Beyond MapReduce and Spark: CAP, HBase, and Hive

Mirek Riedewald

Key Learning Goals

- What do the letter C, A, P in the CAP conjecture/theorem stand for?
- What do we mean when we say that one can only achieve two out of the three (C, A, P)?
  - Which does a traditional relational DBMS achieve?
  - Which does a distributed DBMs achieve?
  - Which do MapReduce and Spark achieve? Don’t they get them all?

Objective

- Understand the inherent limitations of processing data in a distributed environment.
- See more of the big picture of big data processing. In particular, understand the tradeoffs between MapReduce, Spark, relational databases, and NoSQL databases.
- Learn about database-style technology available for the MapReduce and Spark ecosystems.

Rationale

- The challenges associated with Big Data have resulted in a flurry of new data management and analysis technology and products. Marketing departments are doing their best to (over-) sell their products' capabilities. In the end, there is no magic bullet, just different tradeoffs. While products and buzzwords can change rapidly, the underlying principles and limitations usually remain a constant. Understanding more about these principles will prepare you better for future data analysis challenges, even after MapReduce, Spark, and other current approaches might not be in fashion any more.
- Understanding the big picture will allow you to make more informed decisions to pick the best data analysis technology or tool for a given task.

The Rise of NoSQL Databases

- Relational databases are highly successful tools for big data analysis. However, a significant number of users considers them limited in terms of performance and scalability. The most common problems include:
  - Relational databases are general-purpose data analysis tools. Text, graphs, and arrays can be stored in tables; and SQL can express many of the desired computations. However, data analysis tools specifically designed for such data can achieve better performance through specialized solutions.
    - Consider a Web search engine such as Google. Even though a database system could support text search and inverted indexes, the massive scale and simple "query" structure motivated Web search companies to custom-build their own highly specialized data management solutions.
  - The requirement to guarantee data consistency (ACID properties) places a great performance burden on a database.
    - Locking, needed for consistency and isolation, adds overhead to each operation, even if there is no conflict. Updates have to be logged, requiring expensive writes to stable storage, typically hard disks. And all processes participating in a distributed transaction must agree on the final outcome.
  - Data needs to be imported into the DBMS before any query can be executed. This includes careful schema design (relations and their attributes, indexes) and data cleaning. Even though this initial effort pays off later in query performance, many users dislike heavy startup cost and prefer to get "something" running quickly, dealing with data quality and structure issues later as needed. That approach is particularly appealing when the user needs a quick estimate to determine if it is worth exploring the data in more depth.

The Rise of NoSQL Databases (Cont.)

- NoSQL databases promise to address these shortcomings. The catchy name refers to a wide variety of approaches that present themselves as alternatives to relational technology. As NoSQL systems come and go, the list of examples below may change, but still illustrates typical approaches:
  - Google BigTable and Apache HBase focus on extreme scalability to hundreds or thousands of machines in a shared-nothing environment. To achieve this, they limit query functionality and do not support relational-style transactions. In essence, these systems are persistent key-value stores, allowing fast parallel lookup of data by key. Amazon's Dynamo service falls into the same category, allowing a weakening of data consistency guarantees in order to achieve greater scalability and data availability.
  - MongoDB is specially designed for text and document management. It achieves scalability by using a weaker consistency model that does not ensure ACID.
  - The Neo4j database is optimized for graph processing, supporting specialized data structures and query constructs. For example, it is easy to express path searches , something that requires non-trivial join expressions in SQL. Neo4j supports transactions.

The Rise of NoSQL Databases (Cont.)

- The following examples illustrate major tradeoffs between relational and NoSQL systems:
  - Relational databases emphasize data consistency and durability, resulting in comparably limited scalability and performance. Database programmers can actually choose weaker consistency levels for a transaction, improving performance at the cost of weaker consistency guarantees. However, relational databases inherently are designed to be general-purpose data management tools that put data consistency first.
  - Some NoSQL systems such as HBase, Dynamo, and MongoDB sacrifice functionality or consistency guarantees to achieve greater performance and scalability. Interestingly, as dealing with weaker consistency notions puts a burden on users, some NoSQL databases are adding stronger consistency features.
  - By specializing a database for a certain type of data or query, high performance can be achieved without sacrificing consistency.
- **This leaves the question if one could have it all**: scalability to many machines and high performance as in key-value stores, but also strong data consistency guarantees and non-trivial query functionality like in a relational DBMS. Could one design such a perfect system? It turns out that there are inherent limits, set by the tradeoffs between data consistency, availability, and the distributed nature of scalable data processing. We will discuss these results next.

The CAP Theorem

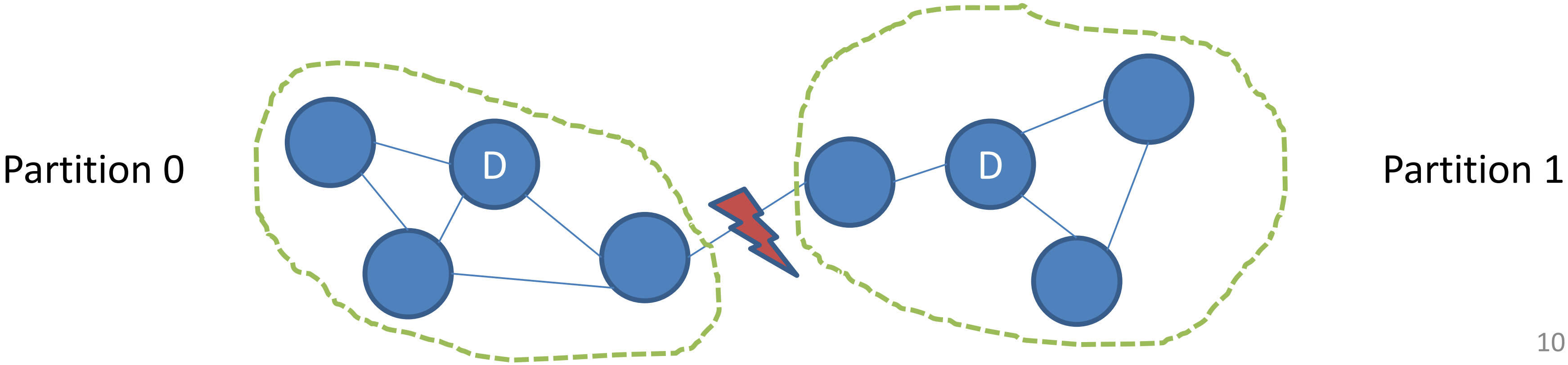
- At the 2000 ACM Symposium on Principles of Distributed Computing (PODC), Eric Brewer proposed the now famous CAP conjecture for networked shared-data systems. The CAP conjecture states that there is an inherent tradeoff between consistency, availability (for data updates), and tolerance to network partitions. A version of CAP was proved in 2002 as a theorem by Nancy Lynch and Seth Gilbert.
  - For more details about the CAP theorem, read Eric Brewer’s 2012 article [Brewer, E., "CAP twelve years later: How the "rules" have changed," IEEE Computer, vol.45, no.2, pp.23-29, Feb. 2012]
  - For a more technical view, read Lynch and Gilbert’s paper [Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News 33, 2 (June 2002), 51-59]

The CAP Theorem (Cont.)

- At the core of the CAP theorem lies the observation that one needs a consensus protocol to maintain consistent state across machines. Due to the need for consensus, it is impossible to have perfect data availability and consistency in the presence of network partitions. In particular:
  - *C+A case*: A system can achieve consistency and availability if there are no network partitions. This is the tradeoff selected by traditional single-node database servers.
  - *C+P case*: If network partitions are possible, then consistency can be achieved as long as only one partition accepts updates for an object. Since updates to this object are not possible in other partitions, availability is limited. A distributed relational database typical chooses this tradeoff.
  - *A+P case*: If network partitions are possible and the system still allows updates in all partitions (to achieve high availability), then copies of an object in different partitions might become inconsistent. Highly available and scalable systems such as Amazon's Dynamo select this tradeoff. To deal with data inconsistency, they rely on protocols that attempt to consolidate the different copies of the object after communication is restored.

Understanding the CAP Theorem

- Consider a distributed system with two nodes holding a copy of data item D, e.g., the bank account info of a customer.
- Assume a network failure separates the two copies. In practice this could also be caused by one of the nodes not responding to a request in a timely manner.
- If at least one copy of D allows updates, consistency is weakened: the copies are not identical to each other any more.
- To preserve consistency, availability has to be weakened. There are two basic ways to achieve this:
  1. During a network partition, both nodes make D available for reading only. This limits data availability for updates in both partitions.
  2. If one node is allowed to update D, the other cannot make it available for reading nor updating. In this case D is completely unavailable in that other network partition.



