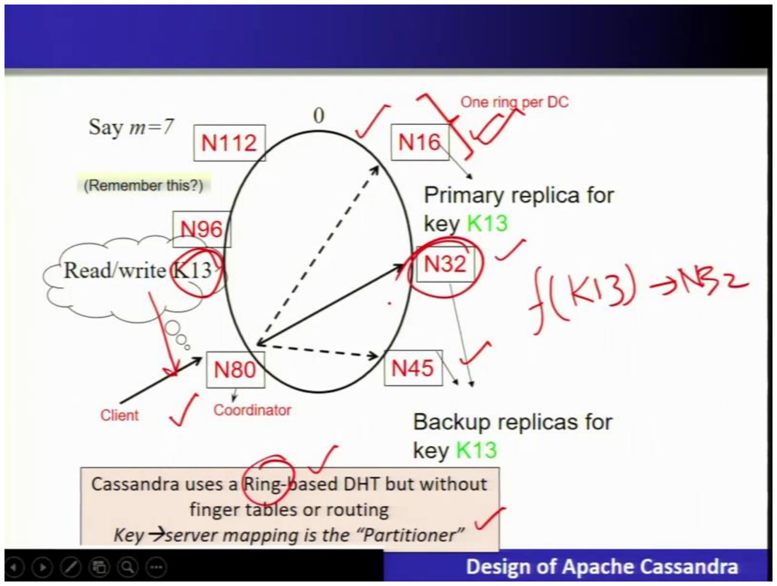
Inside Cassandra: Key -> Server Mapping

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

Design of Apache Cassandra

So, how do you decide which server a key-value store resides on? So, that means you say that this Cassandra data store is using the entire cluster and entire cluster means it will be having hundreds and thousands of nodes in the cluster. So, the question is how do you decide which server this key-value Store resides on? That is called key to the server mapping inside Cassandra.

(Refer Slide Time: 27:48)



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

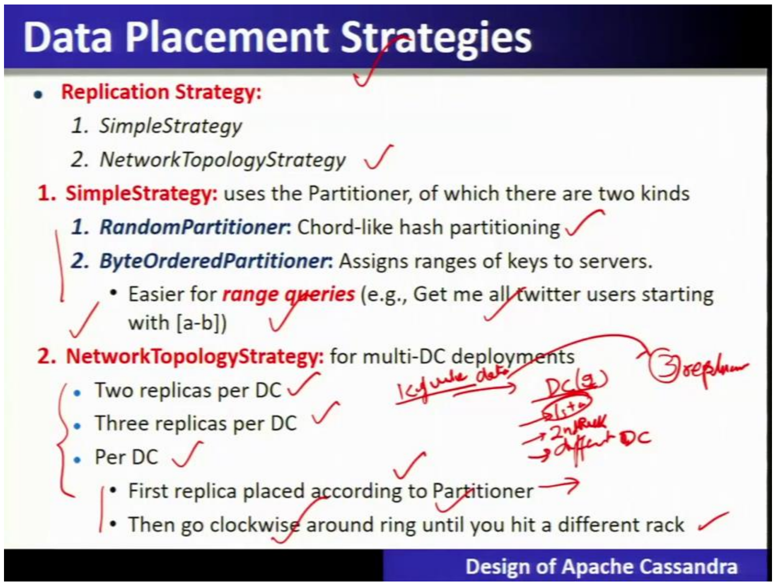
Key → server mapping is the "Partitioner"

Design of Apache Cassandra

So, key to the server mapping inside Cassandra is supported with the concept which is called a ring based DST, without other details of DST uses like finger table or the routing table. So, key to the server is mapping is called the partitioner in Cassandra. So, this is the example which you can see that if let us say you want to read and write a particular key that is K13, so this particular request goes through a client module and this particular ring will have a coordinator designated coordinator. So, it will intercept this client request, which is running at the node N80 where it gets the, to resolve the key K13 what it will do, that is what is shown over here.

So, K13 will be now put on a hash function. So, K13 will be now resolved for N16. So, the data which is stored here in the form of a ring is replicated also. So, if it is K16. So, the primary copy will be in the N16 node. And the next node, that is N32 will have its secondary replica. Similarly, N45. So, there are three different entries. One entry will be given by hash function. So, if you say that K13, if you apply to the hash function, it will give this kind of N16 entry, and it will give N32, so which is shown over here. So, N32, and then maybe a backup replicas also will be supported.

(Refer Slide Time: 29:40)



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

Design of Apache Cassandra

Now let us see if I say that a particular key-value store or a key-value entry is stored not at one place, but it is replicated. So, what is that replication strategy which is also called the data placement strategies and this is very much needed to avoid the single point of failure. So, replication strategy which is followed, which is applied in Cassandra are of two types. The first type is called simple strategy, the other is called network topology strategy.

Simple strategy uses partitioner of which there are two kinds of partitioner, one is called RandomPartitioner which is called like hash partitioning. The other is called

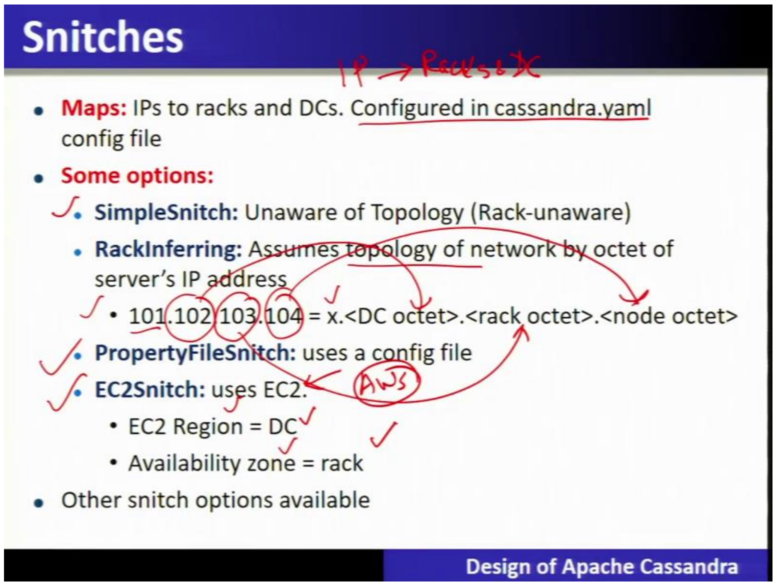
ByteOrderPartitioner which assigns the range of keys to the server. So, ByteOrderPartitioner is easier for the range queries. So, for example if the query is get me all the Twitter users starting with A to B. So, this is a kind of range query, and ByteOrderPartitioner is very good in resolving this range queries.

The other type of strategy is called the random topology strategy. This strategy is used for a multi-datacenter deployment. So, it says that two replicas per datacenter and the third replica will be on another datacenter. So, per datacenter, first replica will be placed according to the partitioner, then you go around the clockwise ring until you hit a different track. So, therefore, if you have three different replicas, so the first replica if it is pointed to a particular datacenter, so this datacenter will have two replicas which is organized in the following manner.

So, it has to be stored on let us say, first rack and then it will be stored on another rack in the same datacenter. So, the third replica will be stored on another datacenter. So, a particular key-value store, key-value data will be stored in three different replicas which are explained here with this particular rule. So, two replicas per datacenter and third replica will be in a different datacenter. Now, per datacenter the first replica will be placed according to the partitioner that is what we have seen.

And then you have to go around, go clockwise around the ring until you hit a different rack. So, that will be in the same datacenter but on a different rack. So, if a rack is not working, even then you continue to access the data if the entire datacenter is down, you can continue accessing the data because it is available in the other. So, this is called NetworkTopologyStrategy and it is quite different than the simple strategy.

(Refer Slide Time: 32:45)



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

IP → Racks & DC
AWS

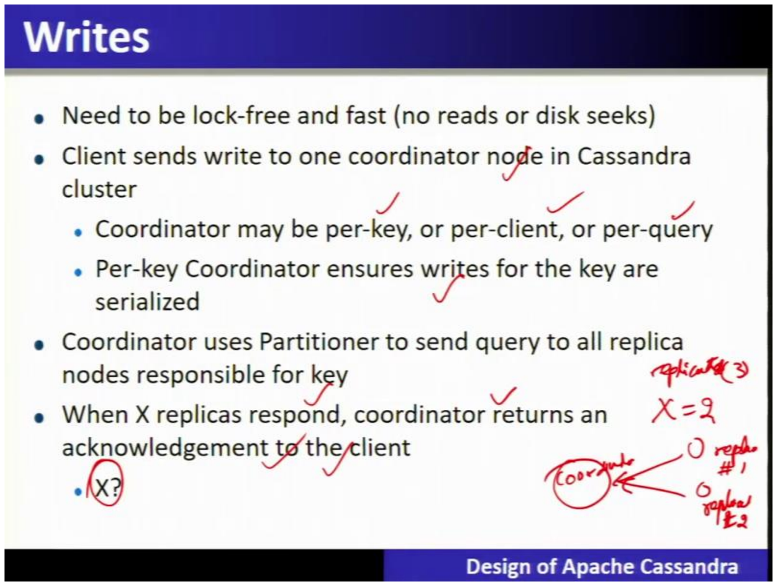
Design of Apache Cassandra

Snitches means the map from IP to the racks and datacenter. So, this is the next level of mapping which is needed from IP addresses to the racks and datacenter. So, let us see how that is done in Cassandra. So, this kind of mapping is configured in configuration file that is called Cassandra.yaml. Now, some options are there for the snitches. The first is called SimpleSnitch. That is, this SimpleSnitch is unaware of topology, that is rack-unaware snitch.

Rack-unaware snitch assumes the topology of the network by octet of servers IP addresses. For example, if the server IP address is 101.102.103.104, then you can understand that the first octet is let us say x, but the second octet, 102 represents the datacenter and the third octet represents the rack within the datacenter, and the fourth octet represents the node within the rack.

So, this kind of rack inferring is snitch. That means from the IP address you can know what is the topology of that particular network where this particular mapping of IP will takes place into the. So, this particular property file snitch uses the configuration file. So, let us say that you have an EC2 snitch. EC2 is a particular service in AWS that is Amazon Web Service cloud offering. So, EC2 snitch uses EC2, and EC2 regions is nothing but the datacenter and the Availability zone is nothing but the rack. So, therefore EC2 snitch is an example of rack inferring snitches.

