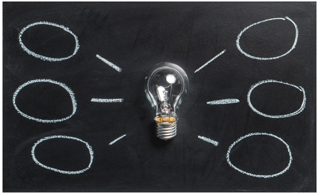
Julian Kunkel

Summary for the Lecture High-Performance Data Analytics



Outline

1 Introduction

2 Data Models

3 Databases

4 Distributed Storage and Processing with Hadoop

5 Big Data SQL using Hive

6 Dataflow Computation

7 Columnar Access

8 Document Storage

Examination Preparation

Importance of Learning Objectives

- The exam will assess the learning objectives
- Study the learning objectives of each slide deck
- Study the learning objectives of the overall lecture

Importance of Exercises

- Exercises are often a good hint regarding the examination
- Won’t ask too specific implementation questions
- But may ask something like "sketch a pig program that does X"...

This Lecture

- Aims to summarize some important points
- Still: need to check all slide decks

Outline

1 Introduction

2 Data Models

3 Databases

4 Distributed Storage and Processing with Hadoop

5 Big Data SQL using Hive

6 Dataflow Computation

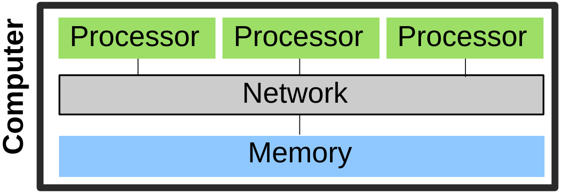
7 Columnar Access

8 Document Storage

Parallel Architectures

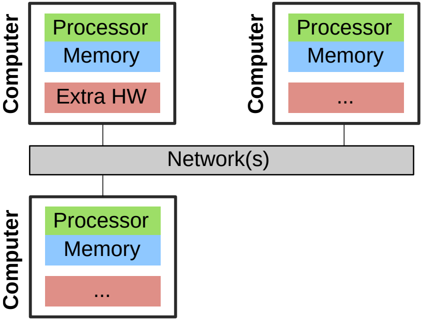
In practice, systems are a mix of two paradigms:

Shared memory



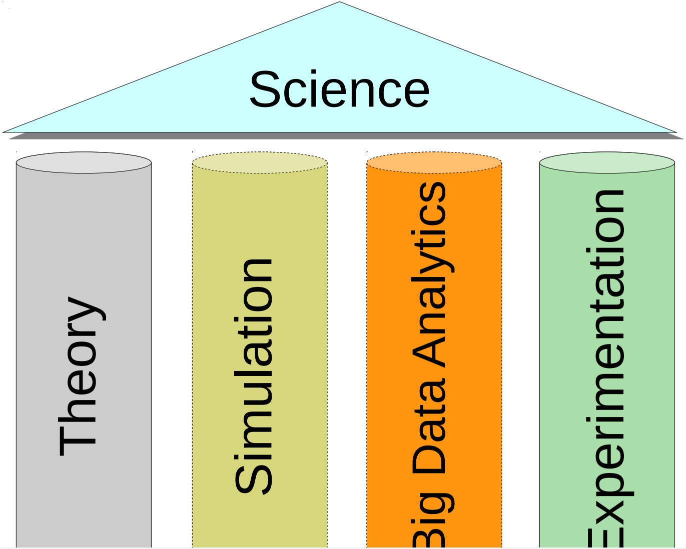
- Processors can access a joint memory
  - Enables communication/coordination
- Cannot be scaled up to any size
- Very expensive to build one big system

Distributed memory systems (again!)



- Processor can only see own memory
- Performance of the network is key

Pillars of Science: **Modern Perspective**



Relation of the Scientific Method to D/P/S Computing

Simulation models real systems to gain new insight

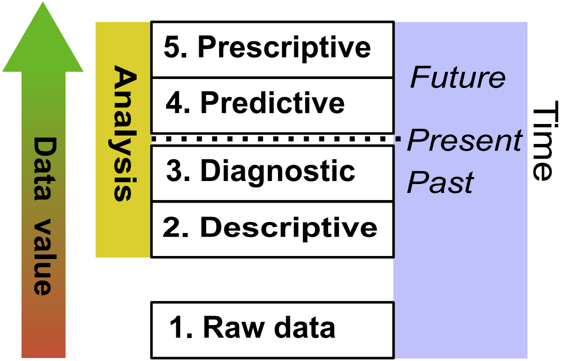
- Instrument to make observations, e.g., high-resolution and fast timescale
- Typically used to validate/refine theories, identify new phenomen
- Classical computational science: hard facts (based on models)
- The frontier of science needs massive computing resources on supercomputers
- Data-intensive sciences like climate imposes challenges to data handling, too

Big Data Analytics extracts insight from data

- Provides a data pool to identify/mine new insight and to validate theories
- In business often approximate insight is enough (a small advantage)
- Distributed and parallel systems are needed to manage and analyze the data
- Gained knowledge is often made available as part of the cloud (for money)...

Abstraction Levels of Analytics and the Value of Data

5. Prescriptive analytics
   - "What should we do and why?"
4. Predictive analytics
   - "What will happen?"
3. Diagnostic analytics
   - "What went wrong?"
   - "Why did this happen"
2. Descriptive analytics[1]
   - "What happened?"
1. Raw (observed) data



Relation to Computational Science

- These analysis steps are still done just by running computational experiments
- Also the output of the simulation must be analyzed

[1] Decriptive and diagnostic analysis are like forensics

BigData Challenges & Characteristics

Dealing with large data is challenging in Big Data Analytics but also in Computational Science

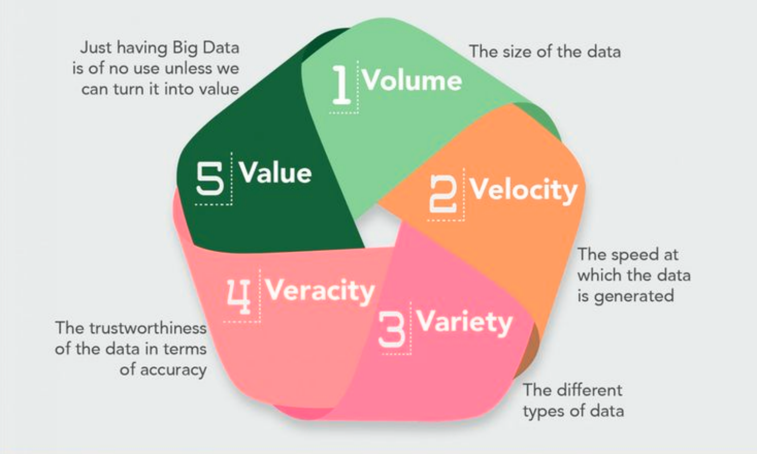

Just having Big Data is of no use unless we can turn it into value
1 Volume
The size of the data
5 Value
2 Velocity
The speed at which the data is generated
4 Veracity
3 Variety
The trustworthiness of the data in terms of accuracy
The different types of data

Figure: Source: MarianVesper (Forrester Big Data Webinar. Holger Kisker, Martha Bennet. Big Data: Gold Rush Or Illusion?)