Dealing With Partitions

- In practice partitions are rare, hence consistency and availability are usually achievable. For instance, even in a wide-area network such as the Internet, there are usually multiple alternative routes between different nodes.
- Still, sometimes nodes that need to achieve consensus cannot communicate with each other. The system has to provide a mechanism for dealing with these *partitioning events*. In Brewer's words, when partitioning occurs, the program has to make a partition decision. There are two basic choices for this decision:
  1. Cancel the operation. This decreases availability.
  2. Proceed with the operation. This risks inconsistency.
- Re-trying the operation only delays the decision.

CAP in Practice

- For distributed system design and data processing, it is not helpful to think of CAP as a statement about having to choose two out of three desirable properties. Instead, CAP impacts design decisions related to performance and latency.
- Wide-area networks tends to have high communication latency. Even without network failures, a consensus protocol such as Two-Phase Commit (2PC) with many participating processes will suffer from the high latency. By sacrificing some consistency, better performance can be achieved.
  - The notion of eventual consistency was proposed in this context. Inconsistencies are allowed temporarily during a *failure state*. A specialized protocol then fixes the consistency issues after recovering from the failure state. Analytical results for eventual consistency protocols usually show that with a “sufficiently long” recovery time after a failure state, consistency will eventually be reached.
- The number of times a program re-tries communication with unresponsive nodes determines the tradeoff between consistency and availability. Consider a protocol that tries up to n times to reach another node, then goes ahead with its operation anyway. More re-tries imply a greater emphasis on consistency at the cost of delayed availability.
  - Since the operation was delayed for n attempts, larger n implies lower availability. On the other hand, going ahead with the operation despite having failed to communicate, risks inconsistency. Hence larger n reduces the probability of suffering an inconsistency.

Now that we have a better understanding of the big picture, let us take a closer look at a distributed key-value store, which could be considered a NoSQL database.

Bigtable and HBase

- Bigtable was proposed by Google in a systems research paper:
  - Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. OSDI'06: Seventh Symposium on Operating System Design and Implementation, Seattle, WA, November, 2006
- Apache HBase is an open-source system modeled after Bigtable. It complements Hadoop MapReduce and Spark.

Associative Access

- The MapReduce and Spark programs discussed so far work with data stored in large files, typically stored in HDFS or cloud storage systems such as S3.
- HBase offers an alternative data storage option. What makes HBase attractive for big data analysis compared to simple files?
- HBase supports fast associative access to small amounts of relevant data "hidden" in big data. To understand why this functionality is important, let's see how relational databases use indexes for associative access.

Database Indexes

- Database indexes will be illustrated with a small Student table.
- Assume the table is stored in a file like the one shown schematically next to it. Instead of the entire tuples, only their SIDs are indicated.
  - Notice that in general these tuples could appear in any order.

Data tuples

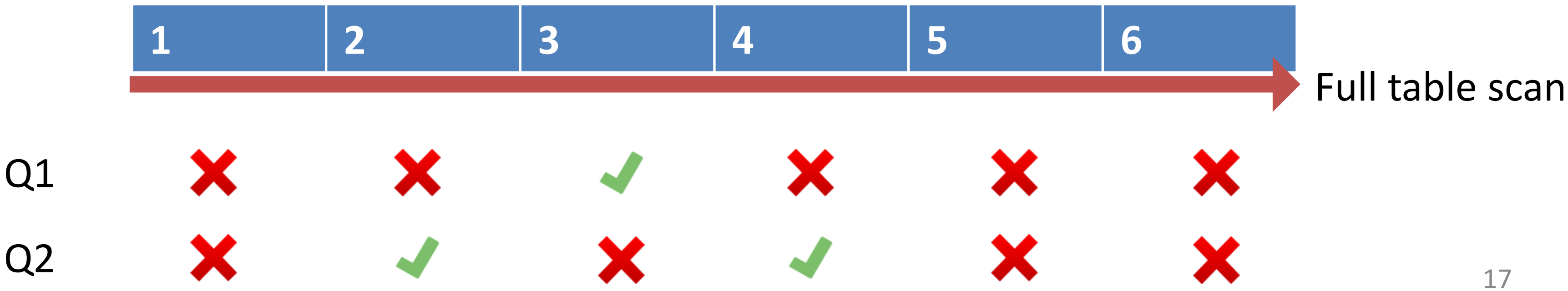
SID	Name	Age	GPA
1	Alice	18	3.5
2	Bob	27	3.4
3	Carla	20	3.8
4	Dan	21	3.9
5	Erin	19	3.6
6	Frank	20	3.8

Data tuples stored in some order in a file

1	2	3	4	5	6

No Index: Scanning a Table

- Consider the following two queries:
  - **Q1**: Find all students named “Carla.”
  - **Q2**: Find all students 21 years or older.
- These queries perform an associative access operation, because they are looking for all tuples *associated* with a given attribute value (the name in Q1) or range of attribute values (the age in Q2).
- Both queries can be answered by scanning the data file from beginning to end. Whenever a tuple satisfying the query condition is encountered during the scan, it will be sent to the output.
- While conceptually simple, scanning an entire table is wasteful for queries that need to access only a few tuples. Such queries are called selective queries. For selective queries, we would want a more efficient way of directly locating the relevant tuples without reading any (or “too many”) of the irrelevant ones. This is where index structures come into the picture.



	1	2	3	4	5	6
Q1	✗	✗	✓	✗	✗	✗
Q2	✗	✓	✗	✓	✗	✗

Hash Index for Equality Conditions

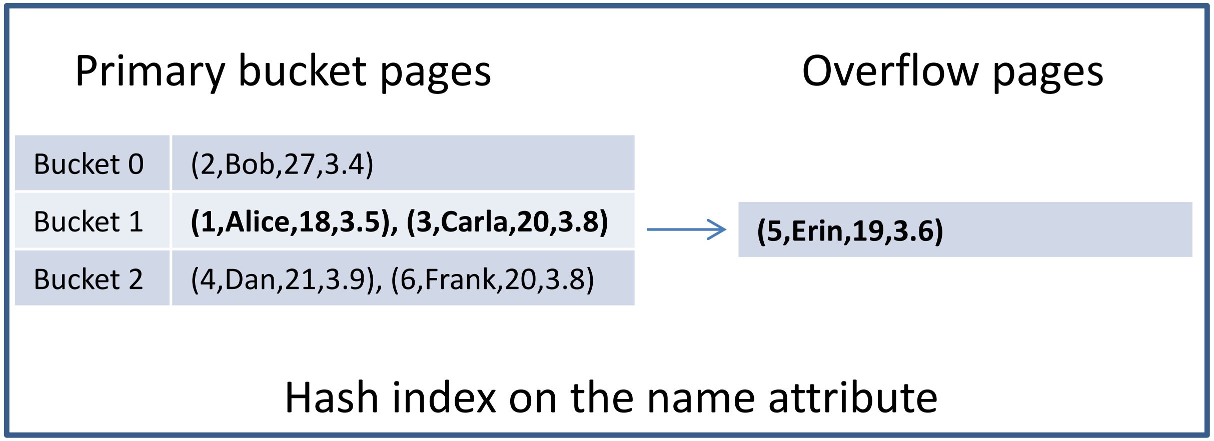
- A database hash index is a disk-based version of the well-known hash map data structure. The basic version of the hash index consists of a hash function h and B buckets.
  - The hash function maps each database tuple to exactly one of the B buckets.
  - Each bucket initially is a single disk page that can hold multiple entries. Once this page fills up, overflow pages are added to the bucket. Depending if the index is clustered or not, the index pages either store the actual tuples or pointers to the tuple locations on disk. For simplicity we will focus on the version that stores tuples in the index.
- Using a hash index with sufficiently large B, relevant tuples can be found in amortized constant (i.e., independent of data size) cost.

Hash Index Example

- The example hash index has three buckets, it stores the tuples, and each index page fits up to two tuples. For Q1's associative access by student name, the index needs a hash function h that hashes each name to one of the three buckets. A good hash function will achieve a nearly uniform assignment.
- How does Q1 take advantage of this index?
  - The database recognizes the equality condition *name = 'Carla'* in the query.
  - It computes h('Carla'), determining that all tuples for name 'Carla' are in bucket 1.
  - Instead of reading the entire table, the database only accesses bucket 1 (including its overflow bucket). The accessed tuples are highlighted in bold. It checks the name constraint on-the-fly, returning only tuple 3.
- Hash indexes can significantly reduce cost for queries with equality conditions.
- For range conditions, they do not work well. Assume Q1 was looking for a all names starting with the letter C. Such names could be hashed to any of the buckets, hence the database would not be able to prune any buckets from the search, resulting in no advantage over the simple scan.

