Database Development and Design (CPT201)

Lecture 12: NoSQL Database and Big Data Storage

Dr. Wei Wang

Department of Computing

Acknowledgement

- Thanks to Prof. Tok Wang LING, National University of Singapore. Some of the slides are based on his talk, "Conceptual Modeling Views of Relational Databases vs Big Data" at DSIT 2019.

Learning Outcomes

- Introduction to Big Data
- Issues and performance problems in RDBMS
- NoSQL and categories
- SQL vs NoSQL
- Big data storage
- MapReduce
- MapReduce vs database

Big Data

- Big data is a broad term for data sets and it can be described by the following 3Vs characteristics:
  - Volume (huge large amount of data: terabytes, petabytes, exabytes)
  - Velocity (speed of data in and out: real-time, streaming)
  - Variety (range of data types and sources, non-relational data such as nested relation, documents, XML data, web data, graph, multimedia, flexible schema or no schema)

Big Data cont'd

- Two more Vs
  - Veracity (correctness and accuracy of information: data quality and reliability)
  - Value (use machine learning, data mining, statistics, visualisation, decision analysis techniques to extract/mine/derive previously unknown insights from data and become actionable knowledge, business value)
- Traditional Relational database management systems (RDBMSs) using SQL are inadequate to handle big data efficiently.

Database Models

- File system
- Hierarchical Model (IMS)
- Network Model (IDMS)
- Relational Model
- Nested Relational Model
- Entity-Relationship Approach
- Object-Oriented(OO) Data Model
- Deductive and Object-Oriented (DOOD)
- Object Relational Data Model
- Semi-structured Data Model (XML)
- RDF and Linked Data
- …

Issues and performance problems in RDBMS

- Normal forms in relational models are to remove redundancies and to reduce updating anomalies. Does data redundancy definitely incur updating anomalies?
  - Example 1: supply (S#, Sname, P#, Pname, price). This supply relation is not even in 2NF. It has redundant information on Sname and Pname, but it does not suffer from updating anomalies as we don't change Sname and Pname of suppliers and parts, resp.
  - Example 2: For Sales transactions, we can add and store the item name, price of item, amount for each item ordered (use the quantity ordered), and total amount of each sales transaction. They are all needed to print the receipts. These are redundant, but they don't incur updating anomalies as we don't change the transactions once they are done.
  - This gives better performance for data analytics! No need to join relations again.

Issues and performance problems in RDBMS cont'd

- Adding redundancy in physical database design may not incur updating anomalies and instead may improve performance significantly, avoid joins.
  - RDBMS cannot handle multi-valued attributes and composite attributes efficiently.
  - Example: nested relation: employee (e#, name, sex, dob, hobby*, qualification(degree, university, year)*)
  - To store employee information, we need 3 normal form relations:
    - employee (e#, name, sex, dob)
    - employee_hobby (e#, hobby)
    - employee_qual (e#, degree, university, year)
  - To get information of an employee, we need to join the 3 relations, very inefficient and very slow and has a lot of redundant information as the joined relation is not in 4NF.

Issues and performance problems in RDBMS cont'd

- RDBMS join operation is very expensive. So it may not be suitable for some applications.
- ACID (Atomicity, Consistency, Isolation, Durability) is to ensure the consistency of data.
  - Two-Phase locking is required. But overhead for enforcing ACID (using locks) is extremely high for managing the locks.
  - Example: large data volume applications which don't modify existing data or at most only add new data, don't require ACID.

NoSQL (not-only SQL)

- Flexible schema or no schema; avoidance of unneeded complexity, e.g. expensive object-relational mapping.
  - NoSQL databases are designed to store data structures that are either simple or more similar to the ones of object-oriented programming languages compared to relational data structures.
- Massive scalability/high throughput
- Relaxed consistency for higher performance and availability
- Quicker/cheaper to set up
- etc

Eventual consistency

- NoSQL softens the ACID properties in relational databases to allow horizontal scalability.
- Desirable properties of horizontally distributed systems:
  - Consistency in a distributed system requires a total order on all operations throughout all nodes of the system
  - A system satisfies availability, if all operations, executed on a node of the system, terminate in a response. Partition cannot be guaranteed in large distributed systems.

Eventual consistency cont'd

- BASE (proposed for Scalable systems) stands for basically available, soft state, eventually consistent.
  - focuses mainly on availability of a system, at the cost of loosening the consistency.
- The eventual consistency property of a BASE system accepts periods where clients might read inconsistent (i.e. out-dated) data.
  - Though, it guarantees that those periods will eventually end.

Eventual consistency cont'd

- Strong consistency: after the update completes, any subsequent access will return the updated value.
- Weak consistency: a number of conditions need to be met before the updated value will be returned.
- Eventual consistency: a consistency model used in distributed computing to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value. (*Vogels, W. (2009. "Eventually consistent". Communications of the ACM. 52: 40.*)

NoSQL categories

- NoSQL databases are categorised according to the way they store the data.
- Four major categories for NoSQL databases:
  - Key-value Stores
  - Wide-Column Stores
  - Document Stores
  - Graph Database