Volume: The size of the Data

What is Big Data

Terrabytes to 10s of petabytes

What is not Big Data

A few gigabytes

Examples

- Wikipedia corpus with history ca. 10 TByte
- Wikimedia commons ca. 23 TByte
- Google search index ca. 46 Gigawebpages[2]
- YouTube per year 76 PByte (2012[3])

[2] http://www.worldwidewebsize.com/

[3] https://sumanrs.wordpress.com/2012/04/14/youtube-yearly-costs-for-storagenetworking-estimate/

Outline

1 Introduction

2 Data Models

3 Databases

4 Distributed Storage and Processing with Hadoop

5 Big Data SQL using Hive

6 Dataflow Computation

7 Columnar Access

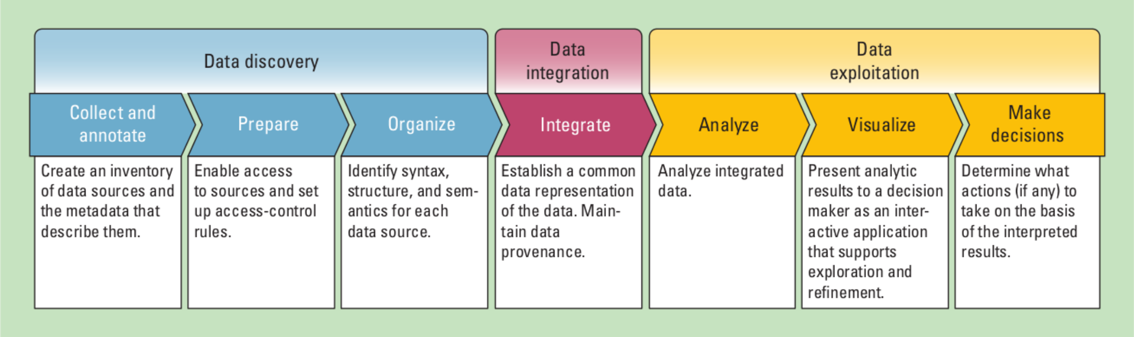
8 Document Storage

Terminology for Managing Data [1, 10]

- **Data governance**: “*control that ensures that the data entry ... meets precise standards such as business rule, a data definition and data integrity constraints in the data model*” [10]
  - Think about reasons that invalidate data that lead to catastrophic results...
  - Example: missinterpretation of data value "NaN" as "0" in a survey
- **Data provenance**: the documentation of input, transformations of data and involved systems to support analysis, tracing and reproducibility
  - e.g., Input (file.csv) ⇒Calculate means via x.py (result: means.csv) ⇒Create diagrams via d.py (result fig1.pdf)
- **Data-lineage** (Datenherkunft): forensics; allows to identify the source data used to generate data products (part of data provenance)
  - e.g., fig1.pdf has been produced from ... using Z...
  - I’m able to reproduce results and track errors from the product
- **Service level agreements** (SLAs): contract defining quality, e.g., performance/reliability & responsibilities between service user/provider

Data Analysis Workflow

The traditional approach proceeds in phases:



Create an inventory of data sources and the metadata that describe them.

Enable access to sources and set up access-control rules.

Identify syntax, structure, and semantics for each data source.

Establish a common data representation of the data. Maintain data provenance.

Analyze integrated data.

Present analytic results to a decision maker as an interactive application that supports exploration and refinement.

Determine what actions (if any) to take on the basis of the interpreted results.

Figure: Source: Gilbert Miller, Peter Mork From Data to Decisions: A Value Chain for Big Data.

- Analysis tools: machine learning, statistics, interactive visualization
- Limitation: Interactivity by browsing through prepared results
- Indirect feedback between visualization and analysis

Alternative Processing Technology

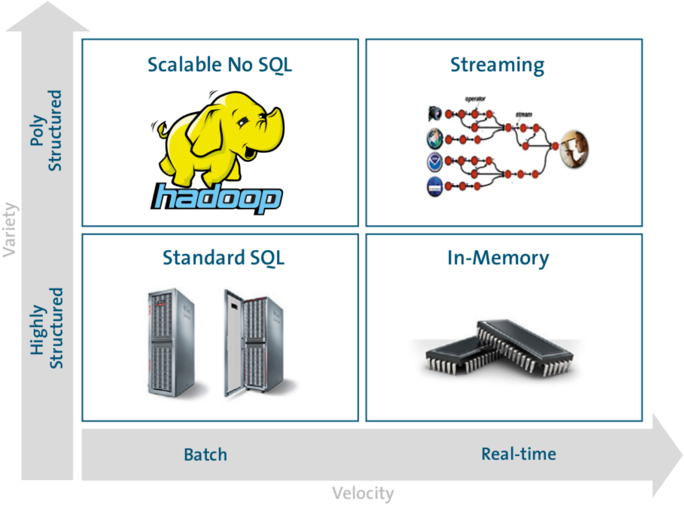
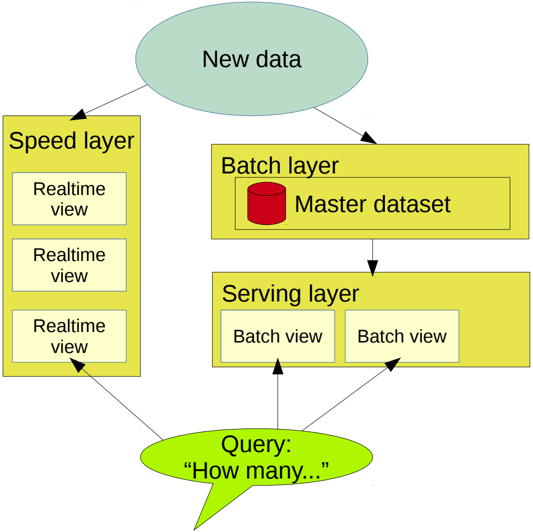


Figure: Source: Forrester Webinar. Big Data: Gold Rush Or Illusion? [4]

The Lambda Architecture [11]



- **Goal**: Interactive Processing
- Batch layer pre-processes data
  - Master dataset is immutable/never changed
  - Operations are periodically performed
- Serving layer offers performance optimized views
- Speed layer serves deltas of batch and recent activities, may approximate results
- Robust: Errors/inaccuracies of realtime views are corrected in batch view

Data Models[4]and their Instances [12]

- A data model describes how information is organized in a system
  - It is a tool to specify, access and process information
  - A model provide operations for accessing and manipulating data that follow certain semantics
  - Typical information is some kind of entity (virtual object) (e.g., car)
- **Logical model**: abstraction expressing objects and operations
- **Physical model**: maps logical structures onto hardware resources (e.g., files, bytes)