(Refer Slide Time: 35:00)



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?

replication (3)

X = 2

Coordinator

replica #1

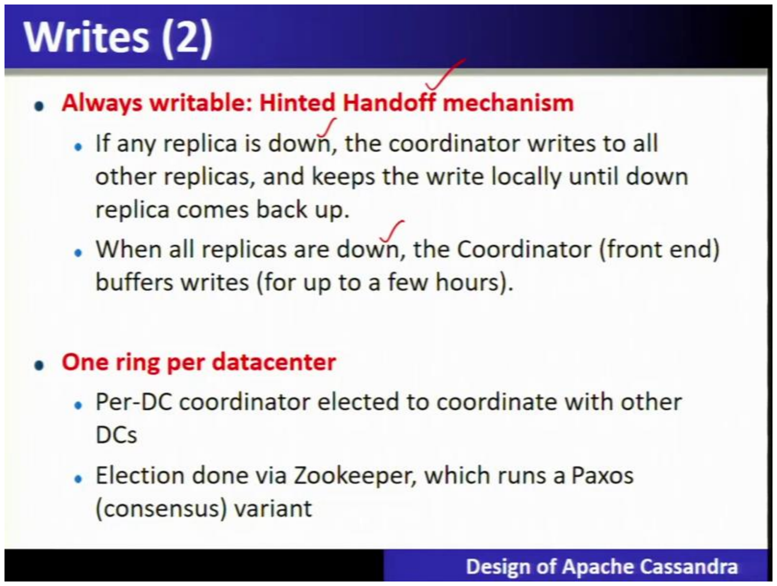
replica #2

Design of Apache Cassandra

So, let us see about the writes. So, writes need to be lock-free and fast, no reads or the disk seeks are needed. So, when a client sends the write to one of the coordinator node in Cassandra cluster, then the coordinator may be per-key, per-client or per-query basis, that we have shown you in the previous diagram. So, per-key coordinator ensures the writes for the keys are serialized and the coordinator uses partitioner to send the query to all the replica nodes responsible for that particular key.

So, when an X replica respond, the coordinator returns an acknowledgment to the client. So, what is this X? So, to make it more understandable is that you know that every data is replicated three times. Now, if the value of X is less than 3, let us say, 2, that means if the two replicas, if they respond to the coordinator, if the two replicas respond not, but not three, even then if the value of X is set as 2. So, this means the coordinator will return an acknowledgment to the client and rest of the third replica will be handled later point of time by the coordinator.

(Refer Slide Time: 36:40)



Writes (2)

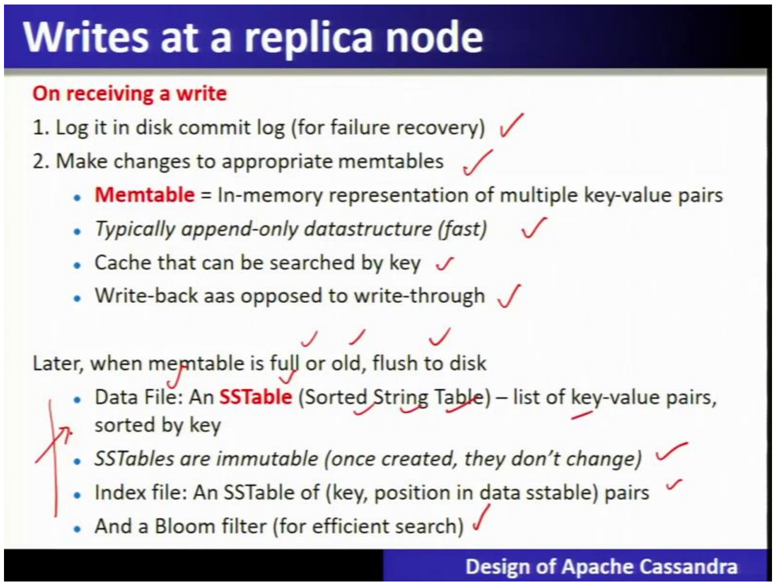
- **Always writable: Hinted Handoff mechanism**
  - If any replica is down, the coordinator writes to all other replicas, and keeps the write locally until down replica comes back up.
  - When all replicas are down, the Coordinator (front end) buffers writes (for up to a few hours).
- **One ring per datacenter**
  - Per-DC coordinator elected to coordinate with other DCs
  - Election done via Zookeeper, which runs a Paxos (consensus) variant

Design of Apache Cassandra

Now writes, always writable, that is Hinted Handoff mechanism is used for the write operations. So, if any replica is down, the coordinator writes to all the other replicas and keep the write locally until the replica comes up. So, when all the replicas are down, the coordinator, that is the front end buffers the write for the first few hours.

So, this is called Hinted Handoff mechanism. Now, then, we are going to see in the write one data, one ring per datacenter is followed as the resources for data storage. So, per datacenter coordinator is elected to coordinate with other datacenter and the election is done via the Zookeeper which runs Paxos, that is a consensus variant protocol.

(Refer Slide Time: 37:40)



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

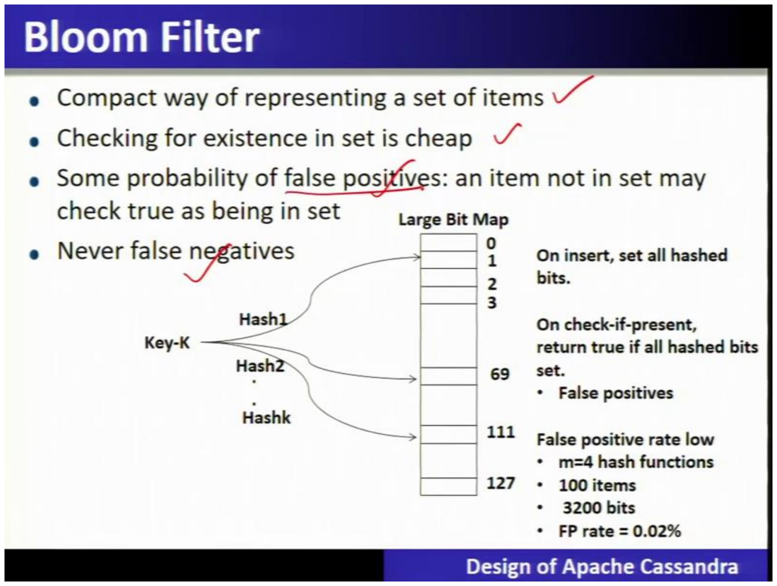
- Data File: An **SSTable** (Sorted String Table) – list of key-value pairs, sorted by key
- *SSTables are immutable (once created, they don't change)*
- Index file: An SSTable of (key, position in data sstable) pairs
- And a Bloom filter (for efficient search)

Design of Apache Cassandra

So, write at the replica node. On receiving the write, it locks it to the disk commit log for failure recovery, make the changes in the appropriate memory table called memtable. So, in-memory representation of multiple key value pairs typically, it is used the append-only datastructure, that is very fast to support the fast write operations. Similarly, the cache that can be searched by the key and write-back as opposed to the write-through. These are some of the important implementation details which Cassandra uses to make this very fast.

So, later the memtable is full or old, then it will be flushed to the disk. For that, as far as the data file is concerned, it is nothing but supported as the SSTable, which is called Sorted String Table, that is a list of key-value pairs which are sorted by the keys. SSTables are immutable. That is, once created, they do not change, and index file, that is an SSTable of the key and the positions in the SSTable pair. It also uses the Bloom filter for efficient searching. Why? Because you know that o many data files will be now stored, so many SSTables will be created.

(Refer Slide Time: 39:00)



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.

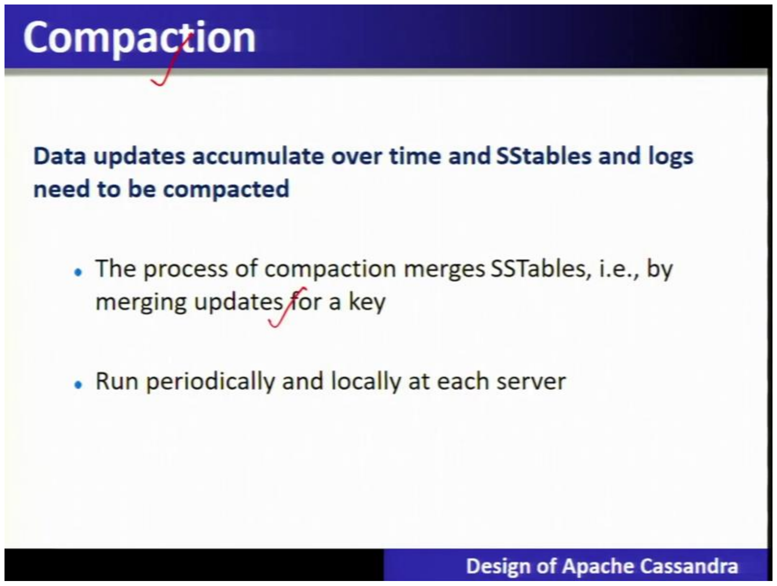
On check-if-present, return true if all hashed bits set.
- False positives

False positive rate low
- m=4 hash functions
- 100 items
- 3200 bits
- FP rate = 0.02%

Design of Apache Cassandra

So, we are skipping the Bloom filter, but nevertheless, Bloom filter will make it the retrieval of that particular key very fast through the SSTable. So, Bloom filter is a compact way of representing a set of items and checking for its existence in the set is quite cheap, and some probability of false positives are there. So, that is that the item not in the set maybe check true as being in the set, but it is supporting never the false negatives. So, false positives is doable in most of the applications but not the false negatives. So, Bloom filter is good to support the fast retrieval.

(Refer Slide Time: 39:45)



Compaction

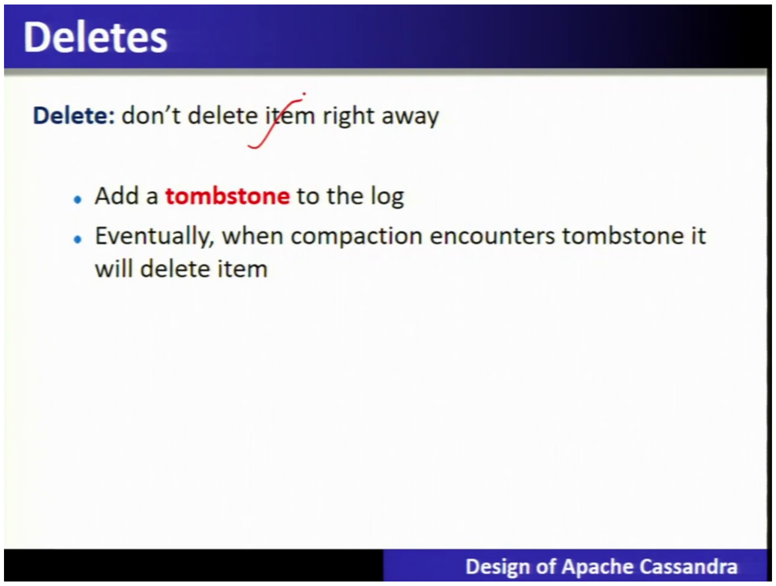
Data updates accumulate over time and SStables and logs need to be compacted

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

Design of Apache Cassandra

Now then, there is a operation which is called the compactation. So, data updates accumulate over the time and SSTables and logs need to be compacted. So, the process of compactation, merges the SSTable by merging the updates for a key. It runs locally and periodically at each server.