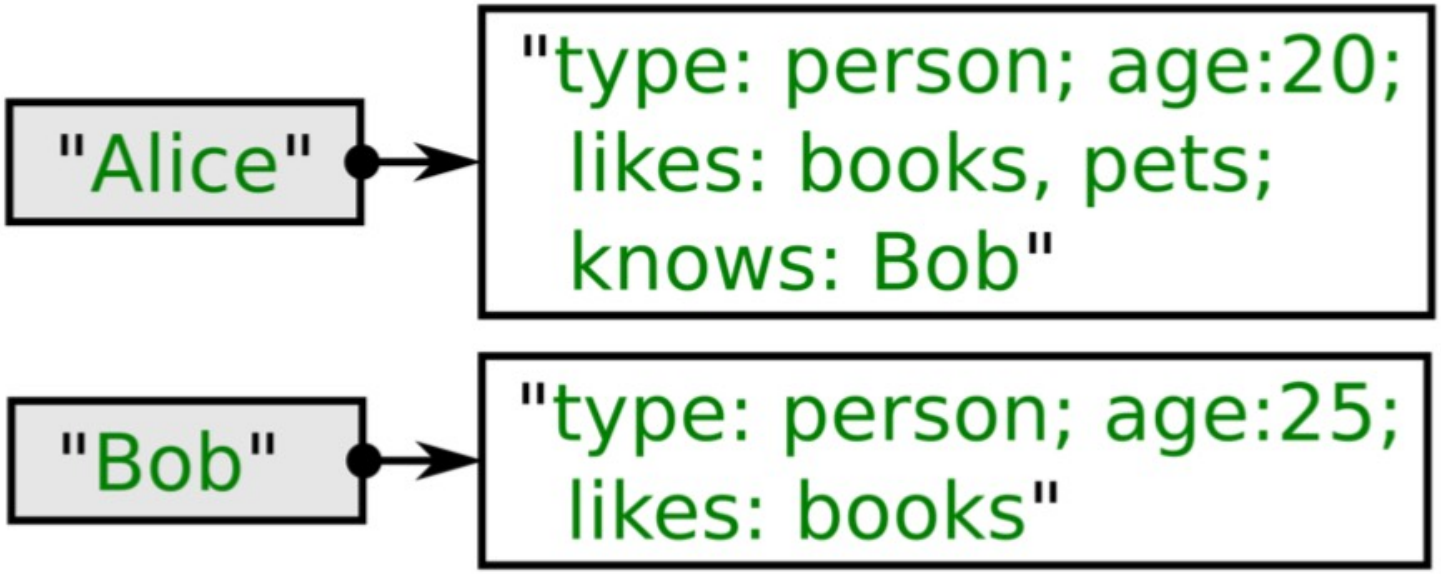
Key-value store

- Key-value storage systems store large numbers (billions or even more) of small (KB-MB) sized records
- Records are partitioned across multiple machines
- Queries are routed by the system to appropriate machines
- Records are also replicated across multiple machines, to ensure availability even if a machine fails
  - Key-value stores ensure that updates are applied to all replicas, to ensure that their values are consistent

Key-value store cont'd

- A key-value store is like associate array
  - data is represented in the form of array["key"] = value or hash table in main memory.
- Each data/object is stored, indexed, and accessed using a key value to access the hash table or array.
- Value is a single opaque collection of objects or data items
  - can be structured, semi-structured, or unstructured. It is just an un-interpreted string of bytes of arbitrary length.
- The meaning of the value in a key-value pair has to be interpreted by programmers.
- No concept of "foreign key", no join
  - data can be horizontally partitioned and distributed

Key-value store cont'd



- Stored value typically can not be interpreted by the storage system.

Key-value store cont'd

- Key-value stores may store
  - Un-interpreted bytes, with an associated key
    - E.g., Amazon S3, Amazon Dynamo
  - Wide-table (can have arbitrarily many attribute names) with associated key
    - Google BigTable, Apache Cassandra, Apache Hbase, Amazon DynamoDB
    - Allows some operations (e.g., filtering) to execute on storage node
  - JSON
    - MongoDB, CouchDB (document model)
- Some key-value stores support multiple versions of data, with timestamps/version numbers

Data representation

- An example of a JSON object is:

```
{
    "ID": "22222",
    "name": {
        "firstname: "Albert",
        "lastname: "Einstein"
        },
    "deptname": "Physics",
    "children": [
        { "firstname": "Hans", "lastname":
    "Einstein" },
        { "firstname": "Eduard", "lastname":
    "Einstein" }
    ]
}
```

Key-value store cont'd

- Typical operations include (but no modification):
  - INSERT new Key-Value pairs (or put)
  - LOOKUP value for a specified key (or get)
  - DELETE key and the value associated with it
- Some systems also support ***range queries*** on key values
- Key value stores are not full database systems
  - Have no/limited support for transactional updates
  - Applications must manage query processing on their own
- Not supporting above features makes it easier to build scalable data storage systems,
  - Also called ***NoSQL*** systems (this is from the textbook, a bit controversial)

Wide-Column Stores

- Data is stored as tables. A table has a row-key and a pre-defined set of column-family columns.
- Each row in the table is uniquely identified by a row-key value.
- Each column family has a large and flexible number of columns (i.e. the No of columns may change from row to row)
  - Each column has a name together with one or more values.
- A column-oriented DBMS stores data tables as column families of data rather than as rows of data.
  - Better for data compression.

Wide-Column Stores cont'd

- A keyspace in a wide-column store contains all the column families (like tables in the relational model), which contain rows, which contain columns.

