1. The Rise of NoSQL Databases

- As discussed in a previous module, 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 will usually 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 ACID 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 in a distributed setting, all processes participating in a distributed transaction have to agree on the final outcome.
  - Data needs to be imported into the DBMS before any query can be executed. This process includes careful schema design (relations and their attributes, usually also index structures) and data cleaning. Even though this initial effort usually pays a dividend at query execution time, many users dislike this heavy startup cost and prefer to get “something” running quickly, dealing with data quality and structure issues later as necessary. The latter approach is particularly appealing for applications where the user needs a quick estimate to determine if it is worth exploring the data in more depth.

1. The Rise of NoSQL Databases

- To address these shortcomings, so-called *NoSQL* databases have appeared. NoSQL is a catchy term referring to a wide variety of approaches that present themselves as alternatives to relational technology. Recent and current (as of summer 2014) examples include the following: (As systems come and go, or change their features, this list might look different today. Still, these examples illustrate prototypical approaches to big data management.)
  - 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.
- These examples illustrate different tradeoffs between relational and NoSQL systems:
  - Relational databases emphasize data consistency and durability, resulting in comparably limited scalability and performance. Recall that 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 can become complicated for 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.

2. The CAP Theorem

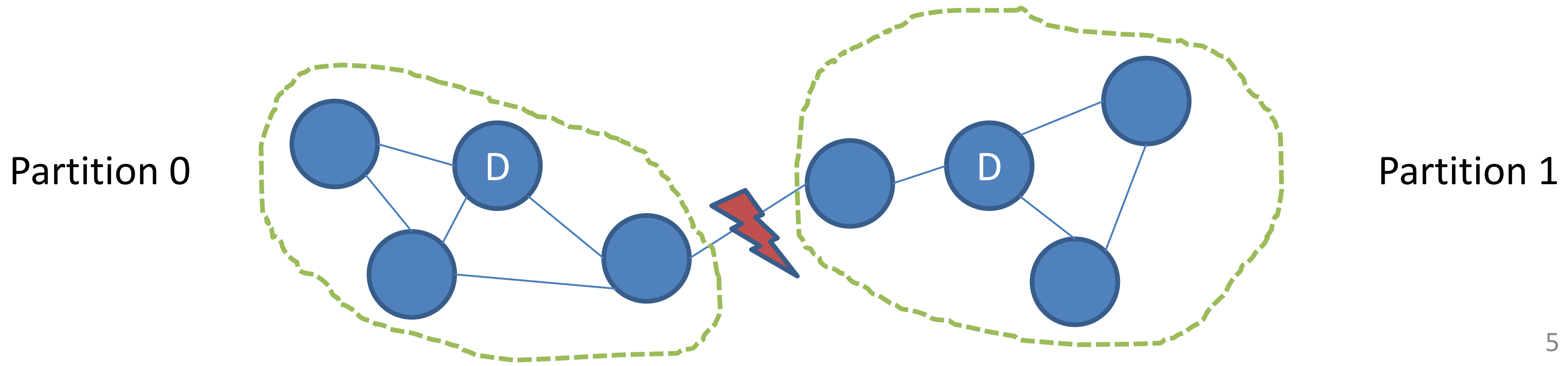
- At the 2000 ACM Symposium on Principles of Distributed Computing (PODC), Eric Brewer proposed the now famous CAP conjecture for networked shared-data systems. The CAP conjecture states that there is an inherent tradeoff between consistency, availability (for data updates), and tolerance to network 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, doi: 10.1109/MC.2012.37; URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6133253&isnumber=6155638]
  – Lynch and Gilbert's paper is available from the ACM Digital Library: [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. DOI=10.1145/564585.564601; URL http://doi.acm.org/10.1145/564585.564601]

2. The CAP Theorem

- 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.

2.1 Understanding the CAP Theorem

- To better understand CAP, consider a distributed system with two different nodes each holding a copy of data item D, e.g., the bank account 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.



2.2 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.
- Notice that re-trying the operation just delays the decision.

2.3 CAP in Practice

- For system design and data processing in distributed systems in practice, it is usually not helpful to think of the CAP theorem as a statement about having to choose two out of three desirable properties. Instead, CAP impacts design decisions related to *performance* and *latency*:
  - In a wide-area network, latency of communication tends to be comparably high. Even if there are no network partitions, running 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 to occur 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 across the system.
  - The number of times a program re-tries communication with unreachable 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.
    - To understand why this holds, consider the case where the program performed the operation after n unsuccessful communication attempts. Since the operation was delayed for these 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 means allowing for greater latency in order to reduce the probability of suffering an inconsistency.

3. 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 and works well with it.

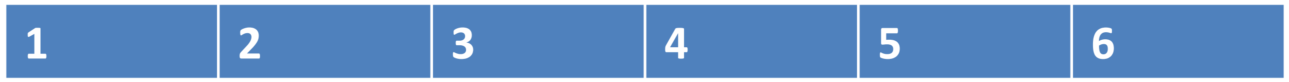
3.1 Associative Access

- The Hadoop programs discussed so far work with data stored in large files in the HDFS distributed file system. HBase offers an alternative data storage option. What makes HBase attractive for big data analysis compared to HDFS?
- 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.

3.1.1 Database Indexes

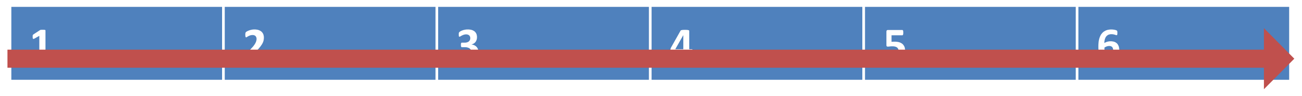
- Database indexes will be illustrated with a small Student table example.
- Assume the table is stored in a file like the one shown schematically next to it. Instead of the entire tuples, only their SIDs are indicated.
  - Notice that in general these tuples could appear in any order.