(Refer Slide Time: 40:10)



Deletes

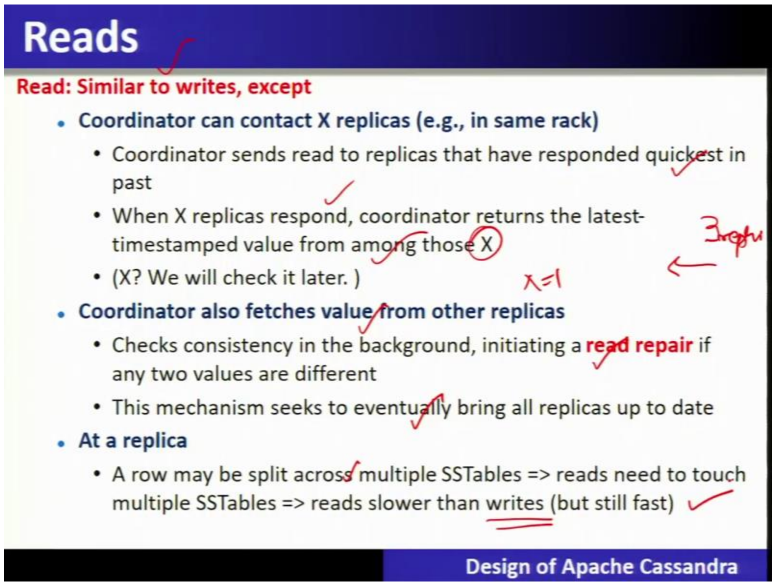
Delete: don't delete item right away

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

Design of Apache Cassandra

The deletes. So, deletes of an item is not done right away but it will be add to the tombstone, to the log and eventually when the compactation encounters tombstone, it will delete the item.

(Refer Slide Time: 40:20)



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)

$x=1$

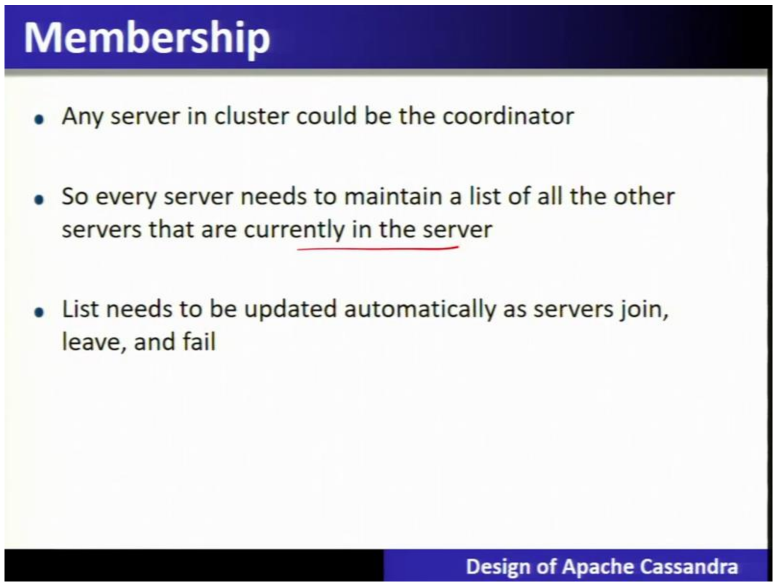
$\leftarrow$ 3 repli

Design of Apache Cassandra

Reads. Similar to the writes, but except the coordinator can contact X replicas, that is in the same rack. Coordinator sends the read to the replicas that have responded quickest in the past. So, when X replicas respond, the coordinator returns the latest timestamp value among those X. So, X is configurable. So, for example if you have three replicas and if let us say that the quickest one has responded and the value of X is equal to 1, then this particular information will be passed on to the client, and the read operation is completed.

Now, coordinator also fetches the values from other replicas. So, to check the Consistency in the background, it will initiate the read repair if any two values are quite different. So, this mechanism seeks to eventually bring all the replicas up to date. So, at a replica, the node may be split across multiple SSTables. That is quite possible. So, reads need to touch multiple SSTables and the reads may be slower therefore, than the writes, but is still quite fast in comparison to the RDBMS.

(Refer Slide Time: 41:41)



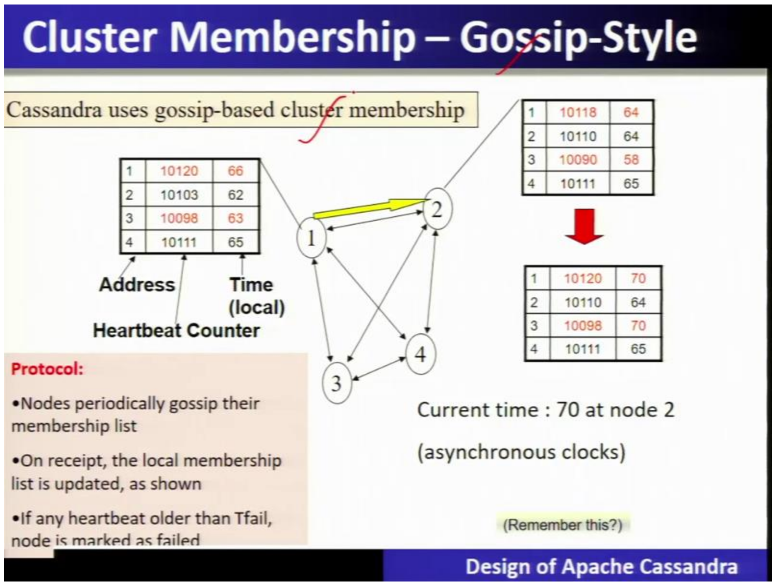
Membership

- 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

Design of Apache Cassandra

So, membership. So, any server in the cluster could be the coordinator and we have seen how the election of that coordinator happens. So, every server need to maintain a list of all other servers that are currently in the server and the list to be updated automatically as the server joins leaves or fails.

(Refer Slide Time: 42:00)



Cluster Membership – Gossip-Style

Cassandra uses gossip-based cluster membership

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

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

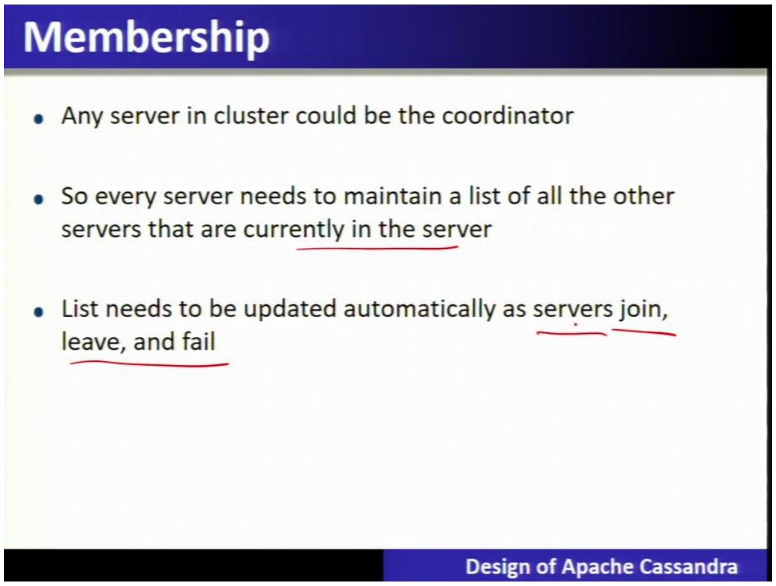
(asynchronous clocks)

(Remember this?)

Design of Apache Cassandra

So, this cluster membership is done through the protocol which is called a Gossip-Style protocol. So, Cassandra uses gossip-based cluster membership.

(Refer Slide Time: 42:10)



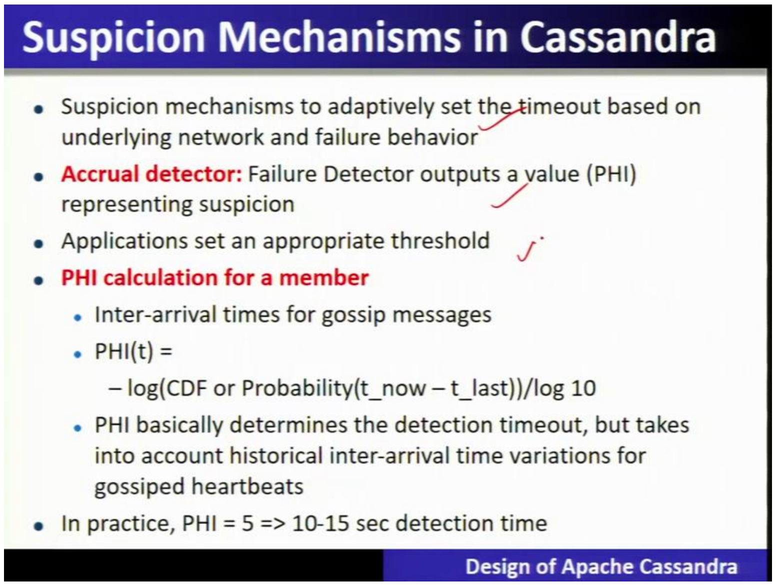
Membership

- 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

Design of Apache Cassandra

Cluster membership means that it has to keep a list of, or it has to update the operation such as when a server joins, leaves and fails in this particular ring, that is done by the cluster membership.

(Refer Slide Time: 42:25)



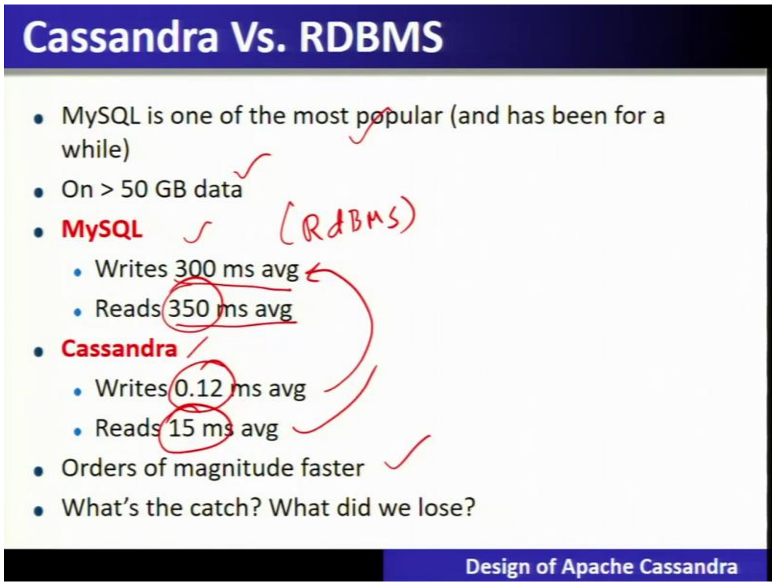
Suspicion Mechanisms in Cassandra

- 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

Design of Apache Cassandra

So, Suspicion Mechanism is also there in the Cassandra. Suspicion Mechanism to adaptively set the timeout based underlying network and the failure behavior. So, Accural detector, means failure detector outputs a value representing the suspicion and the application certain appropriate threshold for that. So, we are not going in detail about PHI calculation. So, that is used for the gossip protocol settings.