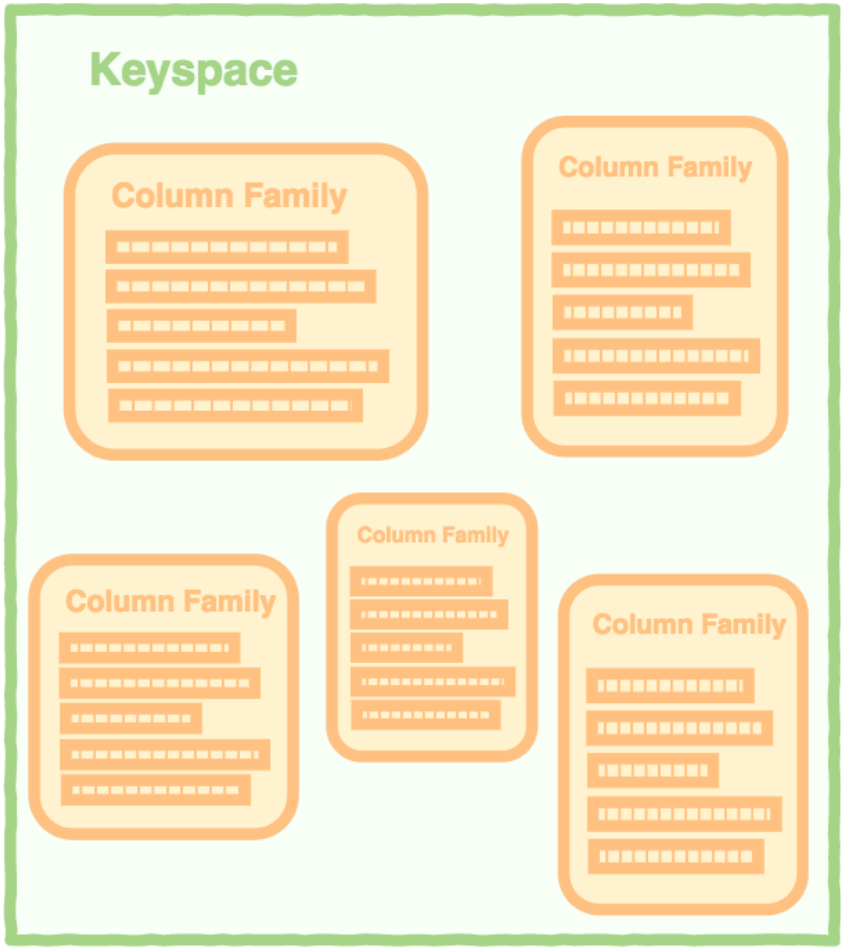


Image source: https://database.guide/what-is-a-column-store-database/

Wide-Column Stores cont'd

- A column family containing 3 rows. Each row contains its own set of columns.