SID	Name	Age	GPA
1	Alice	18	3.5
2	Bob	27	3.4
3	Carla	20	3.8
4	Dan	21	3.9
5	Erin	19	3.6
6	Frank	20	3.8

h('Alice') = 1
h('Bob') = 0
h('Carla') = 1
h('Dan') = 2
h('Erin') = 1
h('Frank') = 2



Primary bucket pages			Overflow pages
Bucket 0	(2,Bob,27,3.4)		
Bucket 1	**(1,Alice,18,3.5), (3,Carla,20,3.8)**	→	**(5,Erin,19,3.6)**
Bucket 2	(4,Dan,21,3.9), (6,Frank,20,3.8)		

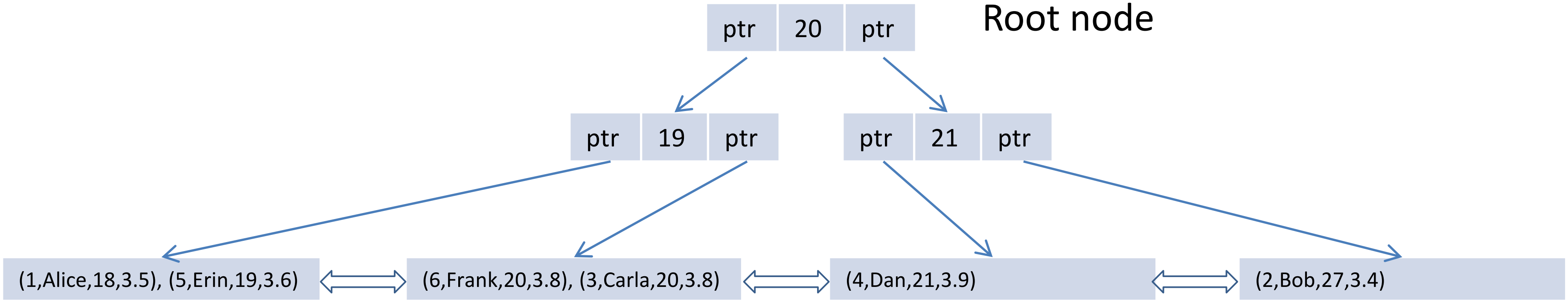
Hash index on the name attribute

B-Tree Index for Range Predicates

- A B-tree supports both equality and range predicates. It organizes the domain of the index key hierarchically. Essentially a B-tree is a disk-based version of the well-known (2,4)-trees and the tree map data structure.
- The non-leaf pages of the B-tree contain search keys and pointers, while the leaf pages contain the actual tuples (or pointers to their locations on disk).
- The search for a given key starts at the tree root, then proceeds along a single path to the leaves. Leaf pages are linked together and store tuples in key order. Hence all matches are found by scanning the leaf level from the initially found page to the “right” or to the “left.”
- By construction, B-trees are balanced, i.e., the path from root to leaf has the same length for every leaf node. This enables the B-tree to guarantee logarithmic (in the data size) cost for finding a relevant leaf. Similarly, B-tree updates can be performed in logarithmic time.

B-Tree Example

- Since Q2 selects tuples by age, the index key has to be the age attribute.
- The root node of the example B-tree contains age 20 as the search key. The left pointer guides the search to all students 20 years or younger, while the right one points to those older than 20 years.
- Similarly, in the next level the pointers guide the search to the corresponding leaf pages based on search keys stored there.
- Notice that the tuples in the leaves are perfectly sorted by age. This happens by design, enabling fast range searches.
- This example gives a flavor of the B-tree structure. In practice, index pages contain many more search keys (and the corresponding pointers).



ptr 20 ptr

Root node

ptr 19 ptr

ptr 21 ptr

(1,Alice,18,3.5), (5,Erin,19,3.6)

(6,Frank,20,3.8), (3,Carla,20,3.8)

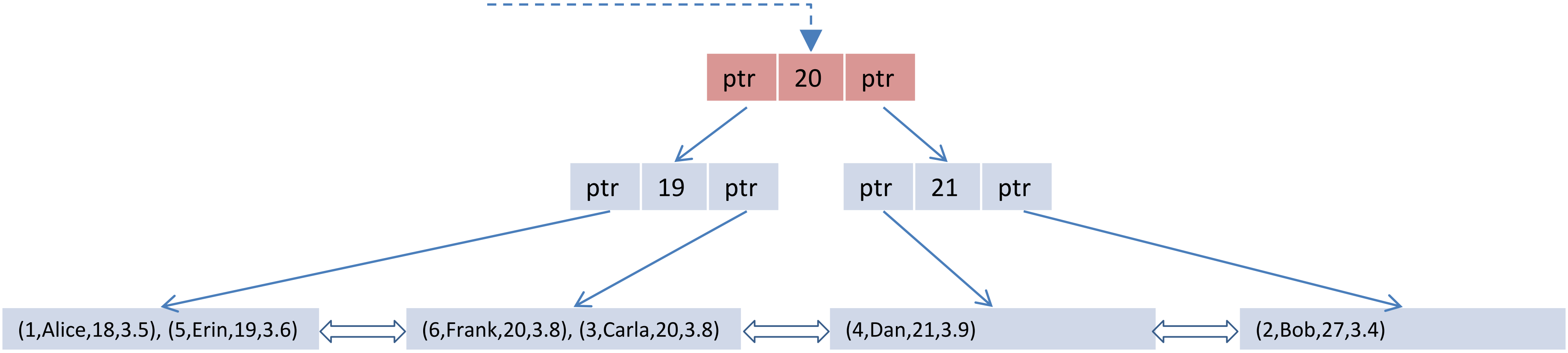
(4,Dan,21,3.9)

(2,Bob,27,3.4)

Leaves, connected through pointers

Processing Q2

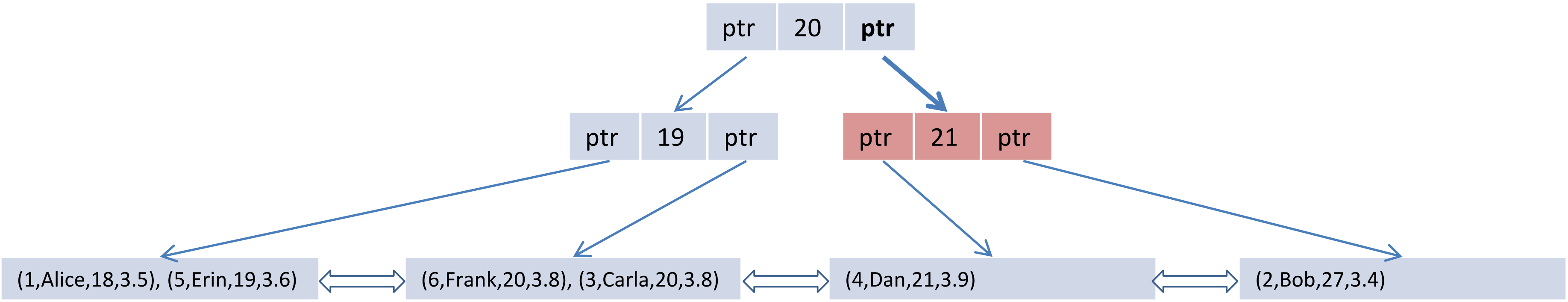
Q2: Find tuples with age **21** or older.


ptr
20
ptr
ptr
19
ptr
ptr
21
ptr
(1,Alice,18,3.5), (5,Erin,19,3.6)
(6,Frank,20,3.8), (3,Carla,20,3.8)
(4,Dan,21,3.9)
(2,Bob,27,3.4)

The search starts at the root node with search key value age = 21.