(Refer Slide Time: 42:55)



Cassandra Vs. RDBMS

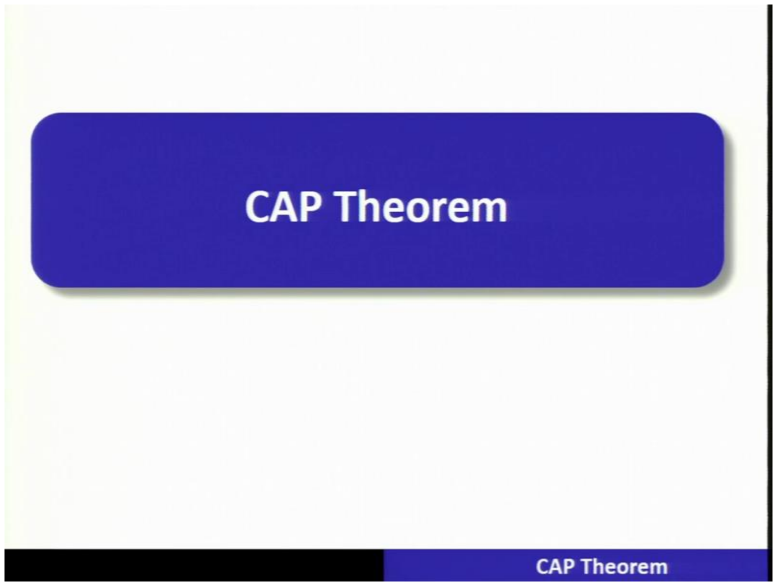
- MySQL is one of the most popular (and has been for a while)
- On > 50 GB data
- **MySQL** (RdBMS)
  - Writes 300 ms avg
  - Reads 350 ms avg
- **Cassandra**
  - Writes 0.12 ms avg
  - Reads 15 ms avg
- Orders of magnitude faster
- What's the catch? What did we lose?

Design of Apache Cassandra

So, therefore let us compare the Cassandra versus RDBMS. So, MySQL is one of the most popular at a while. So, if let us say that we want to compare the Cassandra versus RDBMS on data which is called 50 GB data. So, what you will, what we have found out that MySQL writes usually takes 300 millisecond on average. Reads in MySQL, that is in RDBMS takes 300 milliseconds, whereas in Cassandra, the writes will take only 0.12 milliseconds on an average.

So, write is much, much faster compared to the RDBMS. Similarly, reads is 15 milliseconds in Cassandra whereas your RDBMS supports 350 milliseconds, that is much, much at a lower time, that is very quick operation. So, magnitude of orders, of magnitude this Cassandra is faster. So, what is the cache and what do we lose out of this particular fast access?

(Refer Slide Time: 44:15)

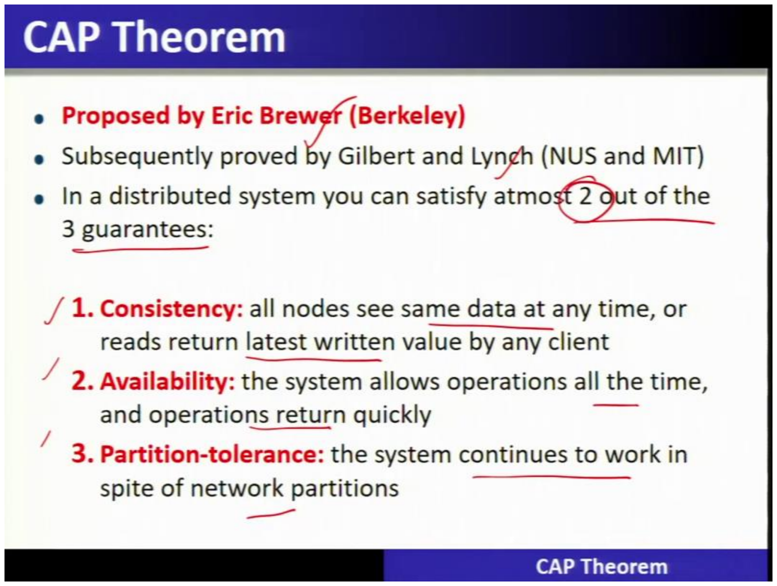


CAP Theorem

CAP Theorem

To understand this difference we have to understand the theorem which is called a CAP Theorem.

(Refer Slide Time: 44:20)



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:

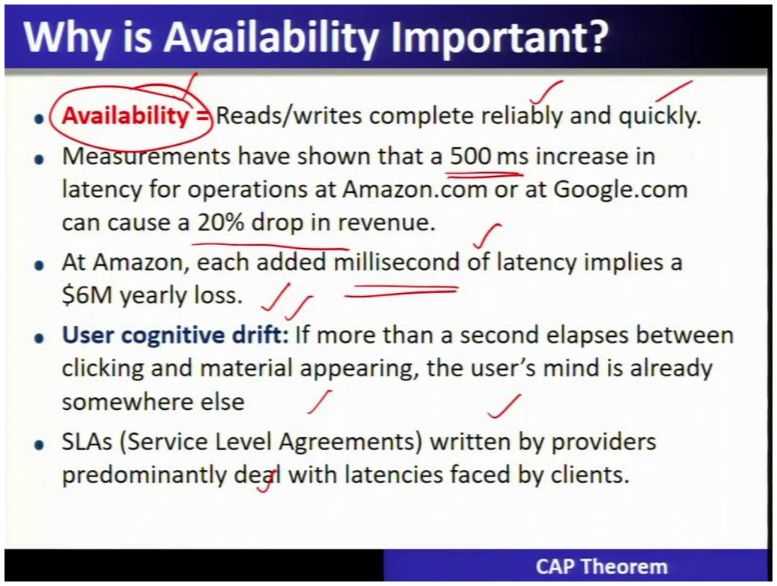
1. **Consistency:** all nodes see same data at any time, or reads return latest written value by any client
2. **Availability:** the system allows operations all the time, and operations return quickly
3. **Partition-tolerance:** the system continues to work in spite of network partitions

CAP Theorem

So, CAP Theorem is proposed by the Eric Brewer from Berkeley, subsequently which is proved by Gilbert and Lynch. So, in a distributed system you can satisfy two out most three guarantees. So, what are these three guarantees which only two of them can be supported according to the CAP theorem? So, the first one is called Consistency, that is, the node sees the same data at any time or the reads return the latest written value by any client. That is called Consistency.

Availability means the system allows the operation at all points of time and the operations return quickly, that is called Availability. Third one is called Partition-Tolerance, that is, the system continues to work in spite of uh network partitions. So, these three are important factors, but CAP theorem says that not three or not all three can be guaranteed, only two out of the three is guaranteed in this distributed system situation.

(Refer Slide Time: 45:22)



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.

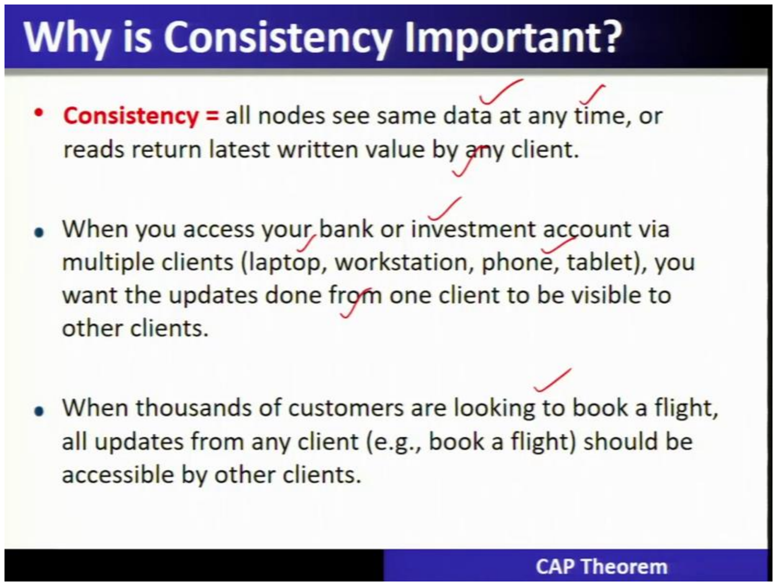
CAP Theorem

So, let us see which is more important for different applications. So, let us see one by one. Availability means that the reads and writes will complete reliably and quickly. So, measurements have shown that 500 milliseconds increase in the latency for the operations at Amazona.com or at Google.com can cause revenue 20 percent drop. So, why? Because at Amazon, each added milliseconds of the latency will imply a revenue loss. Why? Because the customers will churn and they will prefer some other website which is very fast responding.

So, Availability is most important that some of these e-commerce websites or let us say that for IoT data also if the analytics is so important that is it is monitoring let us say this logistics of this company called Amazon.com and let us say that this particular latency is added that is it is showing, so Availability has to ensure at all points of time. Similarly, the user cognitive drift. So, if more than a second elapse between the clicking the material appearing, and so users mind will change.

So, this is governed by thing which is called service level agreements. So, the providers, they have to predominantly deal with the latencies which are faced by. So, Availability again is an important factor in running this particular business or on this data store, using the data store. So, the functionality of the data stored is very important because all the data is to be fetched out of this data store. So, Availability is a key aspect.

(Refer Slide Time: 47:16)



Why is Consistency Important?

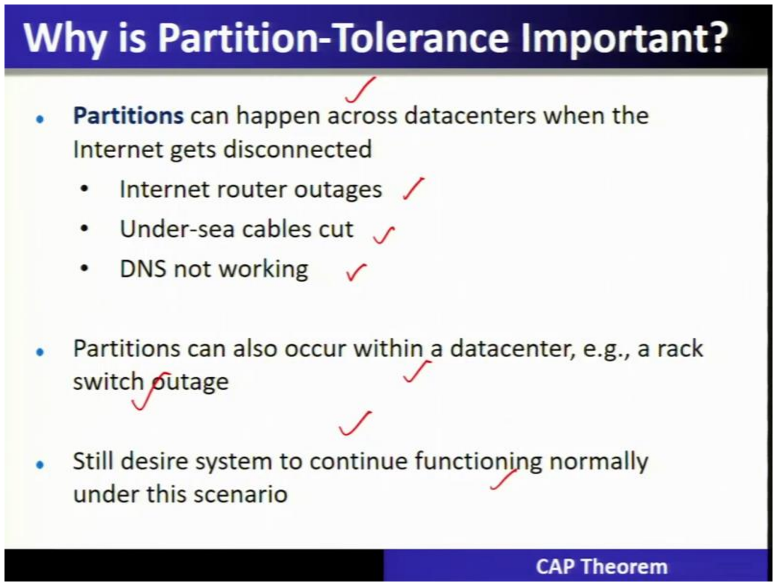
- **Consistency =** all nodes see same data at any time, or reads return latest written value by any client.
- When you access your bank or investment account via multiple clients (laptop, workstation, phone, tablet), you want the updates done from one client to be visible to other clients.
- When thousands of customers are looking to book a flight, all updates from any client (e.g., book a flight) should be accessible by other clients.