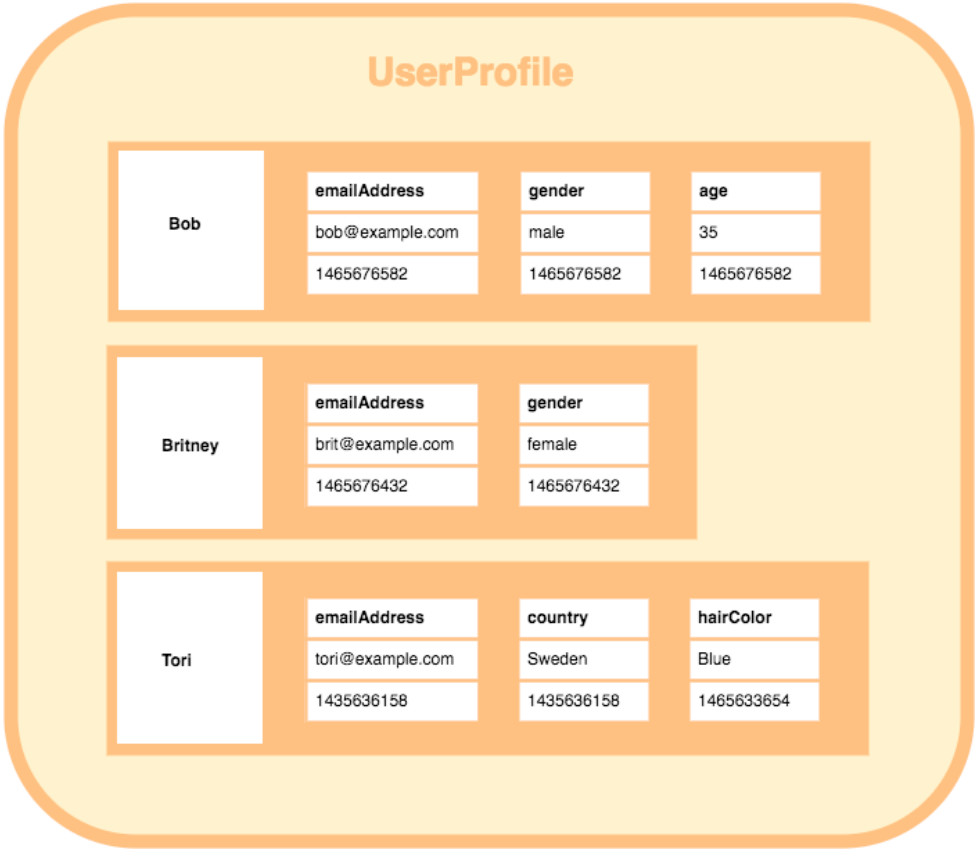


Image source: https://database.guide/what-is-a-column-store-database/

Wide-Column Stores cont'd

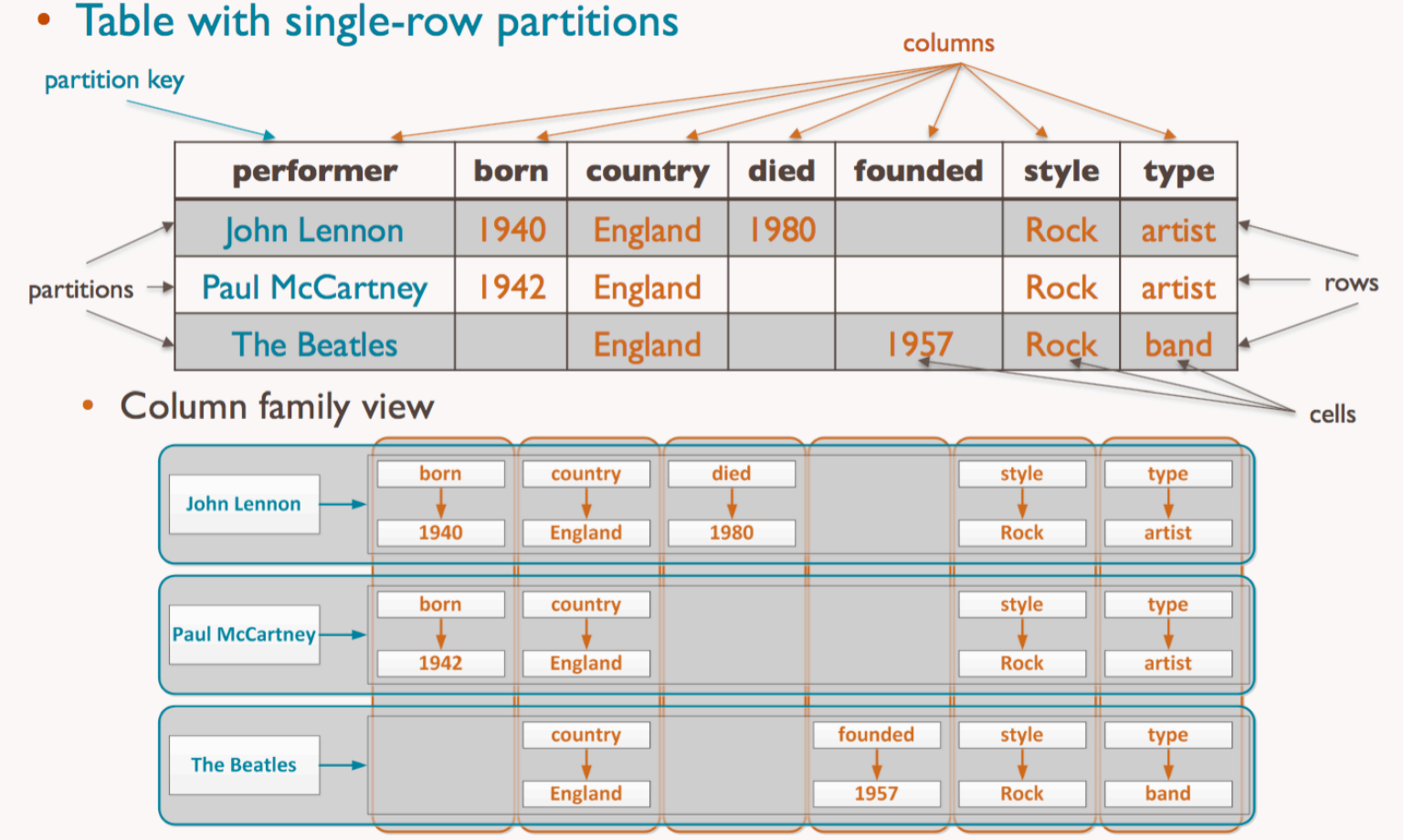


Image source: https://studio3t.com/knowledge-base/articles/nosql-database-types/#wide-column-store

Google's BigTable

- A sparse, distributed, persistent multi-dimensional sorted map.
- Used by several Google applications such as web indexing, MapReduce, Google Maps, Google Earth, YouTube, Gmail, etc.
- The map is indexed by a row key, column key, and a timestamp; each value in the map is an un-interpreted array of bytes.
  - For webpage, the row key value is a reversed url.
  - BigTable maintains data in lexicographic order by row key.
  - So webpages in the same domain are grouped together into contiguous rows.