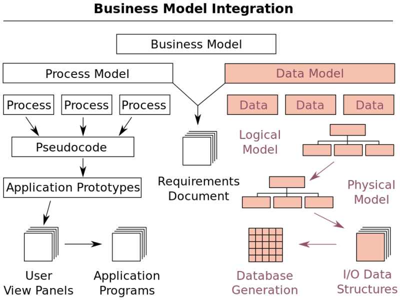


Figure: Source: [12]

- DM theory: Formal methods for describing data models with tool support
- Applying theory creates a **data model instance** for a specific application

[1]: The term is often used ambivalently for a data (meta) model concept/theory or an instance

Operations

- Operations define how you can interact with the data
  - Minimal: Need to somehow store and retrieve data
  - Users may want to search for data, update existing data
- May want to offload some operations to the server side: active storage
  - Reduce data, e.g., compute mean/sum
  - Conditional updates

Typical Operations

- POSIX: create, open, write (anywhere), read (anywhere)
  - Does not distinguis between write and update
- CRUD: Create, Read, Update, Delete
- Amazon S3: Put (Overwrite), Get (Partially), Delete

Relational Model [10]

- Database model based on first-order predicate logic
  - Theoretic foundations: relational algebra and relational calculus
- Data is represented as tuples
  - In its original style, it does not support collections
- Relation/Table: groups tuples with similar semantics
  - Table consists of rows and named columns (attributes)
  - No (identical) duplicate of a row allowed
- Schema: specify structure of tables
  - Datatypes (domain of attributes)
  - Consistency via constraints
  - Organization and optimizations

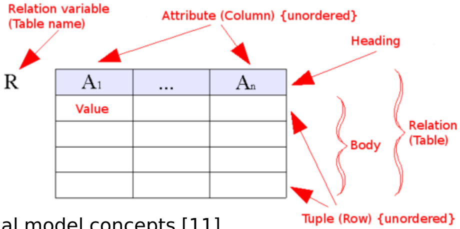


Figure: Source: Relational model concepts [11]

Columnar Model

- Data is stored in rows and columns (similar to tables)
- A column is a tuple (name, value and timestamp)
- Each row can contain different columns
  - Columns can store complex objects, e.g., collections
- Wide columnar model: very sparse table of 100k+ columns
- Example technology: HBase, Cassandra, Accumulo

Row/Column:	student name	matrikel	lectures	lecture name
1	"Max Mustermann"	4711	[3]	-
2	"Nina Musterfrau"	4712	[3,4]	-
3	-	-	-	"Big Data Analytics"
4	-	-	-	"Hochleistungsrechnen"

Table: Example columnar model for the students, each value has its own timestamp (not shown). Note that lectures and students should be modeled with two tables

Key-Value Store

- Data is stored as value and addressed by a key
- The value can be complex objects, e.g., JSON or collections
- Keys can be forged to simplify lookup (evtl. tables with names)
- Example technology: CouchDB, BerkeleyDB, Memcached, BigTable

Key	Value
stud/4711	<name>Max Mustermann</name><attended><id>1</id></attended>
stud/4712	<name>Nina Musterfrau</name><attended><id>1</id><id>2</id></attended>
lec/1	<name>Big Data Analytics</name>
lec/2	<name>Hochleistungsrechnen</name>

Table: Example key-value model for the students with embedded XML

Document Model

- Collection of documents
- Documents contain semi-structured data (JSON, XML)
- Addressing to lookup documents are implementation specific
  - e.g., bucket/document key, (sub) collections, hierarchical namespace
- References between documents are possible
- Example technology: MongoDB, Couchbase, DocumentDB

```
<students>
  <student><name>Max Mustermann</name><matrikel>4711</matrikel>
    <lecturesAttended><id>1</id></lecturesAttended>
  </student>
  <student><name>Nina Musterfrau</name><matrikel>4712</matrikel>
    <lecturesAttended><id>1</id><id>2</id></lecturesAttended>
  </student>
</students>
```

Table: Example XML document storing students. Using a bucket/key namespace, the document could be addressed with key: “uni/stud” in the bucket “app1”

Graph

- Entities are stored as nodes and relations as edges in the graph
- Properties/Attributes provide additional information as key/value
- Example technology: Neo4J, InfiniteGraph

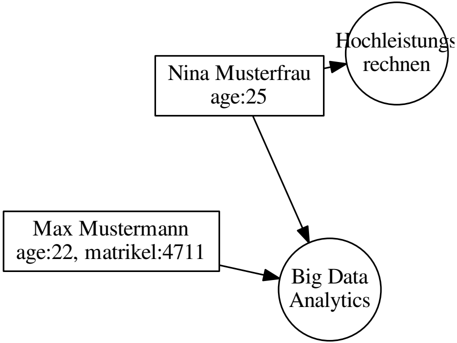


Figure: Graph representing the students (attributes are not shown)

Fact-Based Model [11][6]

- Store raw data as timestamped atomic facts aka log files of change/current status
- Never delete true facts: Immutable data
- Make individual facts unique to prevent duplicates

Example: social web page

- Record all changes to user profiles as facts
- Benefits
  - Allows reconstruction of the profile state at any time
  - Can be queried at any time[5]

Example: purchases

- Record each item purchase as facts together with location, time, ...

[5] If the profile is changed recently, the query may return an old state.

[6] Note that the definitions in the data warehousing (OLAP) and big data [11] domains are slightly different

From Big Data to the Data Lake

- With cheap storage costs, people promote the concept of the data lake
- Combines data from many sources (data silos) and of any type and model
- Allows for conducting future analysis and not miss any opportunity

Attributes of the data lake

- Collect everything: all time all data: raw sources and processed data
  - Decide during analysis which data is important, e.g., no “schema“ until read
- Dive in anywhere: enable users across multiple business units to
  - Refine, explore and enrich data on their terms
- Flexible access: shared infrastructure supports various patterns
  - Batch, interactive, online, search

http://hortonworks.com/blog/enterprise-hadoop-journey-data-lake/

Outline

1. Introduction
2. Data Models
3. **Databases**
4. Distributed Storage and Processing with Hadoop
5. Big Data SQL using Hive
6. Dataflow Computation
7. Columnar Access
8. Document Storage

Entity Relationship Diagrams

- Illustrate the relational model and partly the database schema
- Elements: Entity, relation, attribute
  - Additional information about them, e.g., cardinality, data types