CAP Theorem

The second important aspect is called the Consistency, that is, all the nodes see the same data at any time or the reads return the latest return value by any client. So, this is very important as far as the banking applications are concerned. So, when you access your bank or the investment account by multiple clients, whether it is a laptop or you are accessing through a mobile phone, so you want these updates to be done from one client to be visible to the other clients.

So, whether, that means whether you are using the desktop or a laptop, all the updates has to show the same data at any point of time. So, when thousands of customers are looking to book a flight and all the updates from any client should be accessible by the other. So, this Consistency also is sometimes important but in the situations like banking applications, reservation situation and so on.

(Refer Slide Time: 48:01)



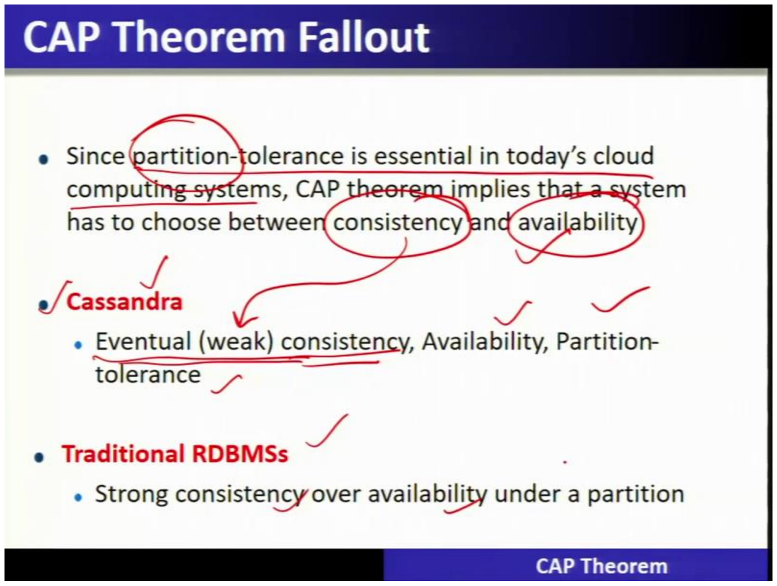
Why is Partition-Tolerance Important?

- **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

So, let us see the third aspect which is called a Partition-Tolerance. Partition can happen across datacenter when the internet gets disconnected. That is due to the internet router outages or under-sea cable cut or DNS is not working. So, these partitions can happen across the datacenter due to these kind of disruptions. So, partition can occur within a datacenter also. For example, when in the condition of a rack switch outage. You still needs a desirable system to continue functioning normally under any of these situations of the outages.

(Refer Slide Time: 48:50)



CAP Theorem Fallout

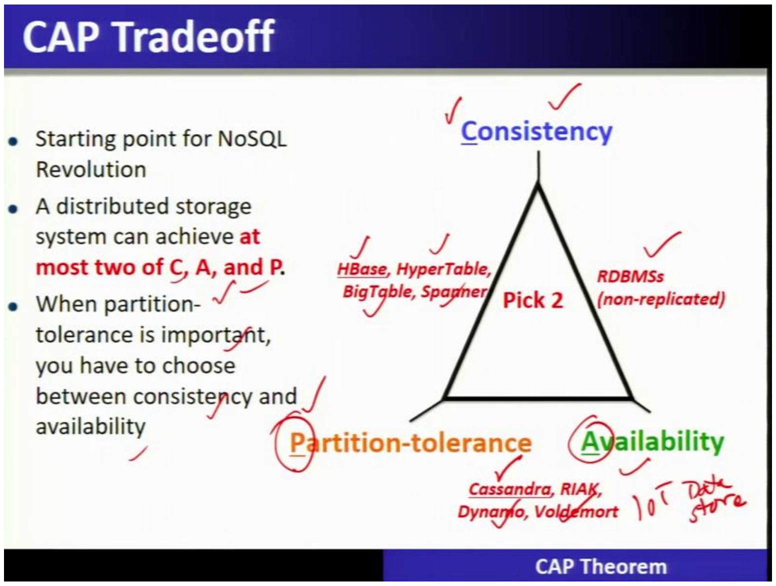
- Since partition-tolerance is essential in today's cloud computing systems, CAP theorem implies that a system has to choose between consistency and availability
- **Cassandra**
  - Eventual (weak) consistency, Availability, Partition-tolerance
- **Traditional RDBMSs**
  - Strong consistency over availability under a partition

CAP Theorem

So, the CAP Theorem Fallout says that the Partition-Tolerance is essential in today's cloud computing and all the services which is supported whether it is a IoT system and so on. So, the CAP theorem implies that the system has to choose between the Consistency and Availability. So, once you fix that Partition-Tolerance is an indispensable characteristics, so therefore now you have to choose between the Consistency and Availability.

Now, let us see the Cassandra. Cassandra chooses Availability over the Consistency and Consistency, it does a compromise, and it is called eventual Consistency. So, therefore Cassandra says that it compromises with the Consistency and the Consistency which Cassandra provides is called eventual or the weak Consistency whereas Availability and Partition-Tolerance is provided. So, traditional RDBMS has the strong Consistency over Availability under the under a partition.

(Refer Slide Time: 49:52)



CAP Tradeoff

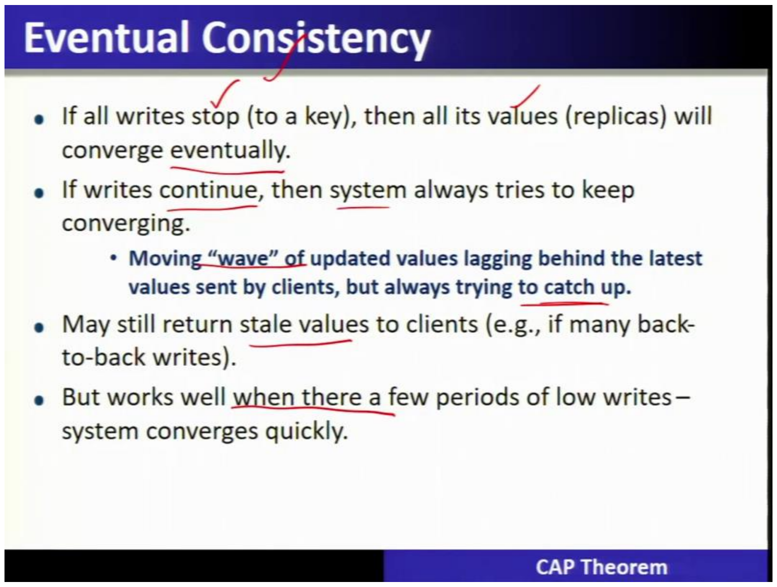
- Starting point for NoSQL Revolution
- A distributed storage system can achieve **at most two of C, A, and P.**
- When partition-tolerance is important, you have to choose between consistency and availability

So, let us see the CAP Theorem Tradeoff. So, Consistency, Partition-Tolerance and Availability is shown by this particular triangle. So, let us see any two, you can choose out of 3. So, let us see about the Cassandra. So, Cassandra chooses Partition-Tolerance and Availability, which is shown over here. Similar data base or NoSQL data store are RIAK, Dynamo, Voldemort. So, these are all systems which are used in IoT data store.

Now, regarding Consistency. So, if the Consistency is important, the banking application. So, if let us say Consistency is needed and Partition-Tolerance is a must, then it has to now compromise with the Availability. So, therefore you might have seen in the banking online application that the server is not available. But that means whenever it is available, it has to ensure the Consistency and Partition-Tolerance. So, the example of a databases which stores or which satisfies Consistency and Partition-Tolerance, they are HBase, then HyperTable, BigTable and Spanner. So, most of these are from Google.

If Consistency and Availability, both are to be ensured, then it is RDBMS. So, therefore starting point for NoSQL revolution is that it is a distributed storage system and can achieve at most two out of three, that is called a CAP theorem. So, when Partition-Tolerance is important, you have to choose between the Consistency and Availability. So, we have understood very details about the CAP theorem tradeoff.

(Refer Slide Time: 51:40)



Eventual Consistency

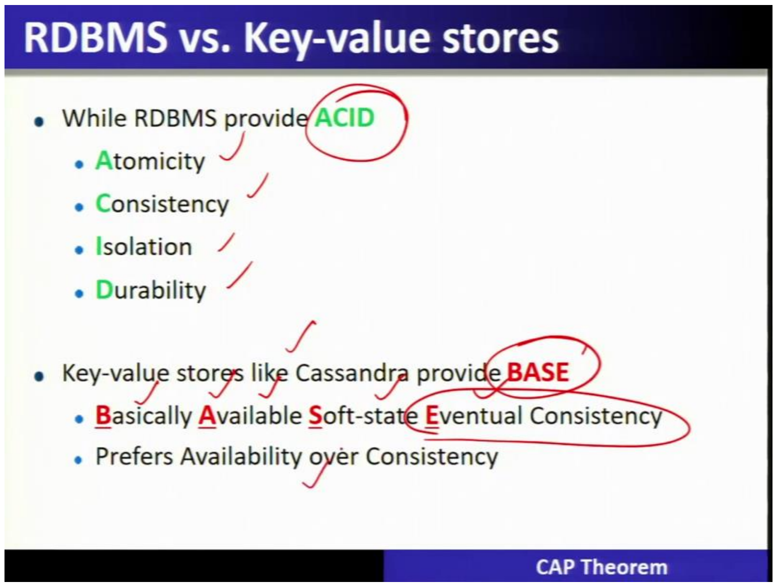
- If all writes stop (to a key), then all its values (replicas) will converge eventually.
- If writes continue, then system always tries to keep converging.
  - Moving "wave" of updated values lagging behind the latest values sent by clients, but always trying to catch up.
- May still return stale values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

CAP Theorem

So, let us see how the eventual Consistency is supported in the Cassandra. So, eventual Consistency that is if all the writes stop to a key all the writes to a key, then all its values, that is replica, has to converge eventually. This is means that eventual Consistency. Now, if the writes continue then the system tries to keep converging. So, there will be a wave of update values which is lagging behind the latest value sent by the client, and eventually they are trying to catch up.

So, eventually it will achieve the Consistency but not instantly. So, may sometimes written the stale values to the client if many back-to-back writes are there, but it will work fine for this kind of situation, let us say IoT and all when there are few periods of the low writes and system converges quite quickly.