Processing Q2

Q2: Find tuples with age **21** or older.



ptr 20 **ptr**

ptr 19 ptr

ptr 21 ptr

(1,Alice,18,3.5), (5,Erin,19,3.6)

(6,Frank,20,3.8), (3,Carla,20,3.8)

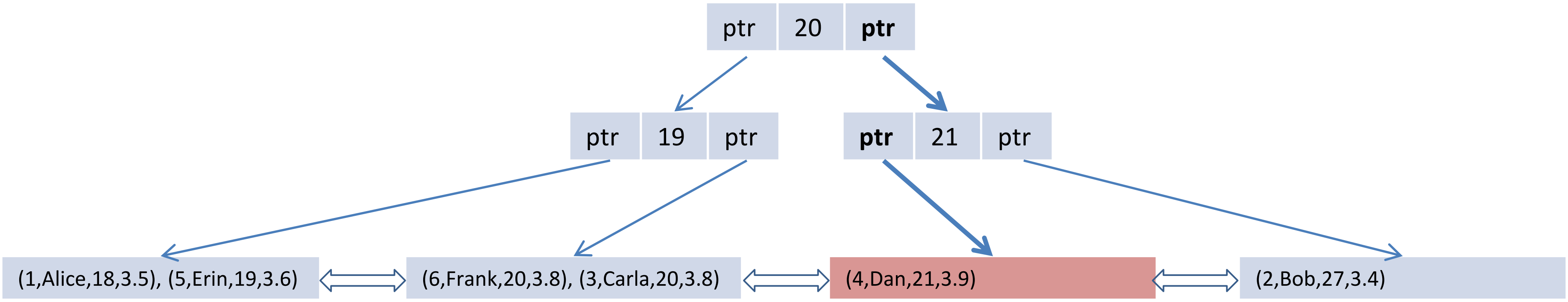
(4,Dan,21,3.9)

(2,Bob,27,3.4)

Since 21 > 20, the search proceeds with the right pointer.

Processing Q2

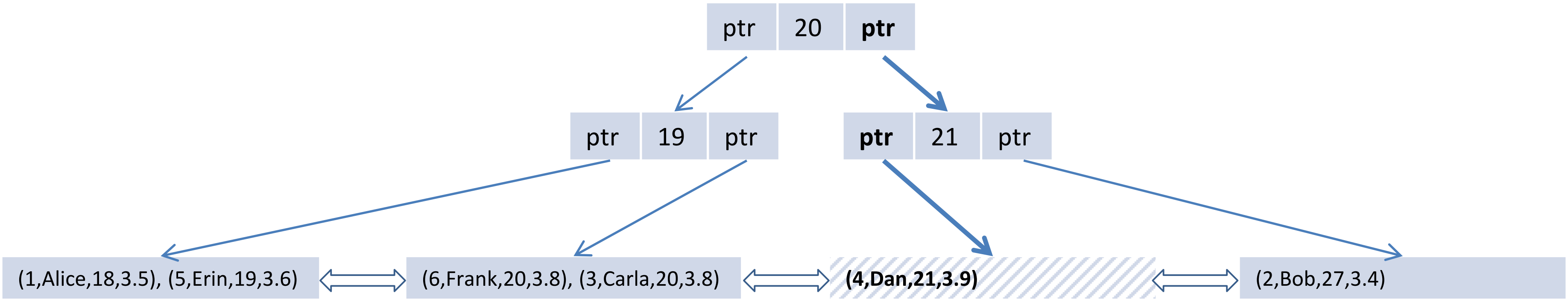
Q2: Find tuples with age **21** or older.


ptr
20
ptr
ptr
19
ptr
ptr
21
ptr
(1,Alice,18,3.5), (5,Erin,19,3.6)
(6,Frank,20,3.8), (3,Carla,20,3.8)
(4,Dan,21,3.9)
(2,Bob,27,3.4)

Since 21 is now equal to the search key, the search proceeds with the left pointer.

Processing Q2

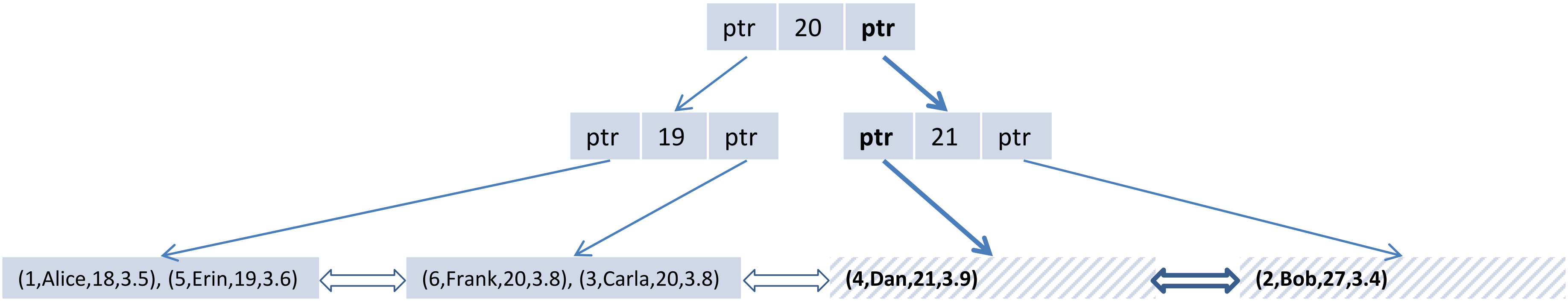
Q2: Find tuples with age **21** or older.


ptr
20
ptr
ptr
19
ptr
ptr
21
ptr
(1,Alice,18,3.5), (5,Erin,19,3.6)
(6,Frank,20,3.8), (3,Carla,20,3.8)
(4,Dan,21,3.9)
(2,Bob,27,3.4)

In the leaf reached, all tuples are checked to see if they satisfy the query condition.

Processing Q2

Q2: Find tuples with age **21** or older.


ptr
20
ptr
ptr
19
ptr
ptr
21
ptr
(1,Alice,18,3.5), (5,Erin,19,3.6)
(6,Frank,20,3.8), (3,Carla,20,3.8)
(4,Dan,21,3.9)
(2,Bob,27,3.4)

Since Q2 specified an inequality that includes all older students, the search continues at the leaf level, following the right pointer to the next leaf until the end.

As the example illustrates, the index nodes guide the search directly to the pages with the relevant tuples. Depending on data and query, cost savings compared to the sequential scan can be significant.

We now discuss how to use HBase as a B-tree.

HBase Overview

- MapReduce and Spark do not have built-in DBMS-style index support. Hence the corresponding programs implementing Q1 and Q2 would scan through the *entire* input file to compute the result. Even though this scan happens in parallel, it wastes resources for selective queries.
- HBase addresses this limitation by adding efficient lookup capability to the Hadoop and Spark ecosystems. One can view it as a somewhat simplified distributed B-tree.
- Like a database, HBase stores tables consisting of rows and columns. And like HDFS, it scales by adding more nodes. With hundreds or thousands of nodes, it can scale to billions of rows and millions of columns.
- However, HBase does not support SQL. It also does not offer transactions or ACID, but ensures row-level atomicity.
- Overall, HBase is neither a distributed file system nor a database. It can most accurately be characterized as a highly scalable distributed key-value store that efficiently supports associative access for selective equality and range queries.