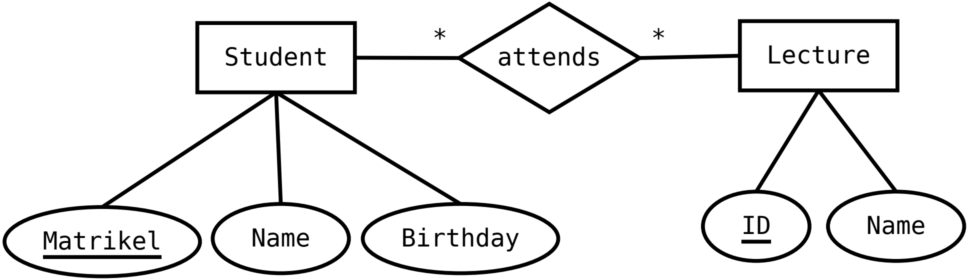


Figure: A student/lecture example in modified Chen notation
* is the cardinality and means any number is fine

Queries [20]

- A query retrieves/computes a (sub)table from tables
  - It does **not change/mutate** any content of existing tables
- Statement: SELECT $< column1 >, < column2 >, ...$
- Subqueries: nesting of queries is possible to create temporary tables

Supported clauses

- FROM: specify the table(s) to retrieve data
- WHERE: filter rows returned
- GROUP BY: group rows together that match conditions
- HAVING: filters grouped rows
- ORDER BY: sort the rows

```sql
SELECT Matrikel, Name FROM students WHERE Birthday='22.04.1955';
-- Returns a table with one row:
--  matrikel | name
-- ----------+------
--       242 | Hans
```

The OLAP Cube: Typical Operations [27]

- Slice: Fix one value to reduce the dimension by one
- Dice: Pick specific values of multiple dimensions
- Roll-up: Summarize data along a dimension
  - Formulas can be applied, e.g., profit = income - expense
- Pivot: Rotate the cube to see the faces

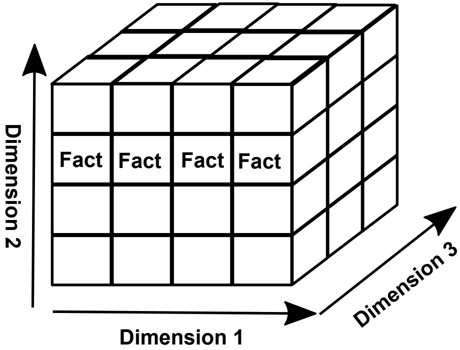


Figure: Example 3D cube

Star Schema Example Model

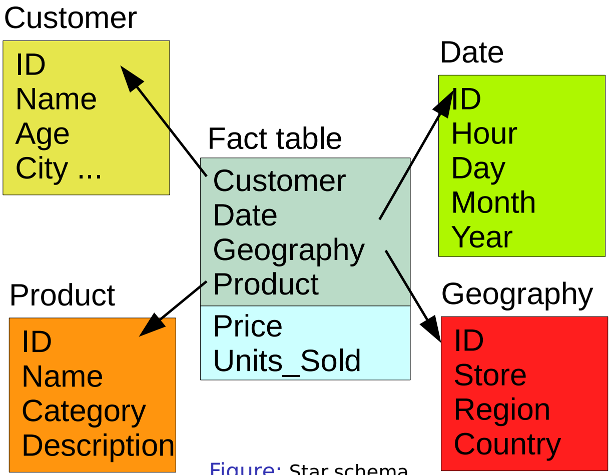


Figure: Star schema

Outline

1 Introduction

2 Data Models

3 Databases

4 Distributed Storage and Processing with Hadoop

5 Big Data SQL using Hive

6 Dataflow Computation

7 Columnar Access

8 Document Storage

System-Level Perspective of Hadoop Clusters

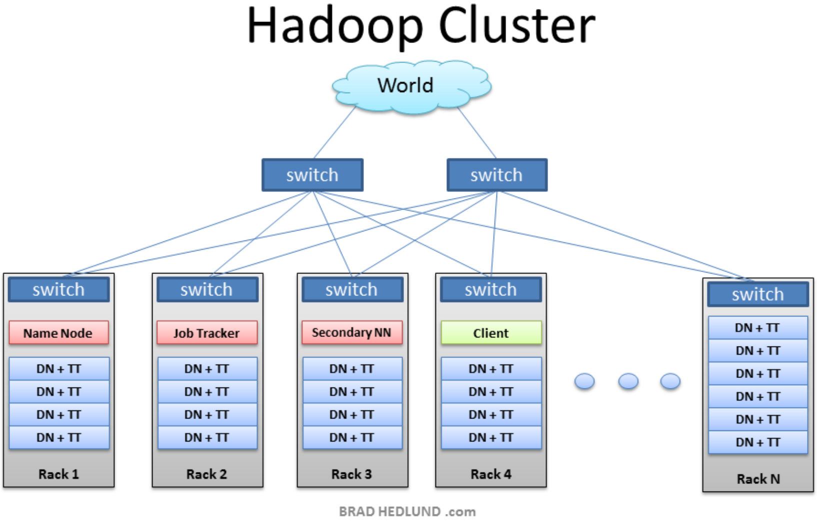
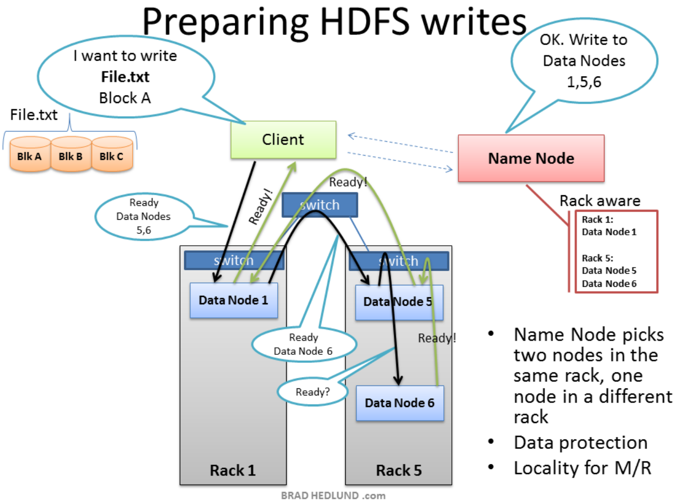


Figure: Source: B. Hedlund [15]

The HDFS Write Path



Preparing HDFS writes

- Name Node picks two nodes in the same rack, one node in a different rack
- Data protection
- Locality for M/R

Figure: Source: B. Hedlund [15]

Map Reduce Execution Paradigm

Idea: Appy a processing pipeline consisting of map and reduce operations

1. Map: filter and convert input records (pos, data) to tuples (key, value)
2. Reduce: receives all tuples with the same key (key, list<value>)

- Hadoop takes care of reading input, distributing (key,value) to reduce
- Types for key, value & format, records depend on the configuration

Example: WordCount [10]: Count word frequency in large texts

```
map(key, text): # input: key=position, text=line
  for each word in text:
    Emit(word,1) # outputs: key/value

reduce(key, list of values): # input: key == word, our mapper output
  count = 0
  for each v in values:
    count += v
  Emit(key, count) # it is possible to emit multiple (key, value) pairs here
```