(Refer Slide Time: 52:30)



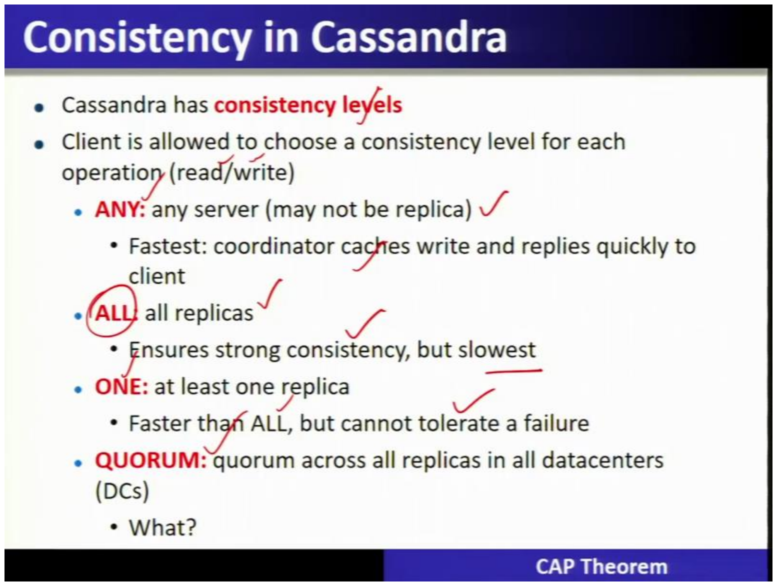
RDBMS vs. Key-value stores

- While RDBMS provide ACID
  - Atomicity
  - Consistency
  - Isolation
  - Durability
- Key-value stores like Cassandra provide BASE
  - Basically Available Soft-state Eventual Consistency
  - Prefers Availability over Consistency

CAP Theorem

So, if you see RDBMS versus key-value store, so RDBMS supports the ACID properties for the transaction, that is Atomicity, Consistency, Isolation and Durability. Now, in contrast, if you see the Cassandra or a key-value store like Cassandra, it uses for the transaction, the property which is called a BASE property. So, BASE stands for Basically Available Soft-state Eventual Consistency. Means that it ensures the Availability, it ensures the Partition-Tolerance, but the Consistency is eventual. So, BASE means Basically Available Soft-state Eventual Consistency, which prefers Availability over the Consistency.

(Refer Slide Time: 53:10)



Consistency in Cassandra

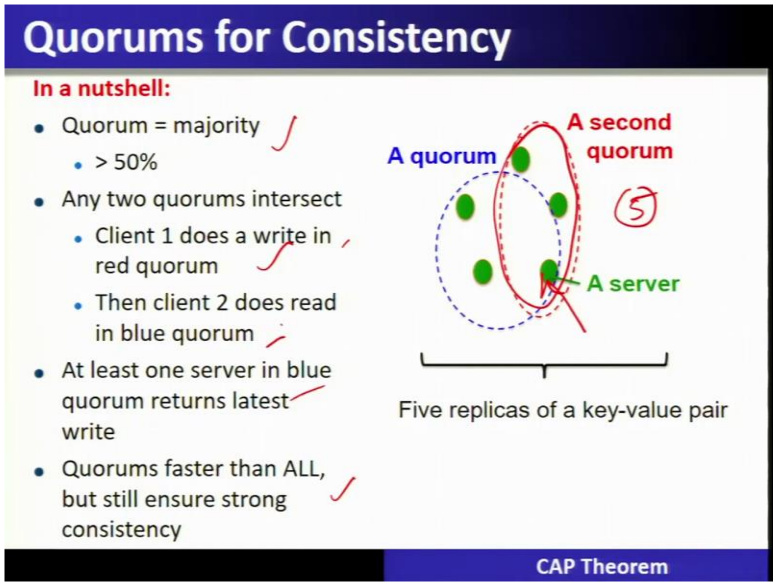
- Cassandra has **consistency levels**
- Client is allowed to choose a consistency level for each operation (read/write)
  - **ANY:** any server (may not be replica) ✓
    - Fastest: coordinator caches write and replies quickly to client
  - **ALL:** all replicas ✓
    - Ensures strong consistency, but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

CAP Theorem

So, Consistency in Cassandra. If you see, Cassandra has the consistency levels. So, clients allowed to choose the consistency level for each operation that is for the read and write. So, consistency level, the client can choose, that is configurable. If it is ANY, that means any server may not be, may not be the replica, that is the fastest one, if the coordinator caches the write and quickly replies to the client.

So, ALL means all the, for all the replicas, the update has to be there. So, that means it is the condition of Consistency, ensuring strong consistency, but it will be quite slow. So, ONE means that at least one replica, it is faster than all, but cannot tolerate a failure. So, there is a QUORUM. So, QUORUM across all the replicas in all the datacenters.

(Refer Slide Time: 54:06)



Quorums for Consistency

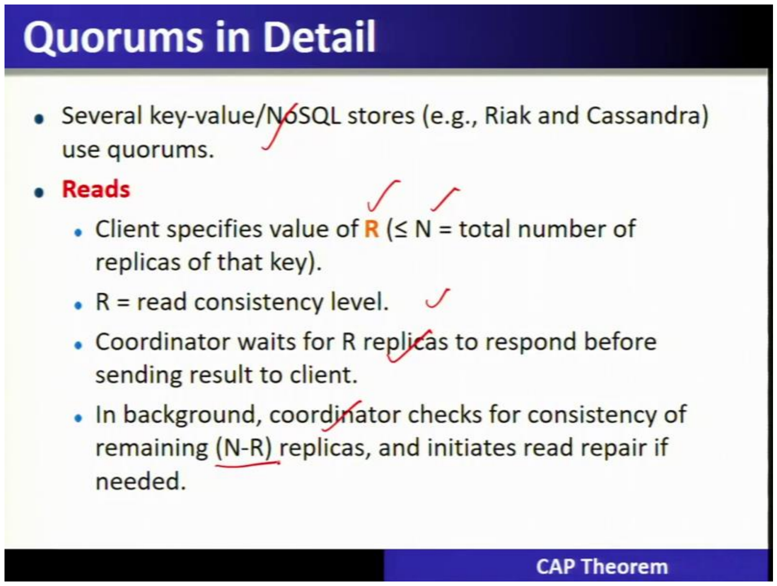
In a nutshell:

- 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

A quorum
A second quorum
5
A server
Five replicas of a key-value pair
CAP Theorem

So, in a nutshell, so quorum means that a majority which is more than 50 percent. So, any two quorums may intersect, you can see over here. So, this example shows the five different servers. So, the majority among five is at a time, three. So, the Client 1 does the write in the red quorum. So, this is the red quorum. Then Client 2 does the read in the blue quorum. So, at least one server in the blue quorum returns the write. So, the quorum are faster than all but it still ensures the strong consistency. Five replicas of a key-value store is shown in this example.

(Refer Slide Time: 54:48)



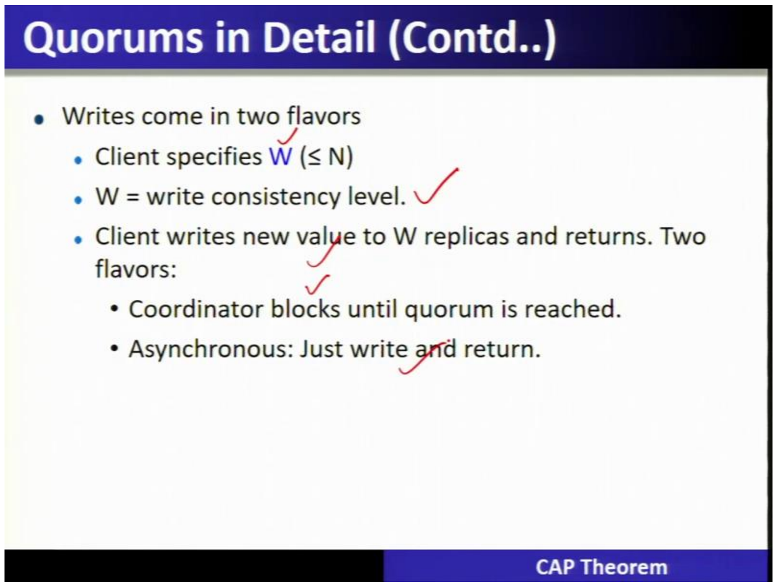
Quorums in Detail

- Several key-value/NoSQL stores (e.g., Riak and Cassandra) use quorums.
- **Reads**
  - Client specifies value of R (≤ 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.

CAP Theorem

So, quorums, let us go in more detail. So, several key-values stores like NoSQL, Riak and Cassandra use the quorums for the read operation. So, client specifies the value of R here in this case which is less than or equal to N, that is the number of replicas of that key and R is the consistency level. So, the coordinator waits for R replicas to respond before sending the result to the client. In the background, the coordinator checks for the consistency of the remaining N minus R replicas and initiate the read repair for eventual consistency.

(Refer Slide Time: 55:21)



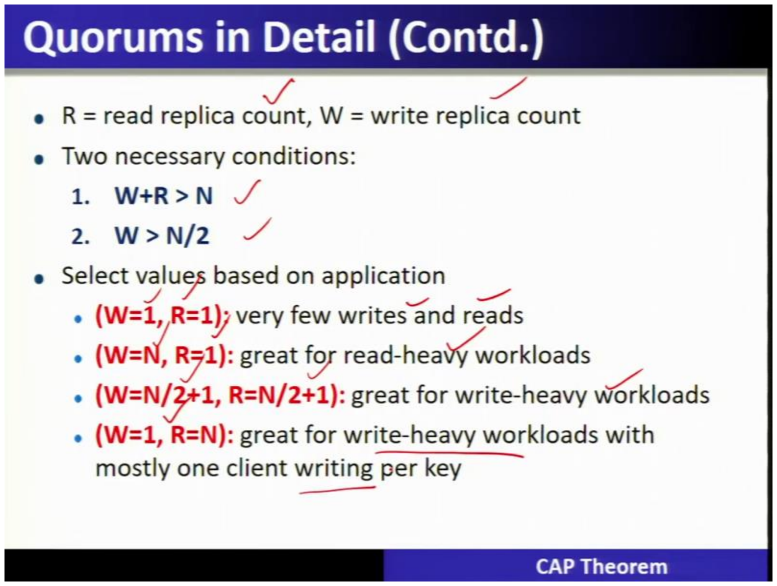
Quorums in Detail (Contd..)

- 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.

CAP Theorem

So, as far as the writes are concerned, writes comes in two flavors. So, client has to specify the value of W, which is less than or equal to N. So, at W means the write consistency level. So, client writes a new value to W and to W replicas and returns in the write. So, two flavors are there. So, coordinator blocks until the quorum is reached. Asynchronous means that it just writes and returns. So, it depends upon these kind of settings.