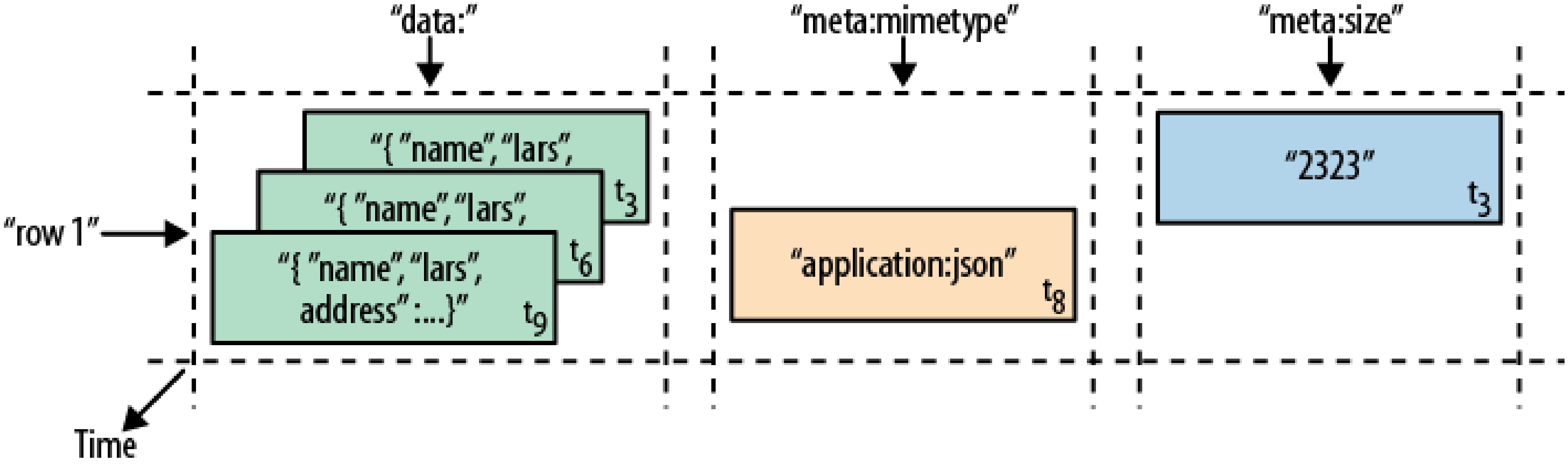
Data Model

- Data is stored in tables, consisting of rows and columns. A cell is defined by a row-column combination; and is versioned.
  - In a table storing a Web crawl, a cell might contain the HTML of www.northeastern.edu on different dates. By default the last three versions are kept. The default version identifier is the time of insertion. Cell content is stored as an un-interpreted array of bytes.
- Each row is uniquely identified by a row key, similar to a database table's primary key. It is a byte array, hence any serializable type can serve as the row key type.
- An HBase table is stored sorted by row key, where the sort is byte-ordered.
  - Like for the B-tree, the sorting property can be exploited to improve performance when accessing data in HBase. Hence the row key needs to be chosen wisely according to data properties and query workload. (This will be discussed soon.)
  - Since keys are sorted based on their byte-array representation, the programmer has to ensure that the byte-encoded key ordering agrees with the desired ordering.
- Columns are grouped into column families. For example, weather data could have a *temperature* family with columns *temperature:air* and *temperature:dew_point.* Column families are stable and need to be specified as part of the table schema definition. On the other hand, individual columns can be added or removed easily. All column family members are stored and managed together.
  - In relational databases, every column has to be specified when creating a table. Adding or removing columns constitutes a major operation.

Illustration

- The following example from "HBase: The Definitive Guide" by Lars George illustrates how data is stored in an HBase table. There are three column families: *data*, *meta*, and *counters*. The example row identified by key “row1” contains multiple versions in the data column family, but only one for the *meta:mimetype* column. Even though the table content conceptually is sparse (note the empty cells in the bottom representation), HBase in practice stores the data very compactly (as shown at the top).
- The timestamp values indicate when the corresponding update was performed. An update can specify values for a subset of the existing columns, or it can create new columns in existing column families on-the-fly.
- Given data as in row 1, where each update affected only some of the columns and where some columns contain multiple versions, what will be returned to a query accessing this row?
  - When asking for a row, by default only the last value in each cell will be returned with its timestamp. In the example, the data with timestamp t_9 in *data*, timestamp t_8 in *meta:mimetype*, timestamp t_3 in *meta:size*, and timestamp t_9 in *counters* will be returned. (The earlier values can also be requested explicitly.)
- When inserting a row, new columns can be created on-the-fly for an existing column family. For instance command put 'mytable', ‘row2', ‘meta:color', ‘blue' adds a row with key ‘row2’ to table ‘mytable.’ New column ‘color’ is created in column family *meta*; the value in this column is ‘blue’.
- Command get 'mytable', ‘row2' will then return

COLUMN	CELL
meta:color	timestamp=…, value=blue



Row Key	Time Stamp	Column “data:”	Column “meta:”		Column “counters:”
			“mimetype”	“size”	“updates”
“row1”	t_3	“{ ”name” : “lars”, “address” : ...}”		“2323”	“1”
	t_6	“{ ”name” : “lars”, “address” : ...}”			“2”
	t_8		“application/json”		
	t_9	“{ ”name” : “lars”, “address” : ...}”			“3”

Data Storage

- HBase can store the data in a variety of file systems, including the local file system, HDFS, and Amazon's S3.
- An HBase table is automatically partitioned into regions, each covering a range of row keys. Each region is managed by a RegionServer. Range partitioning on row keys benefits queries that look up a single row key or a range of row keys.
  - For example, assume a table containing rows with keys 0 to 99 is partitioned into four regions [0,24], [25,49], [50,74], and [75,99]. A client trying to access the row with key 32 only needs to contact the single RegionServer that manages range [25,49]. It finds this RegionServer by contacting the HBase master. Similarly, a client reading rows with keys 57 to 70 would only communicate with the RegionServer responsible for [50,74].
  - What would happen if the table was hash-partitioned into regions? Then keys in range 57 to 70 could be scattered all over the RegionServers, requiring the client to read from all of them.

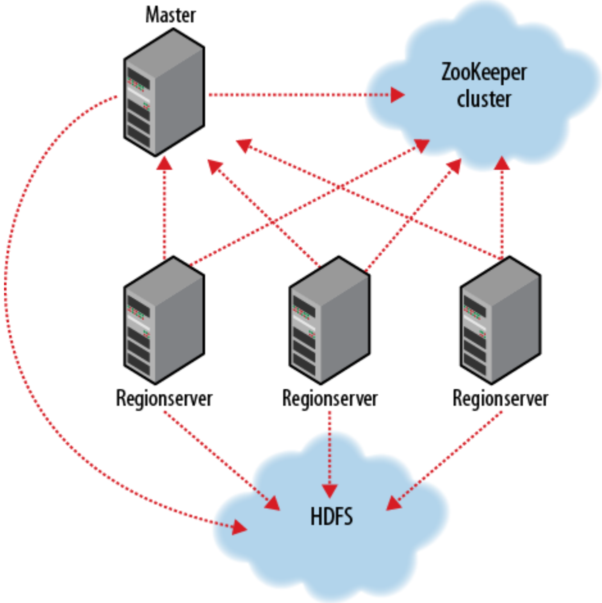


Image source: Hadoop: The Definitive Guide by Tom White

Challenge Question

- For MapReduce and Spark programs, we always attempted to balance load across different machines. For HBase, why do we want to minimize the number of RegionServers contacted? Doesn't this lead to load imbalance on the RegionServers?
  - Yes, but that is a feature, not a bug. Balancing load is desirable for reducing running time. And if HBase had to process only a single query, then that would be a viable goal.
  - However, typically an HBase table is accessed by multiple clients at the same time. In that scenario it is better to optimize for throughput, i.e., the number of requests processed per second. Throughput is higher, the lower the cost per query—and total cost per query decreases with decreasing number of RegionServers accessed by it.

Storage Discussion

- Notice the similarity to the B-tree index: The HBase Master corresponds to the B-tree root node, which contains search keys and pointers to the corresponding RegionServers. A RegionServer corresponds to a giant leaf node in the B-tree. Hence HBase is similar to a very shallow B-tree with giant nodes.
- When using HBase in practice, one needs to be aware that a table is only partitioned once it reaches a certain size. A small table is stored in a single partition on a single RegionServer. As data is inserted and a partition exceeds the maximum allowed size, it will be split and distributed over multiple RegionServers.
  - One can force even a small table to be split by providing key-range data.