BigTable: Storing Web Pages

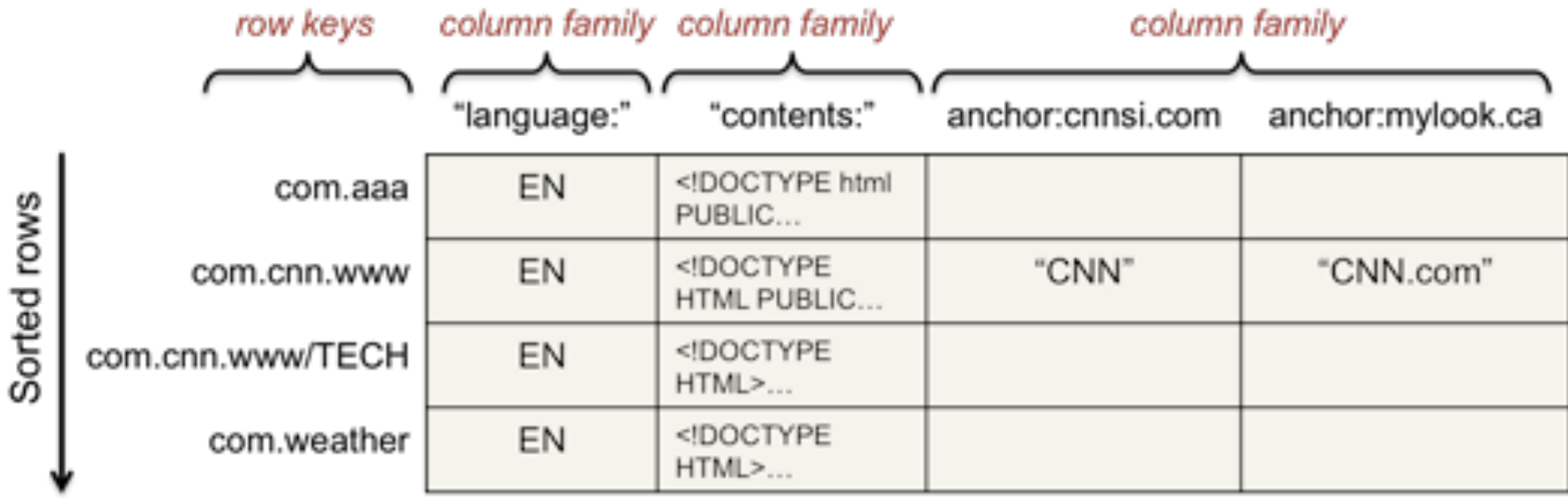


Image source: https://www.cs.rutgers.edu/~pxk/417/notes/content/bigtable.html

Graph Database

- Best suited for representing data with a large number of interconnections
  - especially when information about those interconnections is at least as important as the represented data
  - for example, social relations or geographic data.
- Graph databases allow for queries on the graph structure, e.g., relations between nodes or shortest paths.
- Examples:
  - RDF graph and linked data
  - Google knowledge graph

Document Stores

- Schema languages are not powerful to express Object- Relationship-Attribute semantics in ER model.
- Data is stored in so-called documents, i.e. arbitrary data in some (semi-)structured format.
  - JSON
  - BSON
  - XML
- Data format is typically fixed, but the structure is flexible.
  - in a JSON-based document store, documents with completely different sets of attributes can be stored together.

SQL VS NoSQL

Characteristics	SQL	NoSQL
Schema	Yes. Schema must be predefined and fixed. Schema evolution is difficult.	Schema is optional, may not be predefined. Schema can be semi-structured or un-structured.
Data type	Flat relations. Fixed length field/record for each relation defined.	Tree/graph structured data. Variable length, multi-valued attribute (repeating, nested tree), can add new tag names any time.

SQL VS NoSQL cont'd

Characteristics	SQL	NoSQL
Data persistence	Databases are stored on disk drive, data persistence. Very slow to access as compared to in-memory stores.	In-memory, use pointer and hashing. Very fast to access. Need to convert data in memory from/to disk drive to archive data persistence.
OLTP or OLAP	OLTP, mission critical online transaction applications. Only keep current database state. If historical data is required then need to use temporal database with time period to store.	OLAP for data warehouse and data analytics. Keep the historical data, time/date dimension is a must for meaningful data analysis. (Seldom mention time attribute in key-value store and data graph.)

SQL VS NoSQL cont'd

Characteristics	SQL	NoSQL
Language to access the data	Standard DBMS declarative query language SQL. Operate on a set of tuples at a time basis.	Different imperative programming languages. Write programs (e.g. MapReduce, JSON programs with API's).
Update to data	Frequent update to database (transaction)	mainly have new data, no or seldom updates (deletion and addition)
Query optimisation	Query optimiser of RDBMS	Optimisation done by programmers for each of their programs.
DBMS	RDBMS	Not really, just as data stores

SQL VS NoSQL cont'd

Characteristics	SQL	NoSQL
Answers for queries	Return accurate/precise query answers	Return "best guess" or "an opinion" answers – similar to data mining and IR answers
ACID	Yes, consistency is the most important issue for OLTP applications	Emphasis on speed performance, use eventual consistent. If no updates, then ACID is not required.
Join operation	Yes, queries may involve many joins, can be very slow.	Avoid or no join. Use redundant data to speed up processing
Ad hoc user queries	Write SQL programs or RDB keyword query search	Need programmers to write programs.
Distributed & parallel processing	Limited	Yes. Data can be partitioned horizontally and/or vertically and distributed to nodes, and process the partitioned data in parallel. Efficient for such applications.