(Refer Slide Time: 55:48)



Quorums in Detail (Contd.)

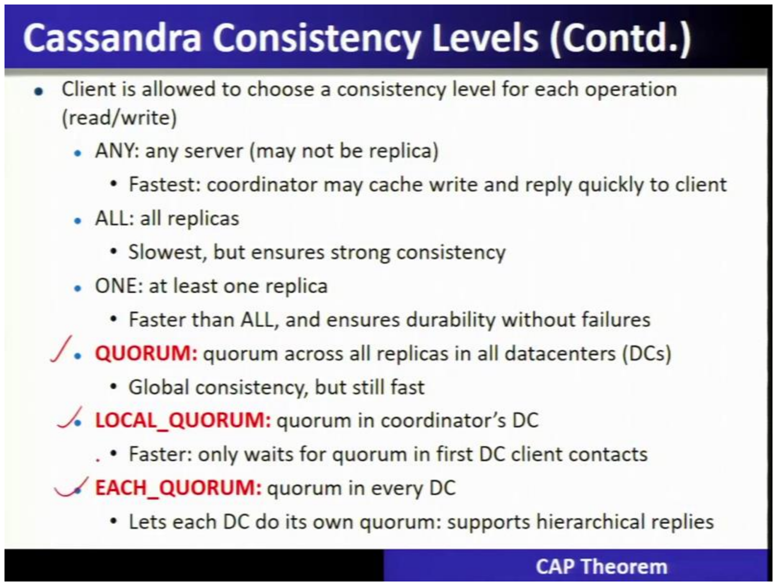
- R = read replica count, W = write replica count
- Two necessary conditions:
  1. **W+R > N**
  2. **W > N/2**
- Select values based on application
  - **(W=1, R=1):** very few writes and reads
  - **(W=N, R=1):** great for read-heavy workloads
  - **(W=N/2+1, R=N/2+1):** great for write-heavy workloads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

CAP Theorem

So, R is the read replica count, W is the write replica count. So, two necessary conditions has to ensure that read plus write should be greater than N and W is more than N by 2. So, we have to select these values based on these applications when W is 1 and R is equal to 1, then very few writes and reads are there. So, and when W is N and R is 1 then it is great for read-heavy workloads.

When W is N by 2 plus 1 and R is also N by 2 plus 1, it is great for right-heavy workloads. And when W is 1 and R is equal to N, it is great for write-heavy workloads with mostly one client writing per key. So, these are all very much design issues which are needed to support the IoT applications.

(Refer Slide Time: 56:40)



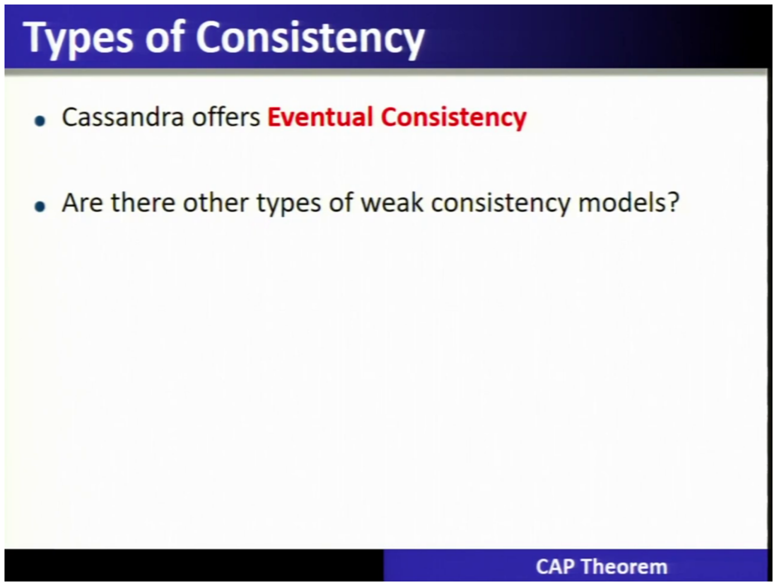
Cassandra Consistency Levels (Contd.)

- 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

CAP Theorem

So, Cassandra Consistency Level. So, Cassandra is allowed to choose these consistency levels which we have already, some of them we have already stated about the QUORUM and the LOCAL_QUORUM quorum and EACH_QUORUM.

(Refer Slide Time: 56:53)



Types of Consistency

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

CAP Theorem

So, the types of Consistency which Cassandra offers is called Eventual Consistency. Now, you may ask are there any other type of weak consistency models?

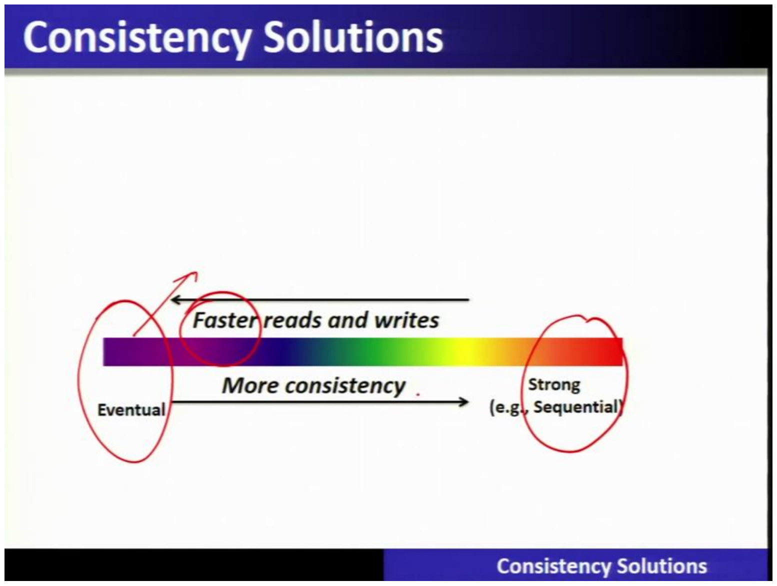
(Refer Slide Time: 57:03)

Consistency Solutions

Consistency Solutions

So, let us go ahead with the Consistency models which are supported across different NoSQL data stores including the IoT data bases.

(Refer Slide Time: 57:06)

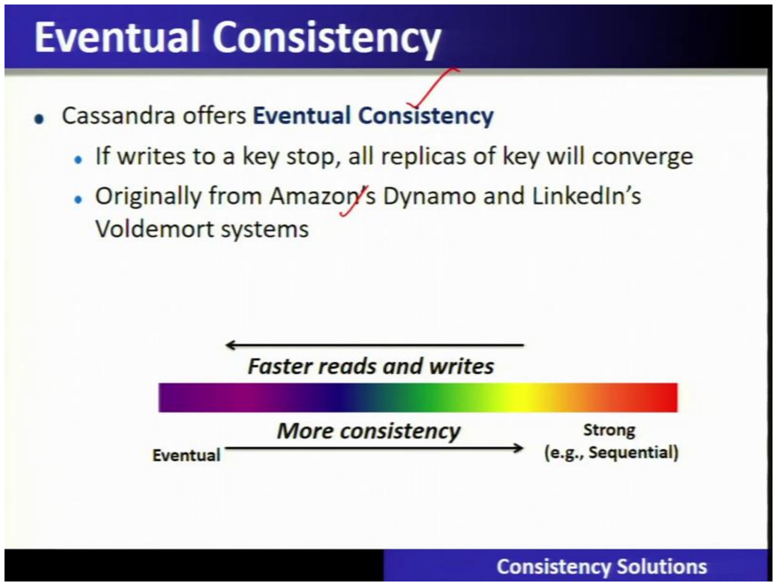


Consistency Solutions

Consistency Solutions

So, Consistency level, on one side is a strong consistency, the other side is called eventual consistency. Sometimes, if it is closer to eventual consistency, then the reads and writes are quite, then the faster reads and writes, it will support. If it is towards strong consistency, then reads and writes are slower but it will support the strong consistency. So, it is a spectrum of different consistency models between eventual and strong.

(Refer Slide Time: 57:40)



Eventual Consistency

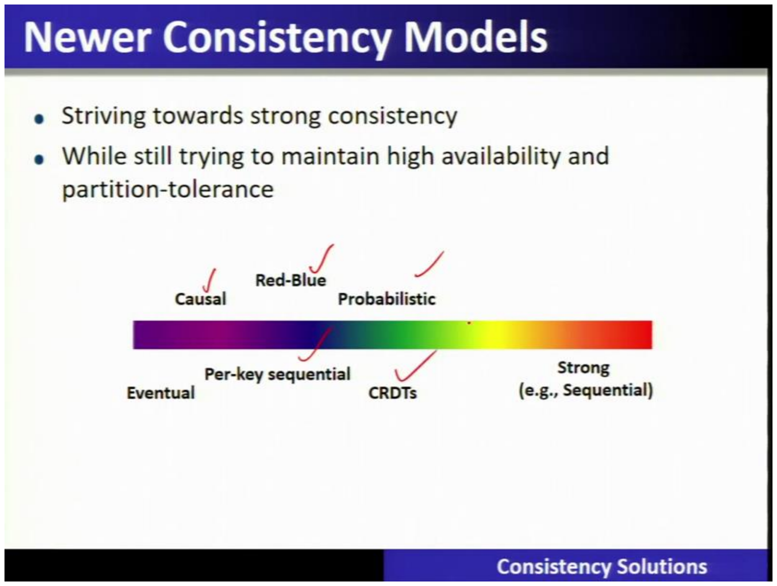
- Cassandra offers **Eventual Consistency**
  - If writes to a key stop, all replicas of key will converge
  - Originally from Amazon's Dynamo and LinkedIn's Voldemort systems

Faster reads and writes
More consistency
Eventual
Strong
(e.g., Sequential)

Consistency Solutions

So, Cassandra offers eventual consistency. So, if the write to a key stops, all replicas will converge. Originally from Amazon's Dynamo and LinkedIn's Voldemort systems supports that.

(Refer Slide Time: 57:54)



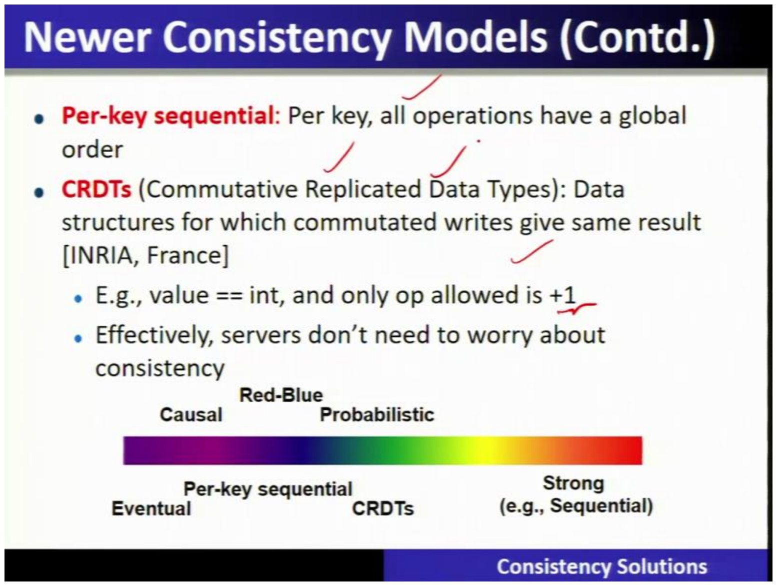
Newer Consistency Models

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

Consistency Solutions

Now, let us see what are the other consistency model which are now available or evolving. So, very close to eventual is called Causal Consistency, then Red-Blue, Probabilistic, Per-Key, CRDT, there are many other models available. So, there are many more databases available in the, by different providers. It depends upon the application.