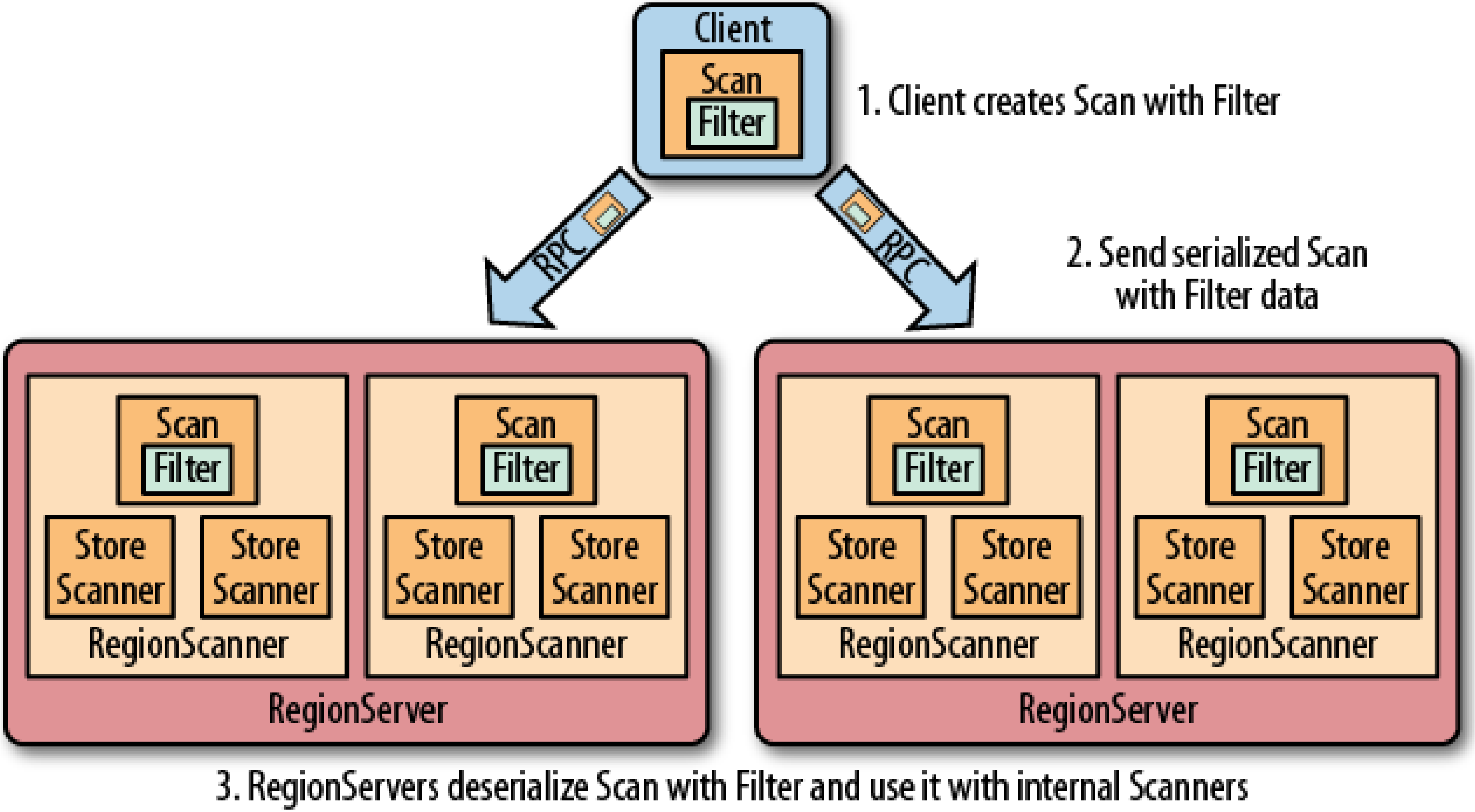
Accessing HBase: The Big Picture

- The process of accessing HBase tables mirrors the way files are accessed in HDFS.
- A client connects to the ZooKeeper service to find the HBase Master. From the Master, it learns about the RegionServer holding the requested data. The client caches the region information it has learned for future accesses.
  - ZooKeeper is a distributed consensus protocol that ensures the existence of exactly one HBase Master, even in the presence of failures.
- The client then contacts the corresponding RegionServer(s) for the actual data directly.
- Write-ahead logging to HDFS ensures durability even if a RegionServer crashes. HBase supports simplified relational-database style redo operations of committed writes.

Accessing HBase: Queries

- Data in an HBase table can be accessed by row key (like a B-tree index lookup) or by scanning.
  - Access by row key requires the client to specify the desired row key.
  - For a scan, the user can specify a *range* consisting of a start row and a stop row. By default, the scan will include all rows.
- For the scan, a filter can also be specified. The filter determines for each row, if it will be sent from HBase to the client. This is useful for selective queries on attributes other than the row key.
  - Consider a query for students with GPA above 3.8 in the Students table, assuming that the HBase table's row key is SID. Since the ordering on SID does not benefit the selection on GPA in any way, the entire table has to be scanned. However, instead of sending all student rows back to the client and removing the irrelevant ones there, adding the filter to the HBase scan will avoid sending the irrelevant rows in the first place. This can significantly reduce cost, saving valuable network bandwidth.

Scan with Filter



Source: HBase: The Definitive Guide, by Lars George

Accessing HBase: Queries (Cont.)

- HBase does not support additional indexes on columns other than the row key. This limits its ability to support associative access on different columns and column combinations. (Though, projects exist for adding such indexes.)
  - Consider again the client looking for students with GPA above 3.8. Without an index on the GPA column, HBase cannot tell which rows contain the desired data, except by reading every row and checking its GPA value. Adding a filter to the scan does not reduce this cost, but only the data transfer to the client.
- In a relational database, a secondary B-tree index on GPA could potentially find the 1% relevant rows by accessing a few index nodes, avoiding access to many of the irrelevant rows.
  - Assume the Students table was organized by age in a B-tree like the example shown before. Since the tuples in the leaves are sorted by age, they are usually not sorted in any way by GPA. Unfortunately, creating the same type of (clustered) index on GPA would resort the tuples by GPA, destroying the order on age. This is where secondary indexes come in handy. Instead of actual tuples, a secondary (non-clustered) index will only store pointers to their locations on disk. These pointers can be sorted by GPA without affecting the sort order of the tuples themselves. While now all pointers to the relevant tuples can be found efficiently as before, the corresponding disk locations could be “all over the place” in the data file. Hence in practice the cost of accessing the tuples will be higher than for the clustered index.
- While secondary indexes often reduce query cost, they increase update cost: Whenever the data changes, the secondary index must be updated, too.

Accessing HBase: Queries (Cont.)

- In contrast to a relational database, HBase does not support joins. Hence the join would have to be executed by the client, e.g., a MapReduce or Spark program that fetches the data from the HBase tables.
- Alternatively, one can store the join result as an HBase table, i.e., de-normalize the data. While this creates redundancy and results in a very large “wide” table (and hence is usually discouraged in relational databases), HBase’s scalability in terms of both the number of rows and columns often makes this approach feasible in practice.

Accessing HBase: Clients

- HBase can be accessed from a variety of clients, including:
  - A MapReduce Java or Spark Scala program.
  - Avro.
  - REST.
  - Thrift.
- Take a look at this simple Java client from "Hadoop: The Definitive Guide" by Tom White that performs a few standard operations against an HBase table:
  - http://www.ccs.neu.edu/home/mirek/code/NewExampleClient.java

HBase and MapReduce/Spark

- HBase is well integrated with Hadoop MapReduce and Spark. For both, one can use the functionality in library org.apache.hadoop.hbase.
  - For instance, class TableInputFormat for Mapper input makes sure each Map task receives a single region of the table. This way regions line up with Map tasks, the same way HDFS file chunks do.
  - Class TableOutputFormat allows a Reducer to write directly to an HBase table, instead of HDFS.

HBase and Hadoop Example

- We illustrate the use of HBase from a MapReduce program with an example taken from "Hadoop: The Definitive Guide" by Tom White.
- In this example, weather station information and their temperature observations need to be loaded into HBase tables. Then these tables are queried to (1) retrieve information about a specified station and (2) retrieve the most recent reports for a specified station.
  - To store information about weather stations and the observations they made, two tables are used: *stations* and *observations*.

The Stations Table

- This table stores detailed information about each weather station, including its name and location.
- Since the data will be looked up based on station IDs, it makes sense to use stationid as the row key.
- It suffices to create a single column family “info”. Individual columns such as info:name, info:location, and info:description will store the station information.

The Observations Table

- This table stores the air temperatures recorded by each station at different times, i.e., each individual measurement is a tuple (stationid, time, airtemp). How should these tuples be stored in the table?
- The most important decision is the choice of the row key. Since HBase stores the rows sorted by key, choosing a good key can significantly improve performance.
- What makes a good key?

Criteria for Choosing a Row Key

- Number of RegionServers accessed: Assume the table is evenly distributed over 100 RegionServers and there are 1000 different stations. If the data is randomly sorted, then a query for records of station S has to access all 100 RegionServers. If the data is sorted by stationid, then most likely all records for station S are located on a single server (assuming approximately uniform distribution). Hence by using stationid as the key, fewer RegionServers need to be contacted.
- Result ordering: HBase scanners can start at a specified row and then return the following rows in order. If these rows are needed in order of some data attribute(s), then making this attribute part of the HBase table's row key will automatically ensure that the scanner delivers the rows in the desired order. (Note the similarity to secondary sort!)
- Uniqueness of rows: The row key should meaningfully distinguish rows from each other.

Uniqueness of Rows

- To better understand this requirement, consider if stationid by itself would make a good row key for the observations table.
  - No. Assume each station reports dozens of temperature measurements daily. By default HBase keeps the three latest versions for each row, therefore with stationid alone as the key, only the last three measurements for each station would be kept. One can increase the number of old row versions to be kept, but this is an inelegant workaround. The real problem here is that the row identifier is not sufficient as a unique identifier.
- Then, how about HBase system time as part of the row key?
  - One might consider wall-clock time of insertion as a “quick-and-dirty” way of creating unique row keys. This usually is not a good idea for two reasons. First, in a distributed system one would need a service that assigns timestamps according to consistent global clock. Otherwise two machines might accidentally choose the same timestamp. Second, from a design point of view, it is preferable to work with identifiers that naturally distinguish different entities. For the observations data, each temperature record is uniquely identified by stationid and time of measurement. Hence the time of measurement is a more sensible choice instead of the time a machine inserted the record into the HBase table.