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



3.1.2 No Index: Scanning a Table

- Consider the following two queries:
  - **Q1**: Find all tuples for students named “Carla”.
  - **Q2**: Find all tuples for 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.



Q1	✗	✗	✓	✗	✗	✗
Q2	✗	✓	✗	✓	✗	✗

3.1.3 Hash Index for Equality Conditions

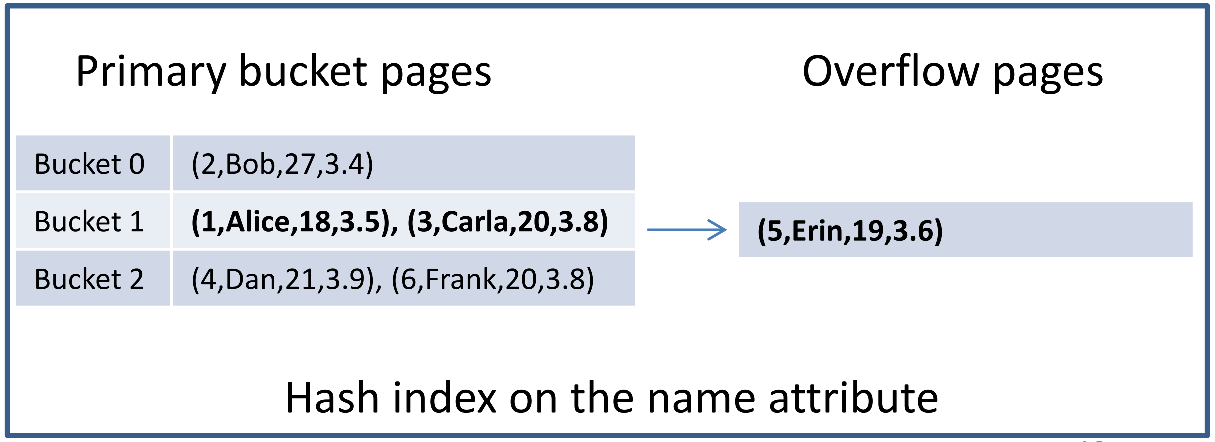
- A database *hash index* is a disk-based version of the well-known hash map data structure.
- The basic version of the hash index consists of a hash function h and B buckets.
  - The hash function maps each database tuple to exactly one of the B buckets.
  - Each bucket initially is a single disk page 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.

3.1.4 Hash Index Example

- For the student table example, assume the hash index has three buckets, it stores the entire 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 a name to one of the three buckets.
  - As the example illustrates, 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 corresponding tuples are shown in bold face in the illustration. Going through all tuples in this bucket, it will only deliver the tuple with SID 3 to the output.
- Depending on query condition and data properties, hash indexes can significantly reduce cost for queries with equality conditions.
- For range conditions, hash indexes do not work well. Assume Q1 was looking for a all names starting with the letter C. Names starting with C could potentially be hashed to any of the three 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

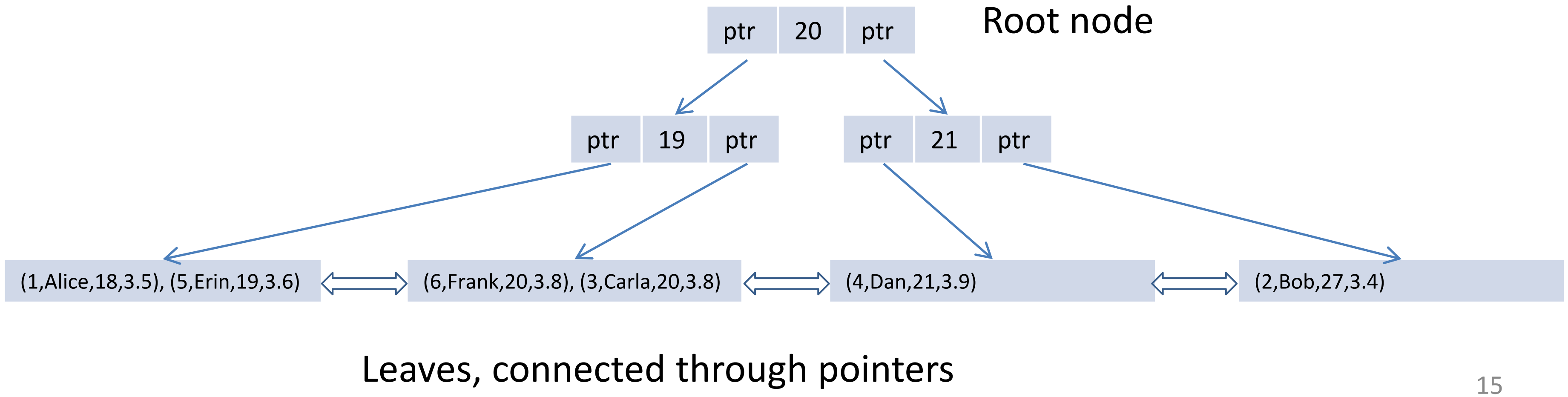

Primary bucket pages
Overflow pages
Bucket 0
(2,Bob,27,3.4)
Bucket 1
(1,Alice,18,3.5), (3,Carla,20,3.8)
(5,Erin,19,3.6)
Bucket 2
(4,Dan,21,3.9), (6,Frank,20,3.8)
Hash index on the name attribute

3.1.5 B-Tree Index for Range Predicates

- A *B-tree* supports both equality and range predicates. It organizes the domain of the index key hierarchically in order to find a given key in logarithmic cost. 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.