Big Data Storage Systems

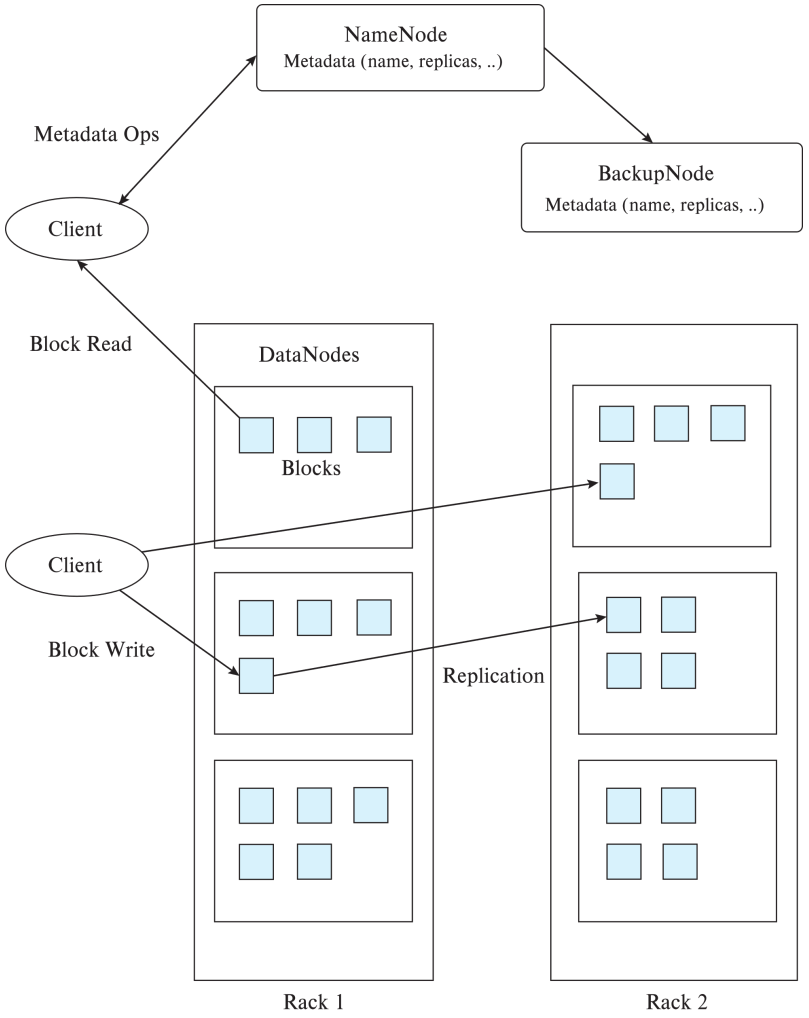
- Distributed file systems
- Sharding across multiple databases
- Key-value storage systems
  - The textbook categorises all the NoSQL storage systems as key-value stores
- Parallel and distributed databases

Distributed File Systems

- A distributed file system stores data across a large collection of machines, but provides single file-system view
- Highly scalable distributed file system for large data-intensive applications.
  - E.g., 10K nodes, 100 million files, 10 PB
- Provides redundant storage of massive amount of data on cheap and unreliable computers
  - Files are replicated to handle hardware failure
  - Detect failures and recovers from them
- Examples:
  - Google File System (GFS)
  - Hadoop File System (HDFS)

Hadoop File System Architecture

- Single Namespace for entire cluster
- Files are broken up into blocks
  - Typically 64 MB block size
  - Each block replicated on multiple DataNodes
- Client
  - Finds location of blocks from NameNode
  - Accesses data directly from DataNode



Hadoop Distributed File System

- **NameNode**
  - Maps a filename to list of Block IDs
  - Maps each Block ID to DataNodes containing a replica of the block
- **DataNode**: Maps a Block ID to a physical location on disk
- Data Coherency
  - Write-once-read-many access model
  - Client can only append to existing files
- Distributed file systems are good for millions of large files
  - But have very high overheads and poor performance with billions of smaller tuples

Sharding

- **Sharding**: partition data across multiple databases
- Partitioning usually done on some ***partitioning attributes*** (also known as ***partitioning keys*** or ***shard keys*** e.g. user ID
  - records with key values from 1 to 100,000 on database 1, records with key values from 100,001 to 200,000 on database 2, etc.
- Application must track which records are on which database and send queries/updates to that database
- Positives: scales well, easy to implement
- Drawbacks:
  - Not transparent: application has to deal with routing of queries, queries that span multiple databases
  - If a database is overloaded, moving part of its load out is not easy
  - Chance of failure more with more databases
    - need to keep replicas to ensure availability, which is more work for application

Parallel Databases and Data Stores

- Supporting scalable data access
  - Approach 1: memcache or other caching mechanisms at application servers, to reduce database access
    - Limited in scalability
  - Approach 2: Partition ("shard") data across multiple separate database servers
  - Approach 3: Use existing parallel databases
    - Historically: parallel databases that can scale to large number of machines were designed for decision support not OLTP
  - Approach 4: Massively Parallel Key-Value Data Store
    - Partitioning, high availability etc. completely transparent to application
- Sharding systems and key-value stores don't support many relational features, such as joins, integrity constraints, etc., across partitions.