Execution of MapReduce – the Big Picture

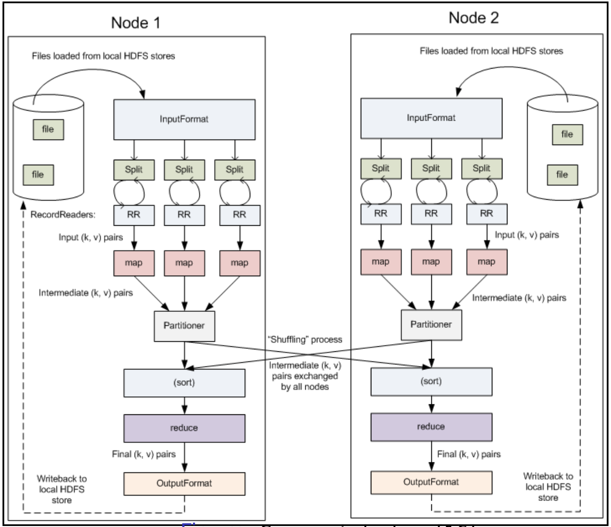


Figure: Source: jcdenton. [16]

Outline

1 Introduction

2 Data Models

3 Databases

4 Distributed Storage and Processing with Hadoop

5 Big Data SQL using Hive

6 Dataflow Computation

7 Columnar Access

8 Document Storage

Data Model [22]

Data types

- Primitive types (int, float, strings, dates, boolean)
- Bags (arrays), dictionaries
- Derived data types (structs) can be defined by users

Data organization

- Table: Like in relational databases with a schema
  - The Hive data definition language (DDL) manages tables
  - **Data is stored in files on HDFS**
- Partitions: table key determining the mapping to directories
  - Reduces the amount of data to be accessed in filters
  - Example key: /ds=<date> for table T
  - Predicate T.ds='2017-09-01' searches for files in /ds=2017-09-01/ directory
- Buckets/Clusters: Data of partitions are mapped into files
  - Hash value of a column determines partition

Hive Architecture and Query Execution in Hadoop

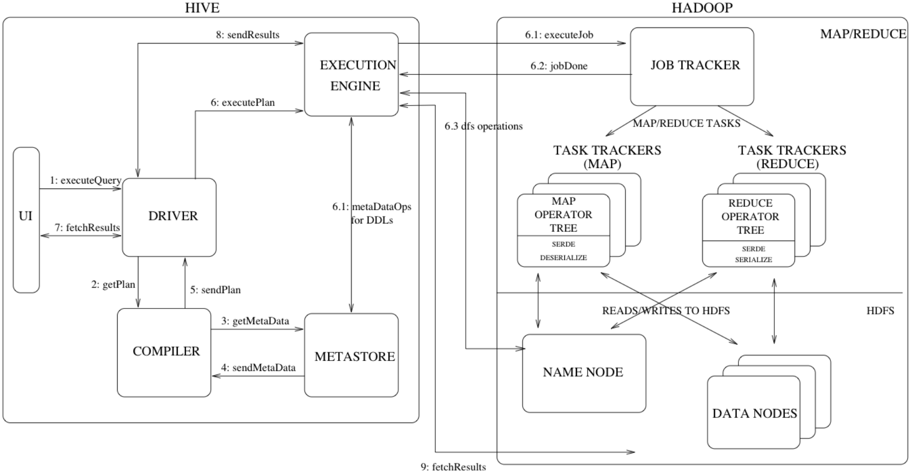


Figure: Architecture. Source: Design – Apache Hive [22]

ORC Files [25]

- Stripe: group of row data
- Postcript: contains file metadata
  - Compression parameters
  - Size of the file footer
- Index data (per stripe & row group)
  - Min and max values
  - Bloom filter (to pre-filter matches)
  - Row position
- Compression of blocks: RLE, ZLIB, SNAPPY, LZO
- Tool to output ORC files:
  `hive -orcfiledump`

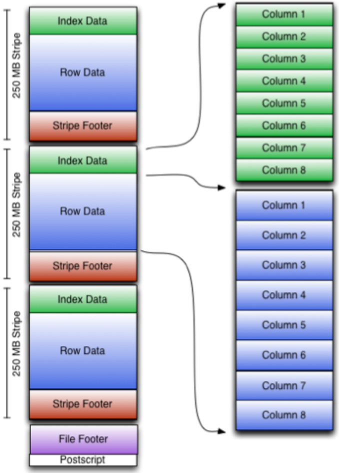


Figure: Source: [25]

Row groups are by default 10k rows of one column

Outline

1 Introduction

2 Data Models

3 Databases

4 Distributed Storage and Processing with Hadoop

5 Big Data SQL using Hive

6 Dataflow Computation

7 Columnar Access

8 Document Storage

General Data Model for Dataflow Languages

Data

- Tuple $t = (x_1, ..., x_n)$ where x_i may be of a given type
- Input/Output = list of tuples (like a table)

Typical Operators for Data-Flow Processing

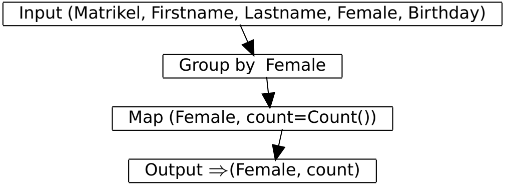
- Operations process individual tuples
  - Map/Foreach: process or transform data of **individual tuples or group**
    - transform a tuple: student.Map((matrikel, name) ⇒ (matrikel + 4, name))
    - count members for each group: groupedStudents.Map((year) ⇒ count())
  - Filter tuples by comparing a key to a value
- Operations that require the complete input data
  - Group tuples by a key
  - Sort data according to a key
  - Join multiple relations together
  - Split tuples of a relation into multiple relations (based on a condition)

Pipe Diagrams[7]

- Goal: Visualize the processing pipeline of data-flows with a schema
  - Optional: Add examples to illustrate processing

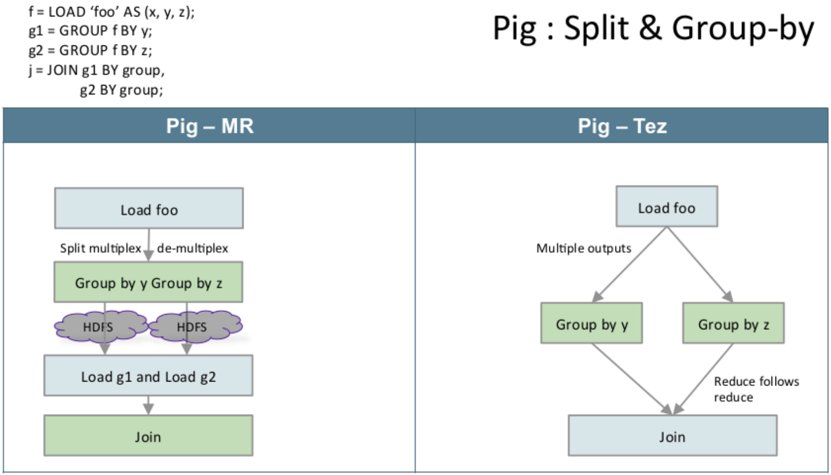
Elements and diagram concepts

- Box: Operation
  - e.g., functions, filter, grouping, aggregating, mapping
  - Indicate also changes in schema
- Arrows show processing order (DAG), joins have two inputs