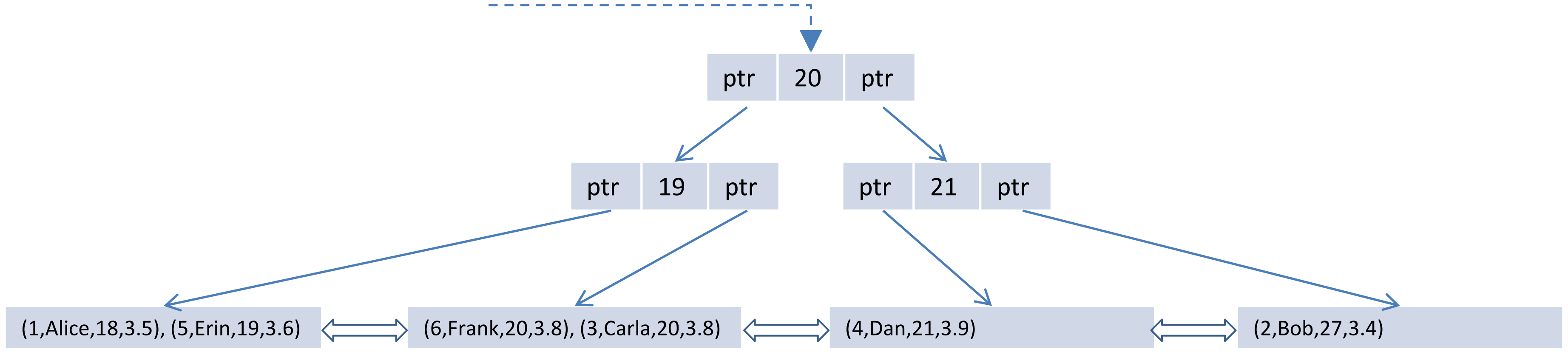
3.1.6 B-Tree Example

- 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

3.1.7 Processing Q2

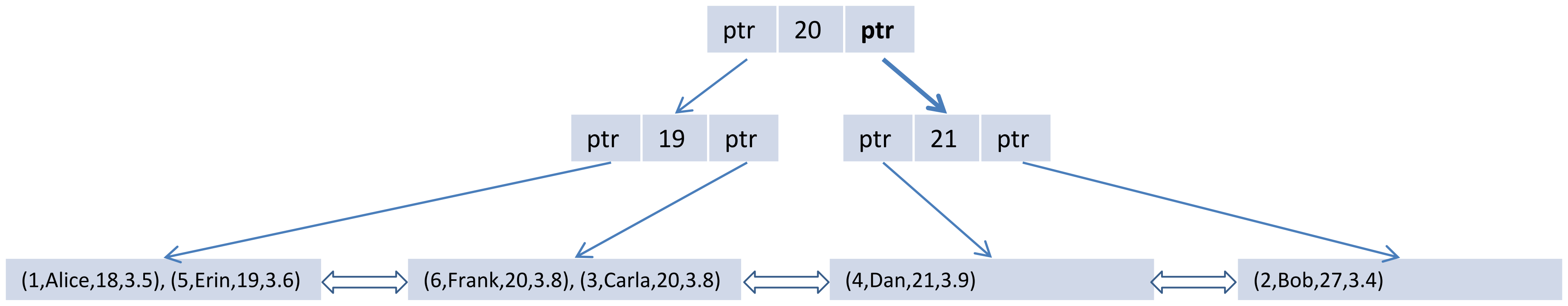
Q2: Find tuples with age **21** or older.


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.

3.1.7 Processing Q2

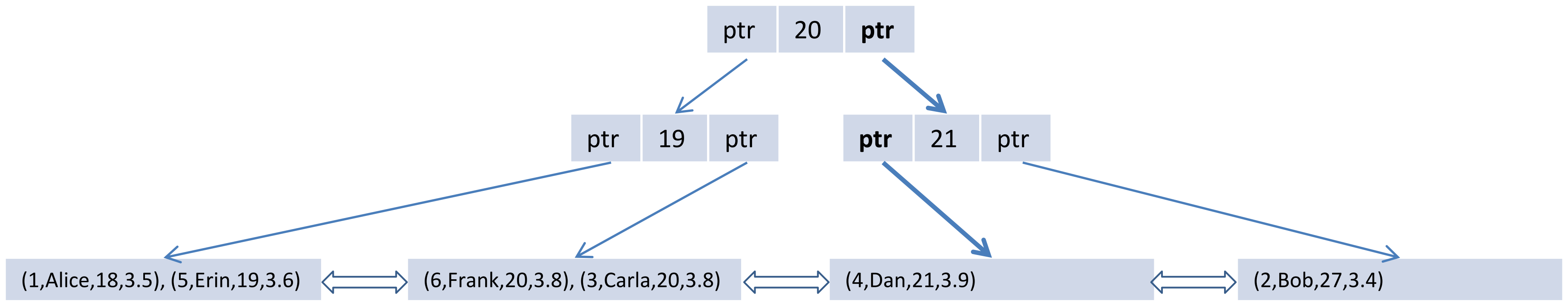
Q2: Find tuples with age **21** or older.


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.

3.1.7 Processing Q2

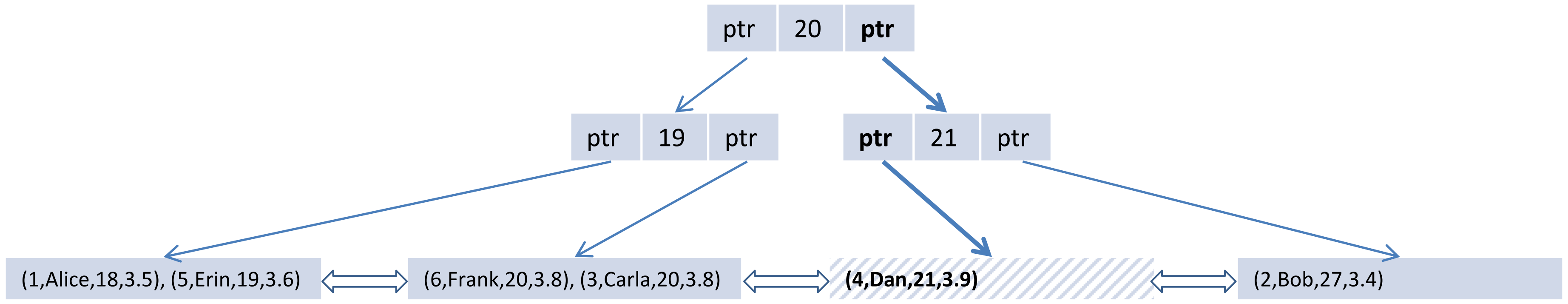
Q2: Find tuples with age **21** or older.