Parallel and Distributed Databases

- Parallel databases run multiple machines (cluser)
  - Developed in 1980s, well before Big Data
- Parallel databases were designed for smaller scale (10s to 100s of machines)
  - Did not provide easy scalability
- Replication used to ensure data availability despite machine failure
  - But typically restart query in event of failure
    - Restarts may be frequent at very large scale
    - Map-reduce systems (coming up next) can continue query execution, working around failures

The MapReduce Paradigm

- Platform for reliable, scalable parallel computing
- Abstracts issues of distributed and parallel environment from programmer
  - Programmer provides core logic (via map() and reduce() functions)
  - System takes care of parallelisation of computation, coordination, etc.
- Paradigm dates back many decades
  - But very large scale implementations running on clusters with 10^3 to 10^4 machines are more recent
  - Google Map Reduce, Hadoop, ..
- Data storage/access typically done using distributed file systems or key-value stores

MapReduce Example: Word Count

- Consider the problem of counting the number of occurrences of each word in a large collection of documents
- Solution:
  - Divide documents among workers
  - Each worker parses document to find all words, map function outputs (word, count) pairs
  - Partition (word, count) pairs across workers based on word
  - For each word at a worker, reduce function locally add up counts
- Input: "One a penny, two a penny, hot cross buns."
  - Records output by the map() function would be
    - ("One", 1), ("a", 1), ("penny", 1),("two", 1), ("a", 1), ("penny", 1), ("hot", 1), ("cross", 1), ("buns", 1).
  - Records output by reduce function would be
    - ("One", 1), ("a", 2), ("penny", 2), ("two", 1), ("hot", 1), ("cross", 1), ("buns", 1)

Pseudo-code of Word Count

```
map(String record):
    for each word in record
      emit(word, 1);

// First attribute of emit above is called reduce key
// In effect, group by is performed on reduce key to create a
// list of values (all 1's in above code).  This requires shuffle
  step across machines.
// The reduce function is called on list of values in each group

reduce(String key, List value_list):
    String word = key
    int count = 0;
    for each value in value_list:
      count = count + value
    Output(word, count);
```

MapReduce Programming Model

- Inspired from map and reduce operations commonly used in functional programming languages like Lisp.
- Input: a set of key/value pairs
- User supplies two functions:
  - **map**(k,v) → list(k1,v1)
  - **reduce**(k1, list(v1)) → v2
- (k1,v1) is an intermediate key/value pair
- Output is the set of (k1,v2) pairs
- For our example, assume that system
  - Breaks up files into lines, and
  - Calls map function with value of each line
    - Key is the line number

MapReduce Example: Log Processing

- Given log file in following format:

2013/02/21 10:31:22.00EST /slide-dir/11.ppt
2013/02/21 10:43:12.00EST /slide-dir/12.ppt
2013/02/22 18:26:45.00EST /slide-dir/13.ppt
2013/02/22 20:53:29.00EST /slide-dir/12.ppt
...

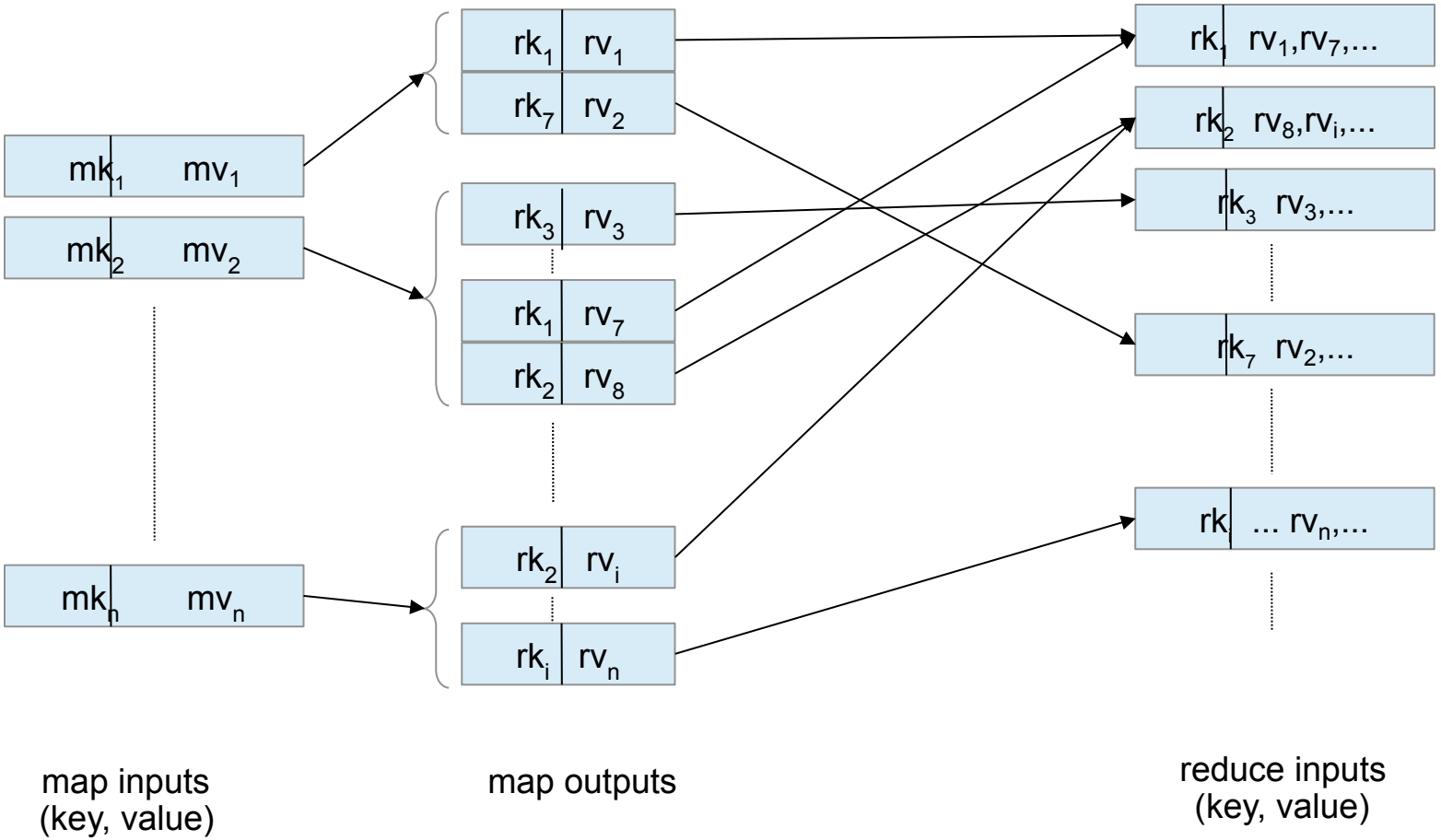
- Goal: find how many times each of the files in the *slide-dir* directory was accessed between 2013/01/01 and 2013/01/31.
- Options:
  - Sequential program too slow on massive datasets
  - Load into database expensive, direct operation on log files cheaper
  - Building parallel program is possible, but very laborious
  - Map-reduce paradigm