(Refer Slide Time: 58:18)



Newer Consistency Models (Contd.)

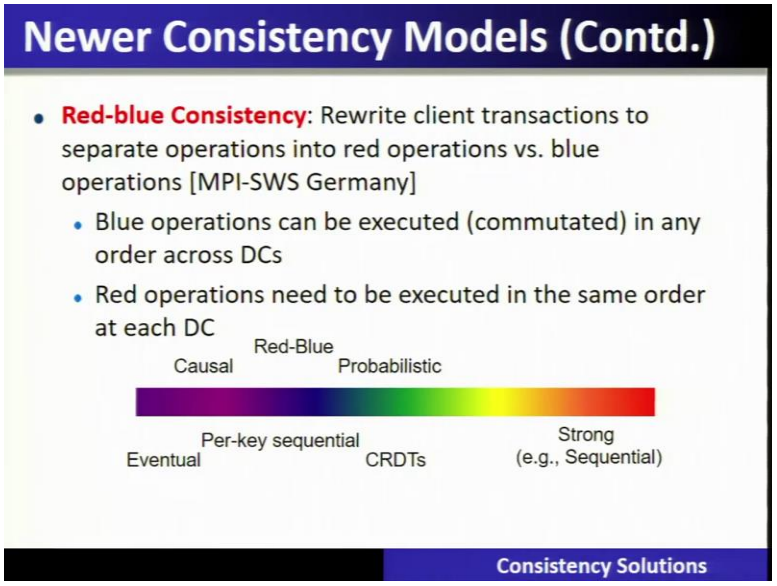
- **Per-key sequential:** Per key, all operations have a global order
- **CRDTs** (Commutative Replicated Data Types): Data structures for which commutated writes give same result [INRIA, France]
  - E.g., value == int, and only op allowed is +1
  - Effectively, servers don't need to worry about consistency

Causal
Red-Blue
Probabilistic
Eventual
Per-key sequential
CRDTs
Strong (e.g., Sequential)

Consistency Solutions

So, Per-Key sequential is all the operations have a global order. CRDTs means Commutative Replicated Data Types. So, data structures for which the commutated writes give the same results. So, for example the integer values, if it is, then the operations plus 1 is allowed whether A plus B or B plus A. So, commuted operations, so, commutative operations are very much needed to support this type of consistency. So, eventually, effectively servers do not need to worry about these consistencies.

(Refer Slide Time: 58:56)



Newer Consistency Models (Contd.)

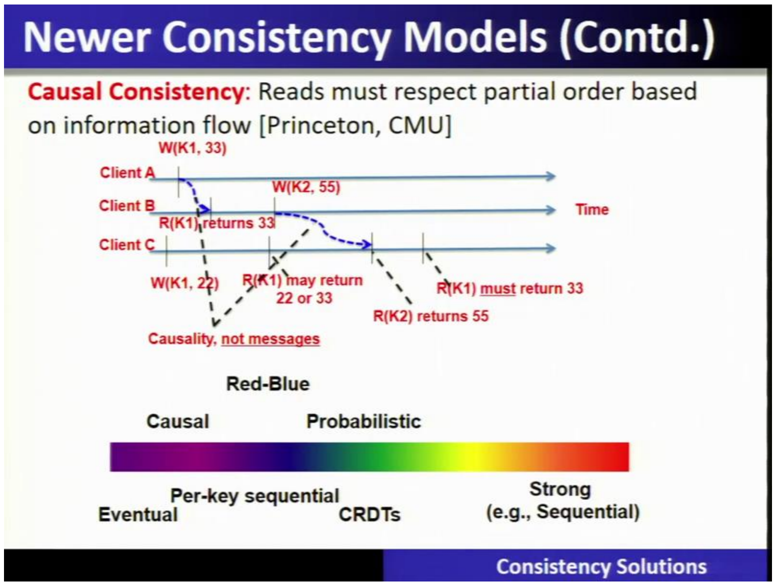
- **Red-blue Consistency**: Rewrite client transactions to separate operations into red operations vs. blue operations [MPI-SWS Germany]
  - Blue operations can be executed (commutated) in any order across DCs
  - Red operations need to be executed in the same order at each DC

Causal
Red-Blue
Probabilistic
Eventual
Per-key sequential
CRDTs
Strong (e.g., Sequential)

Consistency Solutions

Red-Blue Consistency, that it will rewrite the client's transaction to a separate operations in a red operations versus the blue operations. So, blue means it can be executed in any order in any datacenter. Red operations need to be executed in some order.

(Refer Slide Time: 59:11)



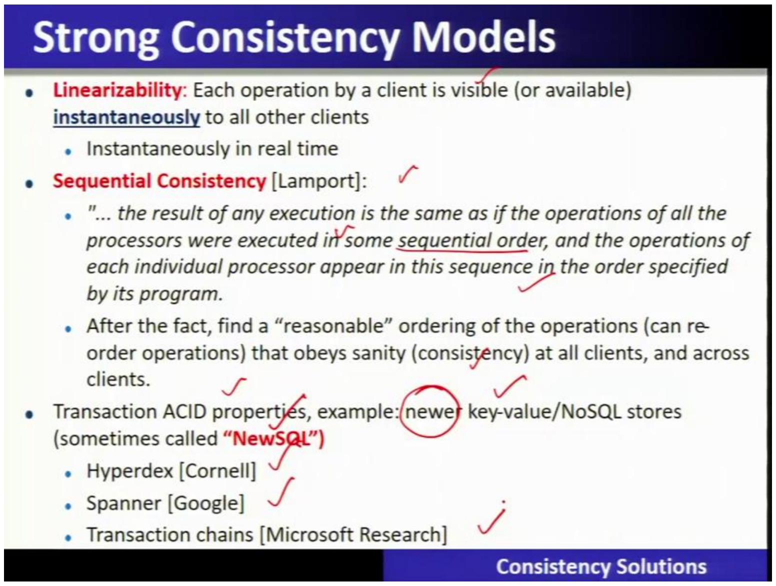
Newer Consistency Models (Contd.)

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

Consistency Solutions

So, these are all, this is called Causal Consistency, that is the reads respect the partial order based on the information flow. And this is developed by Princeton, CMU. It is shown over here.

(Refer Slide Time: 59:25)



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]

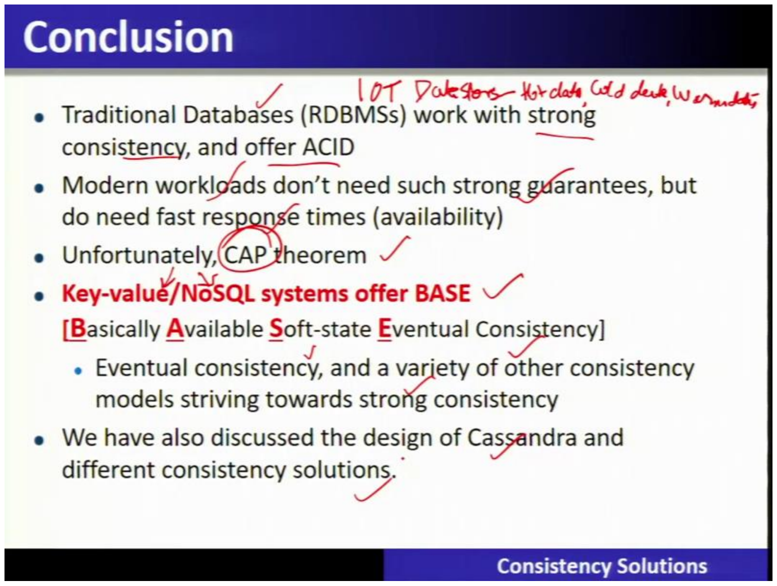
Consistency Solutions

So, Strong Consistency Model supports the options of property which is called Linearizability. So, each operation by the client is visible or available instantaneously to all other clients. So, this is called Linearizability. So, instantaneously is in the real time, so without any lag. So, Sequential Consistency, which is supported by the Lamport is the result of any execution is the same as if the operations are all the processors are executed in some sequential order and the operations of each individual processor appears in this sequence in the order specified by the program.

So here, the, to find out the value, the reasonable ordering of the operations can be, can reorder operations that obey consistency at all the clients, across the clients. So, the transaction, that is the ACID properties if let us say that you want to follow for NoSQL databases or then a new kind of development which is happening around the cloud providers. So, they are calling it as NewSQL.

So, this is, NewSQL means that it is the NoSQL data store and also it supports the ACID properties. So, the examples are Hyperdex by Cornell, and Spanner by Google and Transaction chains by the Microsoft Research. So, these are the newer developments called NewSQL. So, we have so far seen the NoSQL and newer SQL is in line.

(Refer Slide Time: 1:01:14)



Conclusion

IOT Datastores – Hot data, Cold data, Warm data

- 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.

Consistency Solutions

So, let us conclude this is that. So, we have started with the motivation of providing the storage technologies for IoT workloads. So, what we have found out is that this hot data analytics, cold data analytics, these are some of these and warm data analytics, the three type of analytics is required to be supported, and for that storage system technologies are required in place. So, for IoT, let me write down, IoT data stores requires hot data analytics, cold data analytics and warm data analytics.

So, what we have so far seen to support that, we have seen here the design of IoT databases and in that line, sometimes traditional databases or RDBMS can also be used which is having the strong consistency property and offer the ACID properties. However, the, today's workload which is so many number of sensors are sending the data to the cloud requires a time series data to be stored somewhere. Do not require these strong guarantees but do but do need fast response time, that is, the Availability.

Unfortunately, the CAP Theorem says that these all three, that is Consistency, Availability and Partition-Tolerance is not possible for all three to be abide by. Therefore, we have seen a design of a system that is the key-value in store that is called NoSQL system by the Cassandra, which offers BASE property, Basically Available Soft-state Eventual Consistency.

And this kind of system which is widely used in many of the production clusters, and also is, can be used here for IoT data store. So, it supports the eventual Consistency and a variety of other Consistency models can also be tunable here in Cassandra, that we have seen. We have also discussed the design of Cassandra and other different consistency solutions which is applicable for IoT data store. Thank you.