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.

3.1.7 Processing Q2

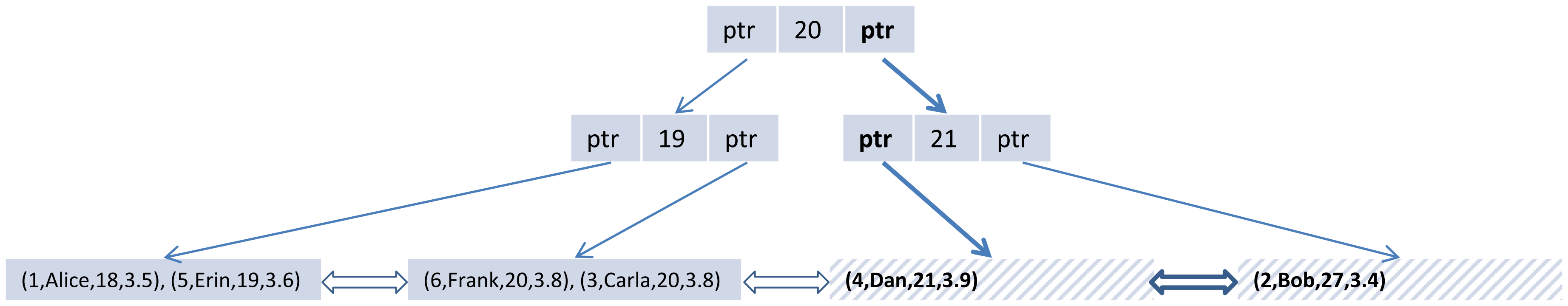
Q2: Find tuples with age **21** or older.


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.

3.1.7 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.

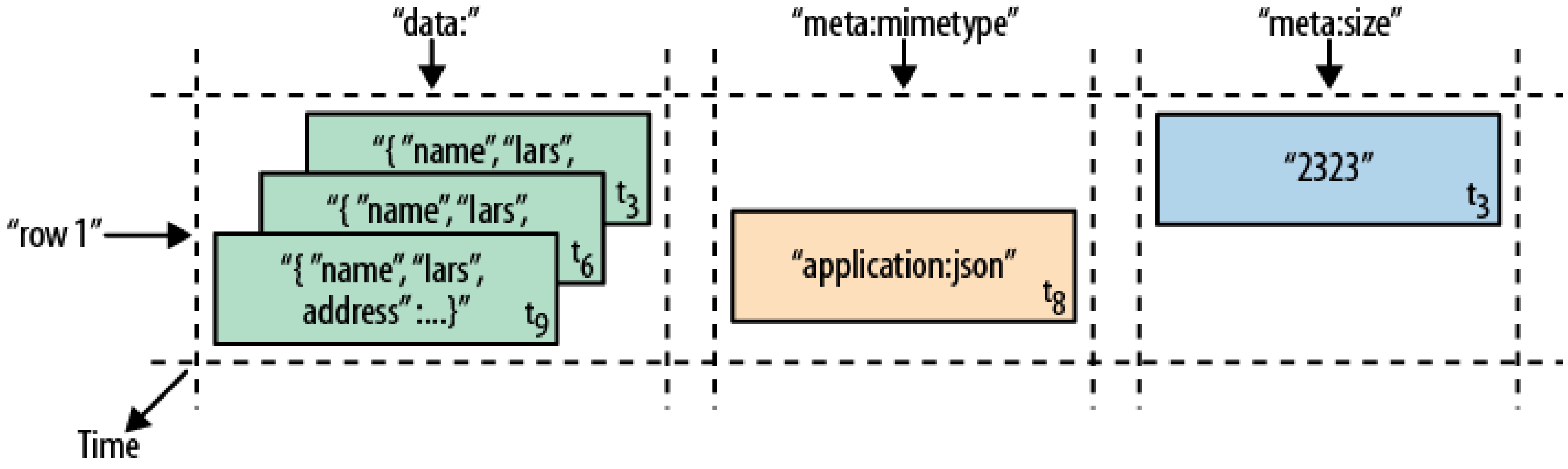
As the example illustrates, the index nodes guide the search directly to the pages with the relevant tuples. Depending on data and query, cost savings compared to the sequential scan can be significant.

3.2 HBase Overview

- In Hadoop as discussed so far, there are no index structures. Hence when implementing Q1 and Q2 in MapReduce, they would scan through the entire HDFS input file to compute the result. Even though this scan happens in parallel, it would still be inefficient (and waste resources) for selective queries.
- HBase addresses this limitation by adding efficient lookup capability to Hadoop. One can view it as a somewhat simplified distributed B-tree.
- Like a database, it 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.

3.3 Data Model

- Data is stored in tables, each consisting of rows and columns. A *cell* is defined by a row and column combination.
- Each cell is versioned, e.g., in a table storing a snapshot of the Web, a cell might contain the HTML content of www.neu.edu on different dates. By default the last three versions of a cell are kept and the default version identifier is the time of insertion. The cell content is stored as an un-interpreted array of bytes.
- Each row is uniquely identified by a *row key*. Intuitively, the row key corresponds to the primary key in a relational database table. 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. (Compare this to the B-tree whose leaf level also stores tuples sorted by the index key!)
  - 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 later in this module.)
  - Since keys are sorted based on their byte-array representation, the programmer has to make sure 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 that contains 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.
  - Contrast this with relational databases. There the exact schema, i.e., every single column, has to be specified when creating a table. Adding or removing columns constitutes a major operation.