Input (Matrikel, Firstname, Lastname, Female, Birthday)
Group by Female
Map (Female, count=Count())
Output ⇒(Female, count)

[7] We will use a variant from [11]

Execution of Pig Queries on MapReduce and TEZ



```
f = LOAD ‘foo’ AS (x, y, z);
g1 = GROUP f BY y;
g2 = GROUP f BY z;
j = JOIN g1 BY group,
         g2 BY group;
```

Pig : Split & Group-by

Figure: Source: H. Shah [20]

Outline

1 Introduction

2 Data Models

3 Databases

4 Distributed Storage and Processing with Hadoop

5 Big Data SQL using Hive

6 Dataflow Computation

7 Columnar Access

8 Document Storage

Zookeeper Overview [39, 40]

- Centralized service for coordination providing
  - Configuration information (e.g., service discovery)
  - Distributed synchronization (e.g., locking)
  - Group management (e.g., nodes belonging to a service)
- Simple: Uses a hierarchical namespace for coordination
- Strictly ordered access semantics
- Distributed and reliable using replication
- Scalable: A client can connect to any server

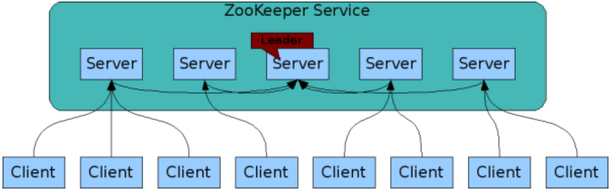


Figure: Source: ZooKeeper Service [40]

Main Operations of HBase

Data access

- **get**: return attributes for a row
- **put**: add row or update columns
- **increment**: increment values of multiple columns
- **scan**: iterate over multiple rows (potentially filtering)
- **delete**: remove a row, column or family
  - Data is only marked for deletion, finally removed during compaction

Schema operations

- **create**: create a table, specify the column families (flexible columns!)
- **alter**: change table properties
- **describe**: retrieve table/column family properties
- **list**: list tables
- **create_namespace**: create a namespace

Distribution of Data [30]

- HBase uses HDFS as backend to store data
  - Utilize replication and place servers close to data
- Server (RegionServer) manage key ranges on a per table bases
  - Buffer I/O to multiple files on HDFS
  - Performs computation (and data filtering)
- Regions: base element for availability and distribution of tables
  - One **store** object per ColumnFamily
  - One Memstore for each **store** to write data to files
  - Multiple StoreFiles (HFile format) for each store (each sorted)
- Catalog Table *HBase:meta*, special non splittable table
  - Contains a list of all regions $< table >, < regionstartkey >, < regionid >$

Table splitting

- Upon initialization of a table only one region is created
- Auto-Splitting: Based on a policy, a region is split into two
  - Typical policy: split when the region is sufficiently large
  - Benefit: increases parallelism, automatic scale-out

Sharding of a Table into Regions

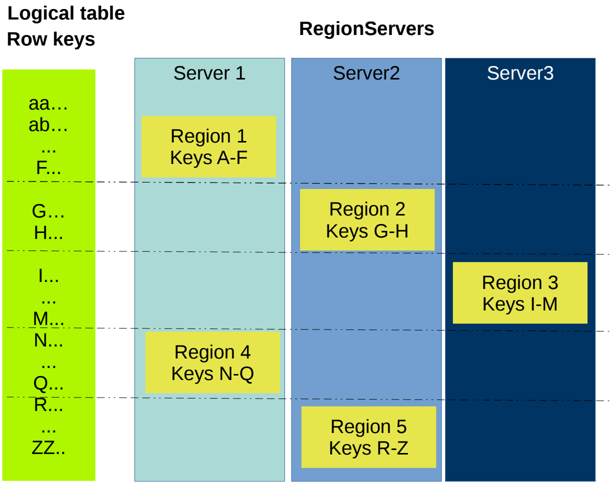


Figure: Distribution of keys to servers, values are stored with the row

Outline

1 Introduction

2 Data Models

3 Databases

4 Distributed Storage and Processing with Hadoop

5 Big Data SQL using Hive

6 Dataflow Computation

7 Columnar Access

8 Document Storage

MongoDB [11]

- Open-source document database
- High-performant and horizontally scalable for clusters
- Interfaces: Interactive mongo shell, REST, C, Python, ...
  - Connector for Hadoop for reading/writing to MongoDB

Data Model

- Database: As usual, defines permissions
- Document: BSON object (binary JSON) – Consisting of subdocuments
  - Primary key: _id field (manually set or automatically filled)

```
"_id" : ObjectId("43459bc2341bc14b1b41b124"),
"students" : [ # subdocument
  { "name" : "Julian", "id" : 4711, "birth" : ISODate("2000-10-01")},
  { "name" : "Hans",   "id" : 4712, "birth", ... }  ]
```

- Collection: Like a table of documents
  - Addressing: Collection name, document _id field (choose appropriately)
  - Documents can have individual schemas
  - Support for indexes on fields (and compound fields)
- Document references via object ids

Partitioning of Data (One Collection) [14]

- Shard key: Immutable field(s) in every collection document
  - Either by hashing of fields or by distributing ranges
  - Performance relevant: Select an appropriate shard key
- Chunk: A contiguous range of shard key values
  - Chunks are automatically split and migrated between shards

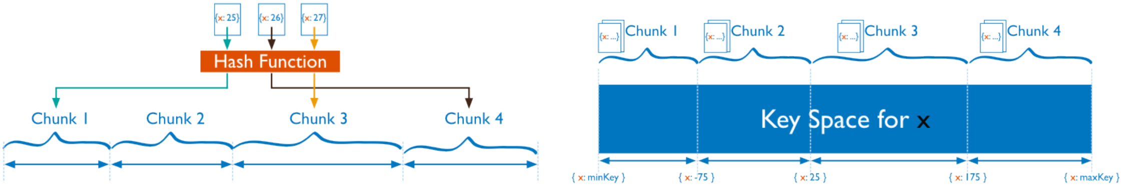


Figure: Hash and ranged sharding – Source: Reference [14]

- Internal processing of queries
  - Broadcast (scatter-gather) necessary if the query filter does not contain the shard key
  - If shard key is part of the query, only the subset of servers is contacted