MapReduce Example: Log Processing cont'd

```
map(String key, String record) {
    String attribute[3];
    //break up record into tokens (based on space character), and store
     the tokens in array attributes
    String date = attribute[0];
    String time = attribute[1];
    String filename = attribute[2];
    if (date between 2013/01/01 and 2013/01/31
            and filename starts with "/slide-dir/")
        emit(filename, 1).
}

reduce(String key, List recordlist) {
    String filename = key;
    int count = 0;
    For each record in recordlist
       count = count + 1
    output(filename, count)
}
```

Flow of keys and values in a map-reduce task



Hadoop MapReduce

- Google pioneered map-reduce implementations that could run on thousands of machines (nodes), and transparently handle failures of machines
- Hadoop is a widely used open source implementation of Map Reduce written in Java
  - Map and reduce functions can be written in several different languages.
- Input and output to map reduce systems such as Hadoop must be done in parallel
  - Google used GFS distributed file system
  - Hadoop uses Hadoop Distributed File System (HDFS)
  - Input files can be in several formats
    - Text/CSV
    - compressed representation such as Avro, ORC and Parquet
  - Hadoop also supports key-value stores such as Hbase, Cassandra, MongoDB, etc.

Types in Hadoop

- Generic Mapper and Reducer interfaces both take four type arguments, that specify the types of the
  - input key, input value, output key and output value
- Map class in next slide implements the Mapper interface
  - Map input key is of type LongWritable, i.e. a long integer
  - Map input value which is (all or part of) a document, is of type Text.
  - Map output key is of type Text, since the key is a word,
  - Map output value is of type IntWritable, which is an integer value.

Hadoop Code in Java: Map Function

```
public static class Map extends Mapper<LongWritable, Text, Text,
    IntWritable>
{
   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();
   public void map(LongWritable key, Text value, Context context)
          throws IOException, InterruptedException
   {
       String line = value.toString();
       StringTokenizer tokenizer = new StringTokenizer(line);
       while (tokenizer.hasMoreTokens()) {
             word.set(tokenizer.nextToken());
             context.write(word, one);
       }
   }
}
```

Hadoop Code in Java: Reduce Function

```
public static class Reduce extends Reducer<Text, IntWritable,
    Text, IntWritable> {
   public void reduce(Text key, Iterable<IntWritable> values,
         Context context)  throws IOException,
    InterruptedException
   {
       int sum = 0;
       for (IntWritable val : values) {
            sum += val.get();
       }
       context.write(key, new IntWritable(sum));
   }
}
```

Hadoop Code in Java: Overall Program

```
public class WordCount {
    public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();
        Job job = new Job(conf, "wordcount");
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);
        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.waitForCompletion(true);
    }
}
```

Map Reduce vs. Databases

- Map Reduce is widely used for parallel processing
  - Google, Yahoo, and 100's of other companies
  - Example uses: compute PageRank, build keyword indices, do data analysis of web click logs, ….
  - Allows procedural code in map and reduce functions
  - Allows data of any type
- Many real-world uses of MapReduce cannot be expressed in SQL
- But many computations are much easier to express in SQL
  - Map Reduce is cumbersome for writing simple queries
- Relational operations (select, project, join, aggregation, etc.) can be expressed using Map Reduce
- SQL queries can be translated into Map Reduce infrastructure for execution
  - Apache Hive SQL, Apache Pig Latin, Microsoft SCOPE

End of Lecture

- Summary
  - Introduction to Big Data
  - Issues and performance problems in RDBMS
  - NoSQL and categories
  - SQL vs NoSQL
  - Big data storage
  - MapReduce
  - MapReduce vs database
- Reading
  - Textbook 7th edition, chapters 10.1, 10.2, 10.3
  - Zollmann, Johannes. "Nosql databases." Retrieved from Software Engineering Research Group: http://www.webcitation.org/6hA9zoqRd, (2012).
  - Strauch, Christof, Ultra-Large Scale Sites, and Walter Kriha. "NoSQL databases." Lecture Notes, Stuttgart Media University 20 (2011).
  - BigTable, https://www.cs.rutgers.edu/~pxk/417/notes/content/bigtable.html