Row Key	Time Stamp	Column "data:"	Column "meta:" "mimetype"	"size"	Column "counters:" "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"

3.4 Data Storage

- An HBase table is automatically partitioned into regions, such that each region corresponds to a range of row keys. Each region is managed by a RegionServer. HBase can store the data in a variety of file systems, including the local file system, HDFS, and Amazon's S3. Range partitioning on row keys benefits queries that access a specific row using its key, or a range of rows starting at a specific key.
  - For example, assume a table containing rows with keys 0 to 99 is partitioned into four regions corresponding to key ranges [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 several rows starting at key 57 would only communicate with the RegionServer responsible for [50,74], until it advances to row 75. [Link: What would happen if the table was randomly partitioned into regions?]
- 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 an distributed over multiple RegionServers.

3.4 Data Storage

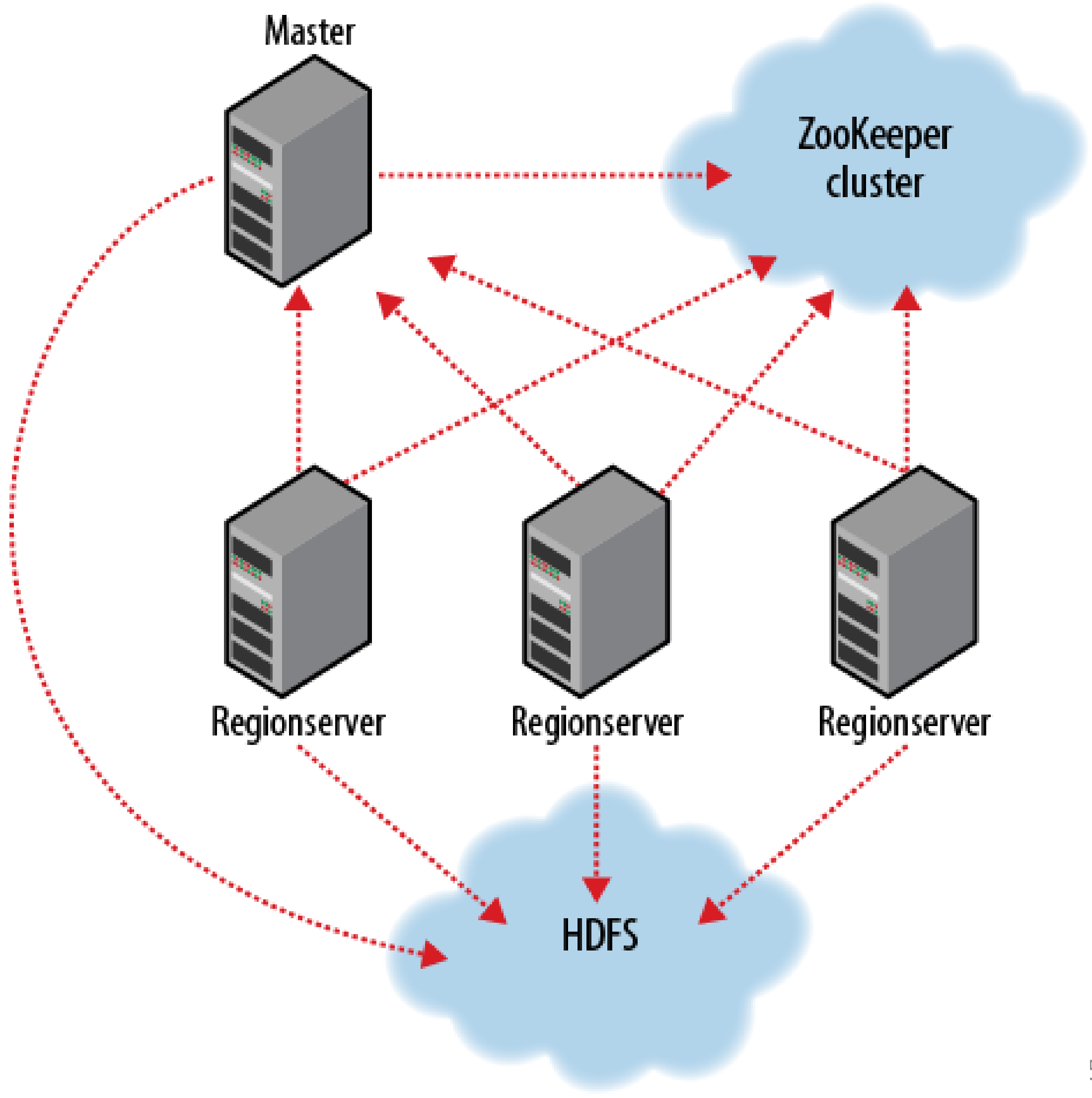


Image source:
Hadoop: The Definitive Guide,
by Tom White

3.5 Accessing HBase: The Big Picture

- The process of accessing HBase tables mirrors the way files are accessed in HDFS.
- A client connects to ZooKeeper 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.
- 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.

3.6 Accessing HBase: Queries

- Data in an HBase table can be accessed by row key (similar to a B-tree index lookup) or by scanning.
  - Access by row key requires the client to specify the desired row key value.
  - 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 to the client and removing the irrelevant ones there, adding the filter to the HBase scan will avoid sending the irrelevant rows to the client. This can significantly reduce cost, saving valuable network bandwidth.