Examples

```
# Bulk insert some values into the collection uni (to be created)
var bulk = db.uni.initializeUnorderedBulkOp();
bulk.insert({"_id": "4711", "name": "Julian", "gender": "male", "major": "computer science", "birth": ISODate("2000-10-01")})
bulk.insert({"_id": "4712", "name": "Hans", "gender": "male", "major": "computer science", "birth": ISODate("2000-10-01")})
bulk.execute()
# BulkWriteResult({ "writeErrors" : [ ],    "writeConcernErrors" : [ ], "nInserted" : 2, "nUpserted" : 0,  "nMatched" : 0, "nModified" : 0, "nRemoved" :
       ↪ 0,"upserted" : [ ] })

# Create an index on the student's name
db.uni.createIndex( { "name": 1 } )

# Return the first 10 student names
db.uni.find( {}, {"name" : 1} ).limit(10)
#{ "_id" : "4711", "name" : "Julian" }
#{ "_id" : "4712", "name" : "Hans" }

# Return the student birth data where the name matches Hans
db.uni.find( { "name" : "Hans" }, {"birth" : 1} )
# { "_id" : "4712", "birth" : ISODate("2000-10-01T00:00:00Z") }

# Update the student, adding an address to all students with name Julian
db.uni.update ( {"name" : "Julian" }, {$set : { "address" : { "plz" : 4711, "city" : "Hamburg" } } }, {multi: true} )
# WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

# Aggregate to count the number of male and female computer science students
# The match stage filters the documents first
# The _id field indicates the field to use for grouping, here gender
db.uni.aggregate( [ { $match: { "major": "computer science"} },
                    { $group: { "_id": "$gender", "count": { $sum: 1 } } }   ] )
# Returns: { "_id" : "male", "count" : 2 }

db.uni.drop() # remove collection
```

Outline

9 In-Memory Computation

10 Stream Processing

11 Designing Distributed Systems

12 Performance Modelling

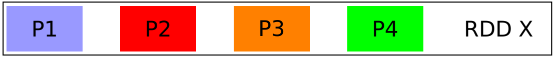
13 Visual Analytics

14 Large Scale Data Analytics

15 Data Storage

Spark Data Model

- Distributed memory model: Resilient Distributed Datasets (RDDs)
  - Named collection of elements distributed in partitions


P1 P2 P3 P4 RDD X

$X = [1, 2, 3, 4, 5, ..., 1000]$ distributed into 4 partitions

  - Typically a list or a map (key-value pairs)
  - An RDD is immutatable, e.g., cannot be changed
  - High-level APIs provide additional representations
    - e.g., SparkSQL uses DataFrames (aka tables)
- Shared variables offer shared memory access
- Durability of data
  - RDDs live until the SparkContext is terminated
  - To keep them, they need to be persisted (e.g., to HDFS)
- Fault-tolerance is provided by **re-computing** data (if an error occurs)

Resilient Distributed Datasets (RDDs) [13]

- Creation of an RDD by either
  - Parallelizing an existing collection

```
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data, 5) # create 5 partitions
```

  - Referencing a dataset on distributed storage, HDFS, ...

```
rdd = sc.textFile("data.txt")
```

- RDDs can be transformed into derived (newly named) RDDs

```
rdd2 = rdd.filter( lambda x : (x % 2 == 0) ) # operation: filter odd tuples
```

  - RDDs can be redistributed (called shuffle)
  - RDD is computed if needed, but RDD can be cached in memory or stored
  - Computation runs in parallel on the partitions
  - RDD knows its data lineage (how it was computed)
- Fault-tolerant collection of elements (lists, dictionaries)
  - Split into choosable number of partitions and distributed
  - Derived RDDs can be re-computed by using the recorded lineage

Execution of Applications [12, 21]

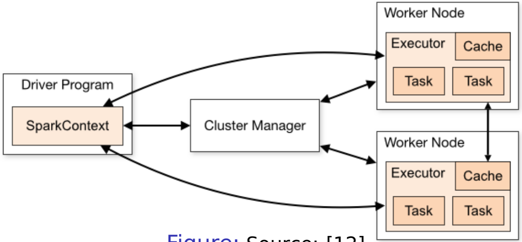


Figure: Source: [12]

- Driver program: process runs `main()`, creates/uses SparkContext
- Task: A unit of work processed by one executor
- Job: A spark action triggering computation starts a job
- Stage: collection of tasks executing the same code; run concurrently
  - Works independently on partitions without data shuffling
- Executor process: provides slots to runs tasks
  - Isolates apps, thus data cannot be shared between apps
- Cluster manager: allocates cluster resources and runs executor

Computation

- **Lazy execution**: apply operations when results are needed (by actions)
  - Intermediate RDDs can be re-computed multiple times
  - Users can persist RDDs (in-memory or disk) for later use
- Many operations apply user-defined functions or **lambda** expressions
- Code and **closure** are serialized on the driver and send to executors
  - Note: When using class instance functions, the object (and all members) are serialized
- RDD partitions are processed in parallel (data parallelism)
  - Concept: Use local data where possible

RDD Operation Types [13]

- **Transformations**: create a new RDD locally by applying operations
- **Actions**: return values to the driver program (or do I/O)
- **Shuffle operations**: re-distribute data across executors

Simple Example

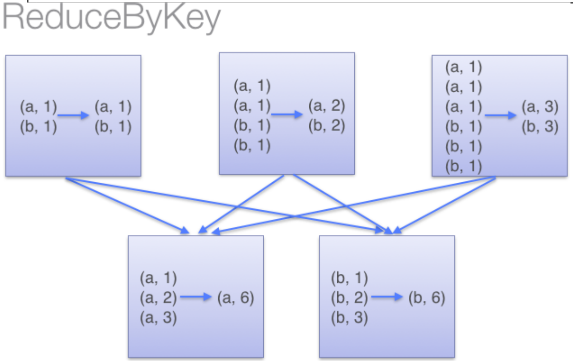
- Example session when using pyspark

```
# Distribute the data: here we have a list of numbers from 1 to 10 million
# Store the data in an RDD called nums
nums = sc.parallelize( range(1,10000000) )

# Compute a derived RDD by filtering odd values
r1 = nums.filter( lambda x : (x % 2 == 1) )

# Now compute squares for all remaining values and store key/value tuples
result = r1.map( lambda x : (x, x*x*x) )
# Store results in memory, cached at first invocation of an action
resultCached = result.cache()

# Retrieve all distributed values into the driver and print them
# This will actually run the computation
print(result.collect())  # [(1, 1), (3, 27), (5, 125), (7, 343), (9, 729), (11, 1331), ... ]
```

Illustrating processing of KV RDDs [30]

- ReduceBy applies reduction function locally, creates new RDD and then globally
- Reduces network traffic, prefferable solution

```
words = ["one", "two", "two", "three", "three", "three"]
wordPairsRDD = sc.parallelize(words).map(lambda word : (word, 1))
wordCountsWithReduce = wordPairsRDD.reduceByKey(lambda a, b : a+b).collect()
# [('two', 2), ('three', 3), ('one', 1)]
```