Choosing the Key for Observations

- Based on the criteria discussed, we choose (stationid, measurement time) as the row key:
  - For the temperature observations, the combination of stationid and time of measurement uniquely identifies each observation tuple, hence (stationid, time) and (time, stationid) are good row key candidates from this point of view.
  - The workload consists of queries looking for measurements from an individual station. This again suggests that stationid should be part of the row key. If the data is sorted by stationid, then such requests will access a comparably small range in the HBase table, i.e., only one or maybe two RegionServers (depending on the number of partitions and measurements per station).
  - The user wants to receive temperature measurements from a station in temporal order, starting with the most recent. This suggests that the time of measurement should be part of the key, but only secondary to the stationid.

Choosing the Key for Observations (Cont.)

- In summary, the row key for the observations table should be the combined byte array consisting of stationid and time. Since keys are sorted based on the natural order of these bytearrays, the following needs to be taken into account:
  - Converting key fields to a byte array sometimes requires padding, i.e., adding additional digits (usually zeros) to make sure the byte array ordering agrees with the intended ordering of the composite keys. For example, if stationid values vary in their length when transformed to a byte array, then padding is needed to make all of these byte arrays equal in length before concatenating them with the corresponding byte array for the time field.
  - The natural conversion from timestamp to byte array would result in an increasing sort order on the time field. Hence the byte array for the time field needs to be *inverted* before concatenating it with the corresponding stationid byte array.
- Take a look at the source code from Tom White's book at http://www.ccs.neu.edu/home/mirek/code/RowKeyConverter.java.
  - It assumes that no padding is needed for stationid; reverseOrderTimestamp achieves sorting in decreasing order of observation time.

HBase MapReduce Code: Data Import

- First import data from a source (usually a file) into HBase. Here is the code from Tom White's book:
  - http://www.ccs.neu.edu/home/mirek/code/NewHBaseStationImporter.java (it uses http://www.ccs.neu.edu/home/mirek/code/NewHBaseStationQuery.java)
  - http://www.ccs.neu.edu/home/mirek/code/HBaseTemperatureImporter.java (it uses http://www.ccs.neu.edu/home/mirek/code/NewHBaseTemperatureQuery.java)
- The Map function of the temperature importer program reads a temperature record and parses it. Instead of writing the corresponding key-value pair to an HDFS file or passing it to a Reducer, this program inserts it into an HBase table.
- The program assumes that the observations table already exists.

Performance Considerations

- To populate an HBase table with big data, use a Map-only job as shown above. (Reducers can use the same approach.) When the HBase libraries are used correctly, the connection from a Map task to HBase is established and closed only once, not for every map call. Spark works similarly with the HBase libraries.
- When inserting into an empty table, this table will not be partitioned over multiple RegionServers until it grows large enough. Hence initially all Mappers will send their records to the same RegionServer, creating a bottleneck. As the table grows and gets distributed over multiple RegionServers, insert performance improves. HBase also allows small tables to be pre-partitioned to avoid this bottleneck.
- To distribute data-import load evenly over the RegionServers, data should be inserted in random order of the row key. Avoid situations where all Mappers are likely to write to the same RegionServer at the same time.
- When performing a huge number of insertions, make sure the insert operations are buffered. Buffering allows efficient bulk inserts into HBase, instead of costly individual ones.
  - Check if the put() operation uses buffering. You might have to disable auto-flush and set the size of the write buffer. When using buffering, make sure the buffer is flushed in the end. Take a look at http://www.ccs.neu.edu/home/mirek/code/HBaseTemperatureDirectImporter.java and http://www.ccs.neu.edu/home/mirek/code/HBaseTemperatureBulkImporter.java.

HBase MapReduce Code: Queries

- A lookup by row key on the stations table is shown in http://www.ccs.neu.edu/home/mirek/code/NewHBaseStationQuery.java
  - Since HBase tables are range-partitioned on row key, the get() request for a specific stationid will request data from a single RegionServer, minimizing cost. That RegionServer can quickly find the desired row using binary search. This is much cheaper than scanning through the entire table.
  - The RegionServer to be contacted for a given row key is found as follows. The HBase Master records for each RegionServer the minimal and maximal key value for the range of rows it manages for an HBase table. Since the data is range-partitioned on the row key, there is at most one RegionServer whose range includes the given key.

HBase MapReduce Code: Queries

- A scan of the observation table is shown in http://www.ccs.neu.edu/home/mirek/code/NewHBaseTemperatureQuery.java
    - This program retrieves a block of rows. The Scan object is instantiated with a starting row, identified through a row key. In the example, we are interested in the maxCount most recent observations for a given station. Not knowing the date of the most recent observation, one has to "play it safe" and use the combination (stationId, Long.MAX_VALUE) to start at the largest possible timestamp. If the start row key does not exist, then the scan will automatically start at the next row.
    - One can similarly specify a stop row, i.e., a key for the last row to be accessed. This further reduces cost, because partitions outside this range would not participate in the query at all. (This is like working with a much smaller HBase table!) In the example, the user is interested in a certain number of rows, hence the scanner stops reading at the right row anyway.
    - The scanner retrieves rows in row-key order. Hence if this order agrees with the desired output order, no sorting in user code is needed. The rows can be traversed in order using a ResultScanner. It works similar to an iterator in Java.

In the final section of this module, we explore Hive.

Note: some of this information might be slightly out of date. It will be updated periodically (next time: summer 2019).

Hive

- Hive was initially developed by Facebook, but is now an Apache open source project.
- It originally added SQL-style data analysis functionality on top of Hadoop MapReduce.
  - Users can write queries in HiveQL, a dialect of SQL. These queries are automatically converted to plain Java MapReduce programs and can then be executed on a Hadoop cluster.
- Hive can now also execute queries via Spark.
- Hive can also take on the role of a database server.

System Overview

- Typically Hive is run on a single machine. There it transforms user queries into MapReduce or Spark jobs.
- One can also run Hive as a database server, allowing applications to connect to it and run Hive commands using interfaces such as Thrift, JDBC, and ODBC.
- Data is represented through tables, like in a relational database. A special database called *metastore* contains the metadata, in particular table schemas.
- Hive can be run in interactive mode (using the Hive shell) or in batch mode, passing an input file containing a Hive query.

Hive Storage

- Hive can use a variety of file systems to store its data, including the local file system, HDFS, and S3. From a physical storage point of view, there are two types of tables:
  - For a managed table, Hive moves the corresponding data files into its own warehouse directory. Assuming the file was already located in the same file system, this simply performs a renaming operation, requiring no physical data movement.
  - An external table remains at a location outside the Hive warehouse directory. When creating an external table, Hive simply records the data file's location. At query time, it will read the data from there. Hence the data file does not even need to exist at table creation time, as long as it is available at query time. This enables assigning different schemas to the same file.

Creating Hive Tables

- As in a relational database, the CREATE TABLE statement creates a database table. It specifies table name and schema, i.e., all attributes and their types. The ROW FORMAT clause tells Hive how to interpret the input data.

```
CREATE TABLE records (year STRING, temperature INT, quality INT)
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '\t';
```

Loading Data

- The LOAD statement does not load any data tuples into the table. For managed tables, Hive moves the input file into its warehouse directory, into a subdirectory with the same name as the table. For an external table, only the input file location is recorded. Data will only be loaded lazily at query execution time.
  - This is different from a relational database where all data has to be physically imported into the tables before it can be queried.
- The lazy approach avoids data import cost until the data is queried. On the downside, the on-the-fly parsing of the input file at query time increases query execution cost and might uncover data parsing errors that would not be discovered beforehand.
  - The lazy approach to data loading makes sense for Hive, because a Hive query is converted to a MapReduce or Spark job executed on the input file. Tables and SQL-like query functionality simply provide a convenient interface.

```
LOAD DATA INPATH 'input/ncdc/micro-tab/sample.txt'
OVERWRITE INTO TABLE records;
```