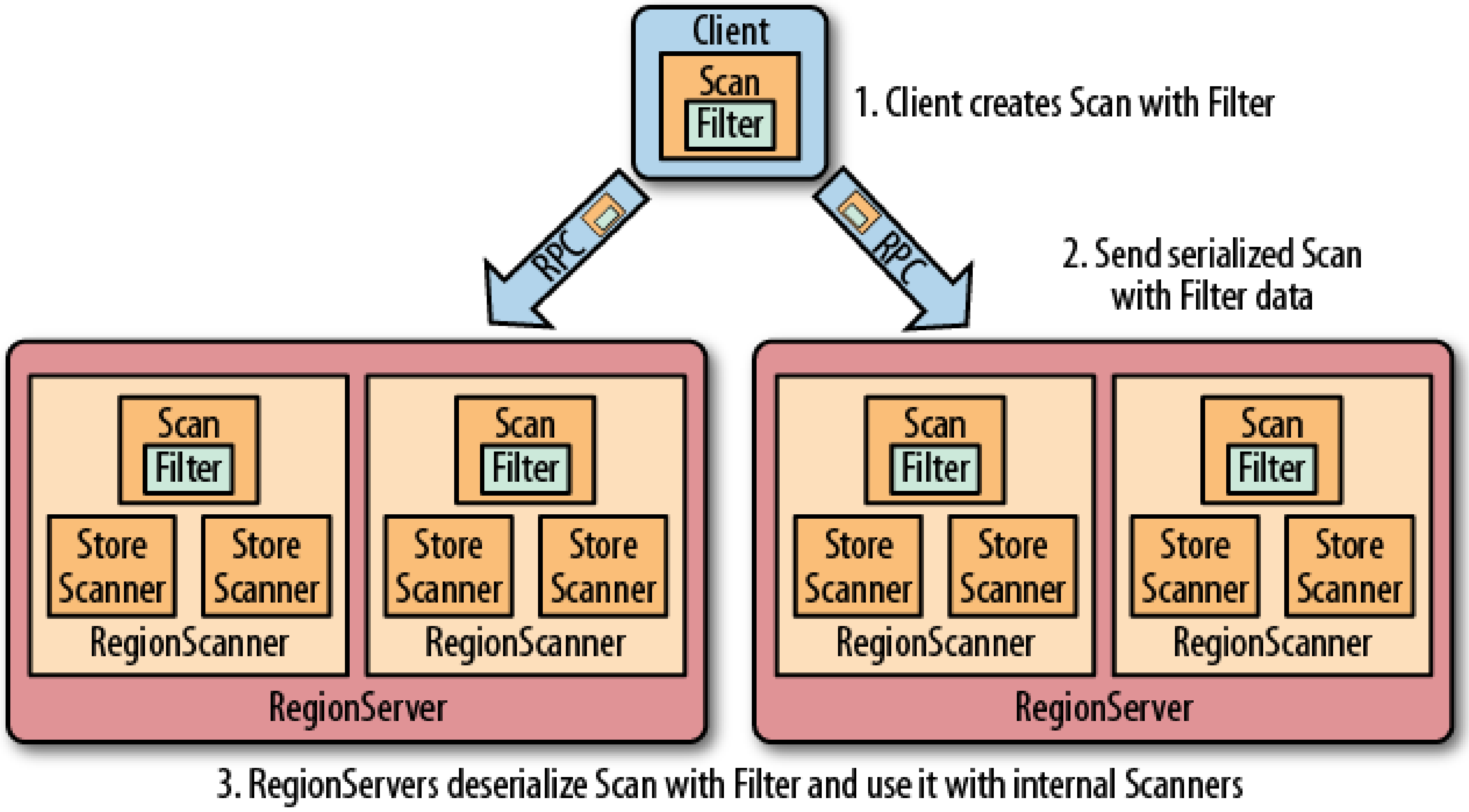
3.6 Accessing HBase: Queries

- Even though HBase can be viewed as a variant of a B-tree on the row key, it does not support additional indexes on any of the other attributes. This limits its ability to support associative access on different columns and column combinations. (Though, projects exist for adding such indexes.)
  - Disadvantage of having no index on data columns: There is no way to quickly home in on rows with certain data values. 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 single row and checking its GPA value. Even if only 1% of the students satisfy the constraint, HBase will still have to read all rows using a scan. If a filter was specified for the scan, HBase would only send the 1% relevant rows 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. [Link: Tell me more.]
  - Advantage of having no index on data columns: Whenever the data changes, the index needs to be updated as well. Having multiple indexes on a table can significantly increase update cost. Hence not having any index enables fast loading of data, which is crucial for big data.

3.6 Accessing HBase: Queries

- 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 PigLatin 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.

Scan with Filter


Client
Scan
Filter
1. Client creates Scan with Filter
RPC
RPC
2. Send serialized Scan with Filter data
Scan
Filter
Store Scanner
Store Scanner
RegionScanner
RegionServer
3. RegionServers deserialize Scan with Filter and use it with internal Scanners

Source: HBase: The Definitive Guide, by Lars George

3.7 Accessing HBase: Clients

- HBase can be accessed from a variety of clients, including:
  - A Java program, e.g., a plain Java MapReduce program.
  - Avro.
  - REST.
  - Thrift.
- In this course, we will only discuss the first option. The following program demonstrates a few basic operations executed from a Java client program.

Code snippet from the book ["Hadoop: The Definitive Guide" by Tom White, 3rd edition], illustrating how tables are created, data is inserted, and then retrieved from an HBase table.

```
public class ExampleClient {
  public static void main(String[] args) throws IOException {
    Configuration config = HBaseConfiguration.create();

    /********** Create table “test” with column family “data” **********/

    HBaseAdmin admin = new HBaseAdmin(config);
    HTableDescriptor htd = new HTableDescriptor("test");
    HColumnDescriptor hcd = new HColumnDescriptor("data");
    htd.addFamily(hcd);
    admin.createTable(htd);

    // Check that the table was properly created
    byte [] tablename = htd.getName();
    HTableDescriptor [] tables = admin.listTables();
    if (tables.length != 1 && Bytes.equals(tablename, tables[0].getName())) {
      throw new IOException("Failed create of table");
    }
```

```
/********** Access the table **********/
HTable table = new HTable(config, tablename);

/********** First, add new row to the table **********/

// Create the new row and assign key “row1” to it
byte [] row1 = Bytes.toBytes("row1");
Put p1 = new Put(row1);

// Put value1 into column data:1. Notice that column family “data” already exists.
// The individual column data:1 did not exist, but will be created automatically.
byte [] databytes = Bytes.toBytes("data");
p1.add(databytes, Bytes.toBytes("1"), Bytes.toBytes("value1"));

// Insert the new row into the table
table.put(p1);

/********** Then read the content of the inserted row **********/

// Access the row using its key “row1”
Get g = new Get(row1);
Result result = table.get(g);
System.out.println("Get: " + result);
```

```
/********** Scan the entire table **********/

  Scan scan = new Scan();
  ResultScanner scanner = table.getScanner(scan);
  try {
    for (Result scannerResult: scanner) {
      System.out.println("Scan: " + scannerResult);
    }
  } finally { scanner.close(); }

  /********** Drop the table **********/
  admin.disableTable(tablename);
  admin.deleteTable(tablename);
 }
}
```

3.8 HBase and MapReduce

- HBase and Hadoop are well integrated. In particular, it is comparably easy to access HBase from a Hadoop program using library org.apache.hadoop.hbase.mapreduce.
- Class TableInputFormat for Mapper input makes sure each Map task receives a single region of the table.
- Class TableOutputFormat allows a Reducer to write directly to an HBase table, instead of HDFS.

3.8.1 HBase and Hadoop Example

- We illustrate the use of HBase from a Hadoop program with an example taken from "Hadoop: The Definitive Guide" by Tom White, 3rd edition.
- 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*.

3.8.2 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.

3.8.3 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? The answer to this question depends on data distribution and workload. The following criteria need to be considered:
  - 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.
  - Uniqueness of rows: The row key should meaningfully distinguish rows from each other. For example, stationid by itself will not make a good row key for the observations table. [Link 1: Why not?] [Link2: How about system time as the row key?]

LINK Text

- Link1: 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 choose a larger number of old row versions to be kept, but this solution would still be clumsy. In the observations table, the unit of interest is an individual observation, not a station. Hence the row identifier needs to contain information that distinguishes individual observations.

- Link2: One might consider using the current system time 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 if different machines can insert data into a table, then one has to make sure that they have access to a consistent global clock. (Otherwise two machines might accidentally choose the same timestamp.) Second, from a design point of view it is considered 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.

3.8.4 Choosing the Key for Observations

- Based on the criteria discussed, we choose (stationid, 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.

3.8.4 Choosing the Key for Observations

- In summary, the row key 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.

3.8.5 Loading Data into HBase

```
public class HBaseTemperatureImporter extends Configured implements Tool {

 // Inner-class for map
 static class HBaseTemperatureMapper<K, V> extends MapReduceBase implements
     Mapper<LongWritable, Text, K, V> {
   private NcdcRecordParser parser = new NcdcRecordParser();
   private HTable table;

   public void map(LongWritable key, Text value,
     OutputCollector<K, V> output, Reporter reporter) throws IOException {
     parser.parse(value.toString());
     if (parser.isValidTemperature()) {
       byte[] rowKey = RowKeyConverter.makeObservationRowKey(parser.getStationId(),
         parser.getObservationDate().getTime());
       Put p = new Put(rowKey);
       p.add(HBaseTemperatureCli.DATA_COLUMNFAMILY,
         HBaseTemperatureCli.AIRTEMP_QUALIFIER,
         Bytes.toBytes(parser.getAirTemperature()));
       table.put(p);
     }
   }
```

3.8.5 Loading Data into HBase

```
  public void configure(JobConf jc) {
    super.configure(jc);
    // Create the HBase table client once up-front and keep it around
    // rather than create on each map invocation.
    try {
      this.table = new HTable(new HBaseConfiguration(jc), "observations");
    } catch (IOException e) {
      throw new RuntimeException("Failed HTable construction", e);
    }
  }

  @Override
  public void close() throws IOException {
    super.close();
    table.close();
  }
}
```

3.8.5 Loading Data into HBase

```
  public int run(String[] args) throws IOException {
    if (args.length != 1) {
      System.err.println("Usage: HBaseTemperatureImporter <input>");
      return -1;
    }
    JobConf jc = new JobConf(getConf(), getClass());
    FileInputFormat.addInputPath(jc, new Path(args[0]));
    jc.setMapperClass(HBaseTemperatureMapper.class);
    jc.setNumReduceTasks(0);
    jc.setOutputFormat(NullOutputFormat.class);
    JobClient.runJob(jc);
    return 0;
  }

  public static void main(String[] args) throws Exception {
    int exitCode = ToolRunner.run(new HBaseConfiguration(),
        new HBaseTemperatureImporter(), args);
    System.exit(exitCode);
  }
}
```

3.8.6 Row Key Generating Code

```
public class RowKeyConverter {

  private static final int STATION_ID_LENGTH = 12;

  /**
   * @return A row key whose format is: <station_id> <reverse_order_epoch>
   */
  public static byte[] makeObservationRowKey(String stationId,
      long observationTime) {
    byte[] row = new byte[STATION_ID_LENGTH + Bytes.SIZEOF_LONG];
    Bytes.putBytes(row, 0, Bytes.toBytes(stationId), 0, STATION_ID_LENGTH);
    long reverseOrderEpoch = Long.MAX_VALUE - observationTime;
    Bytes.putLong(row, STATION_ID_LENGTH, reverseOrderEpoch);
    return row;
  }

}
```