Partitions

- Tables can be physically divided into partitions based on the value of designated partition column(s). This feature can reduce query cost for big data.
  - Consider an Internet application that creates a massive log of user activities. Assume a typical query focuses on users from a single country and their activity in the most recent month. In this case the log table should be partitioned by country and date, so that only a small number of partitions is accessed.
- The partition attribute’s values determine the directory structure where the files with the corresponding data are stored. For example, the data for 10-10-2014 and USA would be stored in subdirectory …/10-10-2014/USA. Since the country and date are encoded in the directory names, the corresponding attributes will not be part of the table schema any more.
  - Partition attributes can still be used in a query. That query can access any number of the partitions.
- The example code creates a partitioned table for a log file whose tuples consist of a date/time field, country, and a line of text. Date and country are used for partitioning, while the time of day value and the text line define the table schema.

```
CREATE TABLE logs (timestamp BIGINT, line STRING)
PARTITIONED BY (date STRING, country STRING);
```

Buckets

- Buckets do not create separate directories like partitions. Instead, they are used for re-ordering data in a table. The CLUSTERED BY clause groups the data by some attribute(s) and the SORTED BY clause ensures sorting within each group.
- Hive determines the bucket a tuple belongs to by hashing on the clustering attribute(s), then taking this hash output value modulo the number of buckets.
- This functionality is particularly useful for equi-joins, if both input tables are bucketed on the same columns (and the join columns are used for bucketing). This enables an efficient hash-join and sort-merge join style implementation—the latter only if sorting on the join attribute(s) was applied within each group.

```
CREATE TABLE bucketed_users (id INT, name STRING)
CLUSTERED BY (id) SORTED BY (id ASC) INTO 4 BUCKETS;
```

Hive Queries

```
SELECT year, MAX(temperature)
FROM records
WHERE temperature != 9999
   AND (quality = 0 OR quality = 1 OR quality = 4 OR quality = 5 OR quality = 9)
GROUP BY year;
```

- The above query in HiveQL finds the maximum temperature for each year, only considering observation reports with valid temperatures and quality.
- HiveQL is very similar to SQL. It does not fully support the SQL-92 standard, but can express many common SQL queries. In fact, the above example is a perfectly fine SQL query.
  - Note that Hive allows complex types such as ARRAY, MAP, and STRUCT to be stored in a database column. This differs from relational databases, which usually limit columns to primitive types. (Complex types need to be expressed through the use of additional tables.)
- For more details about HiveQL, consult the Hive manual.

Joins in Hive

- Hive supports inner join, outer join, and semi join. It uses a rule-based optimizer for generating efficient query plans before performing the conversion to a Hadoop program. A more sophisticated cost-based optimizer could be added in the future.
- The examples below illustrate various joins of tables *sales* and *things*. All these HiveQL queries are also valid SQL queries.

```
SELECT sales.*, things.*
FROM sales JOIN things ON (sales.id = things.id);
```

```
SELECT sales.*, things.*
FROM sales LEFT OUTER JOIN things ON (sales.id = things.id);
```

```
SELECT * FROM things LEFT SEMI JOIN sales ON (sales.id = things.id);
```

Advanced Query Features

- HiveQL supports sub-queries, but in a much more limited way than relational databases. (Initially sub-queries were restricted to the FROM clause, but this might change as the system evolves.)
- Hive also supports the creation of views, i.e., tables defined by a HiveQL query. However, these views cannot be materialized and hence need to be generated on-the-fly when the query referencing a view is executed.
- Like in relational databases, HiveQL functionality can be extended through user-defined functions. These functions can be written in Java.

Hive versus Relational Databases

- Both Hive and relational databases need to create tables, but they differ in the way data is loaded.
  - In a relational database, data needs to be physically loaded into the tables before it can be queried. At the time the data is loaded, the table schema is enforced. This approach is called "schema on write."
  - Hive's LOAD statement does not read or parse the data file at all. Instead, Hive enforces the table schema on-the-fly at query execution time. This approach is referred to as "schema on read."
  - Schema on write requires greater effort at data load time, but usually results in better query performance because the data is already parsed and readily available in a database-controlled format. Having the data managed by the database also allows it to create index structures or apply compression. Schema on read eliminates the initial startup cost for data loading, but it increases query execution cost.
- A major advantage of schema on read is that it makes it easy to change the schema of a table. In a relational database, a schema change leads to potentially costly physical data re-organization. In Hive, it merely changes the way the data is being interpreted when it is read from file during query execution. Furthermore, with external tables, it is possible to associate different table names and schemas with the same input file, without having to copy the data. This would not work with Hive's internal tables, because the file is stored in a directory named after the table; hence one cannot associate two different table names with the same file.

Hive versus Relational Databases (Cont.)

- In relational databases, table entries are fully updateable. Hive does not support updates or deletions of existing data, only inserting of new rows.
- Hive indexing support is rudimentary compared to relational databases, but is being improved and expanded. However, note that in order to create an index, the data file has to be physically loaded, the index be materialized in some form, and then be managed by Hive.
- Hive initially did not support ACID transactions, but locking at different levels of granularity (table and partition) was added. Expect Hive to become more like a relational database in the near future.
- Hive's data model supports complex types (ARRAY, MAP, STRUCT) in table columns, something generally not allowed by relational systems.
- Hive currently relies on a much less sophisticated optimizer than most commercial databases.

Summary

- When processing big data in a distributed system, we are facing inherent tradeoffs between data consistency, data availability, and the system's ability to tolerate network partitions (or high latency). This means that any scalable data processing system has to make compromises on at least one of these aspects.
  - Relational databases traditionally emphasize consistency and availability, limiting their scalability in terms of the number of nodes participating in a transaction.
  - Some NoSQL databases relax consistency in order to achieve high availability even when running on many nodes.
  - Hadoop and Spark appear to magically achieve high data availability and scalability to many nodes, without compromising consistency. In reality, they limit availability because existing data cannot be modified. And dealing with machine failures through task re-execution can result in slightly weaker consistency guarantees if a program is non-deterministic.

Quick Note about Nondeterministic Programs

- A program is non-deterministic if its output depends on random choices. However, not all MapReduce or Spark programs using random number generators will necessarily suffer from data consistency issues.
- Citation from [Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004]
  - “In the presence of non-deterministic operators, the output of a particular reduce task R1 is equivalent to the output for R1 produced by a sequential execution of the non-deterministic program. However, the output for a different reduce task R2 may correspond to the output for R2 produced by a different sequential execution of the non-deterministic program. Consider map task M and reduce tasks R1 and R2. Let e(Ri) be the execution of Ri that committed (there is exactly one such execution). The weaker semantics arise because e(R1) may have read the output produced by one execution of M and e(R2) may have read the output produced by a different execution of M.”

Summary (Cont.)

- HBase is a highly scalable key-value store that is well integrated with Hadoop MapReduce and Spark. It adds the ability to look up individual rows or ranges of rows based on a row key, essentially providing some form of indexing capability for MapReduce and Spark.
- Hive provides a scalable distributed data warehousing environment. It trades scalability for somewhat restricted functionality compared to traditional relational data warehousing technology.

CYK: Question 1

- For each question, select if it is true, false, or not enough information to answer it.

1. HBase supports relational-database style transactions and SQL.
2. HBase works really well for cases where a MapReduce program needs to look up only a few records based on a key attribute.
3. We can write plain-Java Hadoop programs that read their input data from an HBase table and store job results in an HBase table.
4. HBase is more scalable than Hive.

Question 2

- Consider an HBase table Reviews with row key (*productID*, *customerID, date*) and columns *price* and *score*. Each customer and product has a unique ID. Every time a customer purchases a product, she submits a score. Assume there are 1000 different products and 10,000 customers. Customer-product pairs are uniformly distributed in the review data. The same review information is also stored in an HDFS file.
- For each of the following queries, assume it is the only query you need to run, but you will need to run it every day. Decide if you want to create the Reviews table as described above, or if you would rather use the file in HDFS. Select both if you believe this cannot be answered clearly in either way.

1. Find the average score for each productID.
2. Find the average score for productID P10.
3. Find all tuples for the combination of productID P5, customerID C12, and date 10-10-2013.
4. Find all tuples for customerID C91.

References

- Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.* 26, 2, Article 4 (June 2008)
  - https://scholar.google.com/scholar?cluster=535416719812038974&hl=en&as_sdt=0,22
- Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. ACM SIGOPS Symposium on Operating Systems Principles* (SOSP), pages 205-220, 2007
  - https://scholar.google.com/scholar?cluster=5432858092023181552&hl=en&as_sdt=0,22

References

- Brewer, E., "CAP twelve years later: How the "rules" have changed," IEEE Computer, vol.45, no.2, pp.23-29, Feb. 2012
  - https://scholar.google.com/scholar?cluster=17642052422667212790&hl=en&as_sdt=0,22
- Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News 33, 2 (June 2002), 51-59
  - https://scholar.google.com/scholar?cluster=17914402714677808535&hl=en&as_sdt=0,22