3.8.7 Performance Considerations

- To populate an HBase table with big data, first copy the input file to HDFS. Then use a Map-only job to read the file in parallel. The Map function uses the HTable.put() method to insert data into an HBase table.
  - A Reducer can also use the TableOutputFormat class to emit records into an HBase table.
- To communicate with an HBase table, an HTable object is used. Creating an HTable object is costly, hence this should ideally be done only once per task. In particular, it is a good idea to create the HTable object in the setup() function and close it in the cleanup() function. The Map function should not create or close HTable objects.
- When inserting into an initially empty table, note that this table will not be partitioned, i.e., it will be managed by a single RegionServer, until it grows large enough to be split. Hence initially all Mappers will send their to-be-inserted records to the same RegionServer, potentially creating a bottleneck. As the table grows and gets distributed over multiple RegionServers, insert performance might improve.
  - To distribute load evenly over the different RegionServers, ideally 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.
  - While this might change depending on the HBase version, HTable.put() by default does not use buffering. To enable buffering, one has to disable HTable auto-flush [Link text: HTable.setAutoFlush(false)] and set the size of the write buffer. When using buffering, it is essential to flush the buffer in the end, e.g., by calling HTable.close() in the cleanup() method.

3.8.8 Querying HBase: Lookup with Specific Row Key

```
public Map<String, String> getStationInfo(HTable table, String stationId) throws IOException {
  Get get = new Get(Bytes.toBytes(stationId));
  get.addColumn(INFO_COLUMNFAMILY);
  Result res = table.get(get);
  if (res == null) {
    return null;
  }

  Map<String, String> resultMap = new HashMap<String, String>();
  resultMap.put("name", getValue(res, INFO_COLUMNFAMILY, NAME_QUALIFIER));
  resultMap.put("location", getValue(res, INFO_COLUMNFAMILY, LOCATION_QUALIFIER));
  resultMap.put("description", getValue(res, INFO_COLUMNFAMILY, DESCRIPTION_QUALIFIER));
  return resultMap;
}

private static String getValue(Result res, byte [] cf, byte [] qualifier) {
  byte [] value = res.getValue(cf, qualifier);
  return value == null? '': Bytes.toString(value);
}
```

3.8.9 Querying HBase: Scan

```
public NavigableMap<Long, Integer> getStationObservations(HTable table,
   String stationId, long maxStamp, int maxCount) throws IOException {

  byte[] startRow = RowKeyConverter.makeObservationRowKey(stationId, maxStamp);
  NavigableMap<Long, Integer> resultMap = new TreeMap<Long, Integer>();
  Scan scan = new Scan(startRow);
  scan.addColumn(DATA_COLUMNFAMILY, AIRTEMP_QUALIFIER);
  ResultScanner scanner = table.getScanner(scan);
  Result res = null;
  int count = 0;

  try {
    while ((res = scanner.next()) != null && count++ < maxCount) {
      byte[] row = res.getRow();
      byte[] value = res.getValue(DATA_COLUMNFAMILY, AIRTEMP_QUALIFIER);
      Long stamp = Long.MAX_VALUE -
        Bytes.toLong(row, row.length - Bytes.SIZEOF_LONG, Bytes.SIZEOF_LONG);
      Integer temp = Bytes.toInt(value);
      resultMap.put(stamp, temp);
    }
  } finally {
    scanner.close();
  }

  return resultMap;
}
```

3.8.9 Querying HBase: Scan

```
/**
 * Return the last ten observations.
 */
public NavigableMap<Long, Integer> getStationObservations(HTable table,
    String stationId) throws IOException {
  return getStationObservations(table, stationId, Long.MAX_VALUE, 10);
}
```

4. Hive

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

4.1 System Overview

- Typically Hive is run on a single machine. There it transforms user queries into Hadoop jobs, which can then be executed on a Hadoop installation.
- 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.
- Like Pig, Hive can be run in interactive mode (using the Hive shell) or in batch mode, passing an input file containing a Hive query.

4.2 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.

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

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

Hive LOAD

- 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.

- 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 actually 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. Overall, the lazy approach to data loading makes sense for Hive, because a Hive query is converted to a MapReduce job executed on the input file. Tables and SQL-like query functionality simply provide a convenient way of ultimately writing a Hadoop program to process this file.

4.2.1 Partitions

- Tables can be physically divided into partitions based on the value of designated partition column(s). This feature can significantly reduce query cost for big data.
  - Consider an Internet application that creates a massive log of user activities. Assume a typical log analysis 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. This way the common analysis queries will only access a small number of partitions, reducing query cost.
- 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 now encoded in the directory names, the corresponding attributes will not be part of the table schema any more.
  - Note that partition attributes can still be used in a query as usual. That query can access any number of the partitions.

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

4.2.2 Buckets

- Buckets do not create separate directories like partitions. Instead, they are used for re-ordering data in a table. More precisely, 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;
```

4.3 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.
- In general, 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.

4.3.1 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);
```

4.3.2 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.

4.4 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”. Contrast this to Hive’s LOAD statement, which 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.)

4.4 Hive versus Relational Databases

- 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.

5. Key Points

- 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 appears to magically achieve high data availability and scalability to many nodes, without compromising consistency. In reality, Hadoop limits availability in the sense that it does not modify existing data. Also, Hadoop's way of dealing with machine failures through task re-execution can result in slightly weaker consistency guarantees if a program is non-deterministic. [make "non-deterministic" a link] [Link2: What could go wrong with a non-deterministic program?]
- HBase is a highly scalable key-value store that is well integrated with Hadoop. 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.
- Hive, like Pig, adds a high-level query language on top of Hadoop. It enables data analysts to write their programs in the SQL-like HiveQL language, simplifying many common data analysis tasks for users familiar with relational databases.

LINK Text

- LINK 1: A program is non-deterministic if its output depends on random choices. However, not all MapReduce programs using random number generators will necessarily suffer from data consistency issues.
- Link2: Source: 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."