Creating an In-memory Table from an RDD

```
# Create a table from an array using the column names value, key
# The data types of the columns are automatically inferred
r = sqlContext.createDataFrame([(’test’, 10), (’data’, 11)], ["value", "key"])

# Alternative: create/use an RDD
rdd = sc.parallelize(range(1,10)).map(lambda x : (x, str(x)) )

# Create the table from the RDD using the columnnames given, here "key" / "value"
schema = sqlContext.createDataFrame(rdd, ["key", "value"])
schema.printSchema()

# Register table for use with SQL, we use a temporary table, so the table is NOT visible in Hive
schema.registerTempTable("nums")

# Now you can run SQL queries
res = sqlContext.sql("SELECT * from nums")

# res is an DataFrame that uses columns according to the schema
print( res.collect() ) # [Row(key=1, value=’1’), Row(key=2, value=’2’), ... ]

# Save results as a table for Hive
from pyspark.sql import DataFrameWriter
dw = DataFrameWriter(res)
dw.saveAsTable("data")
```

Outline

9 In-Memory Computation

10 Stream Processing

11 Designing Distributed Systems

12 Performance Modelling

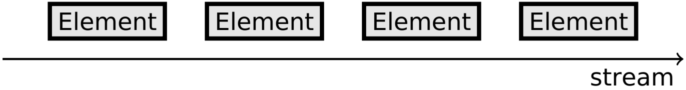
13 Visual Analytics

14 Large Scale Data Analytics

15 Data Storage

Stream Processing [12]

- Stream processing paradigm = dataflow programming
- Programming
  - Implement operations (kernel) functions and define data dependencies
  - Uniform streaming: Operation is executed on all elements individually
  - ⇒ Default: no view of the complete data at any time
- Advantages
  - Pipelining of operations and massive parallelism is possible
  - Data is in memory and often in CPU cache, i.e., in-memory computation
  - Data dependencies of kernels are known and can be dealt at compile time



Overcoming restrictions of the programming model

- Windowing: sliding (overlapping) windows contain multiple elements
- Stateless vs. stateful (i.e., keep information for multiple elements)

Storm Data Model [37, 38]

- Tuple: an ordered list of named elements
  - e.g., fields (weight, name, BMI) and tuple (1, “hans”, 5.5)
  - Dynamic types (i.e., store anything in fields)
- Stream: a sequence of tuples
- Spouts: a source of streams for a computation
  - e.g., Kafka messages, tweets, real-time data
- Bolts: processors for input streams producing output streams
  - e.g., filtering, aggregation, join data, talk to databases
- Topology: the graph of the calculation represented as network
  - Note: the parallelism (tasks) is statically defined for a topology

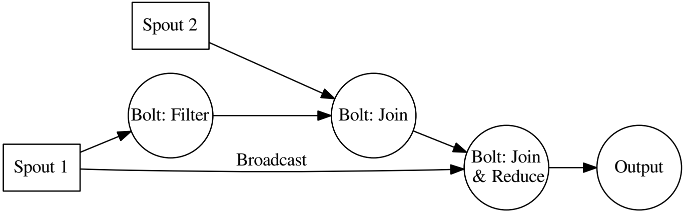
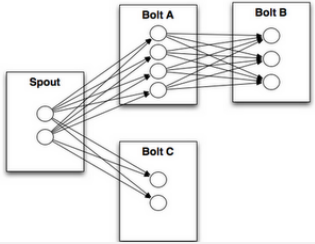


Figure: Example topology

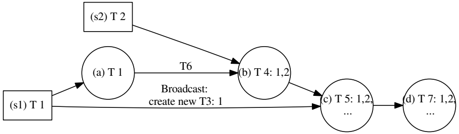
Partitions and Stream Groupings [38]

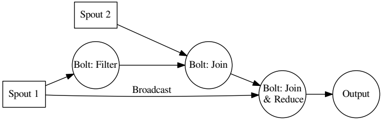
- Multiple instances (tasks) of spouts/bolts each processes a partition
- Stream grouping defines how to transfer tuples between partitions
- Selection of groupings (we note similarities to YARN)
  - Shuffle: send a tuple to a random task
  - Field: send tuples which share the values of a subset of fields to the same task, e.g., for counting word frequency
  - All: replicate/Broadcast tuple across all tasks of the target bolt
  - Local: prefer local tasks if available, otherwise use shuffle
  - Direct: producer decides which consumer task receives the tuple



Processing Strategy [11, 54]

- Track tuple processing
  - Each tuple has a random 64 Bit message ID
  - Explicit record **all spout tuple IDs** a tuple is derived of
- **Acker task** tracks the tuple DAG implicitly for each tuple
  - Spout informs Acker tasks of new tuple
  - Acker notifies all Spouts if a “derived” tuple completed
  - Hashing maps spout tuple ID to Acker task
- Acker uses 20 bytes per tuple to track the state of the tuple tree[8]
  - Map contains: tuple ID to Spout (creator) task AND 64 Bit ack value
  - Ack value is an XOR of all “derived” tuple IDs and all acked tuple IDs
  - If Ack value is 0, the processing of the tuple is complete





[8] Independent of the size of the topology!

Exactly-Once Semantics [11, 54]

- Semantics guarantees each tuple is executed exactly once
- Operations depending on exactly-once semantics
  - Updates of stateful computation
  - Global counters (e.g., wordcount), database updates

Strategies to achieve exactly-once semantics

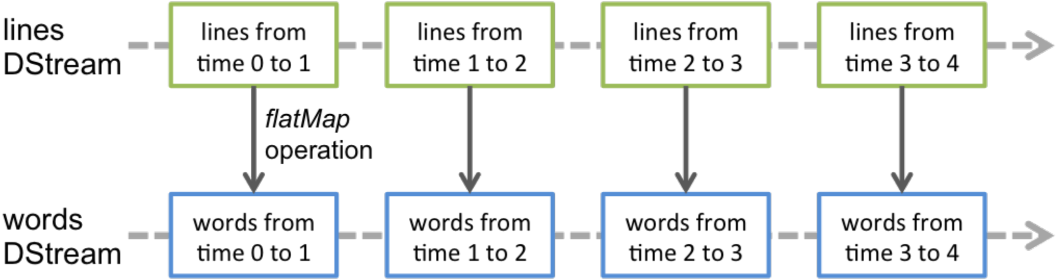
1. Provide idempotent operations: $f(f(tuple)) = f(tuple)$
   - Stateless (side-effect free) operations are idempotent
2. Execute tuples strongly ordered to avoid replicated execution
   - Create tuple IDs in the spout with a strong ordering
   - Bolts memorize last seen / executed tuple ID (transaction ID)
     - Perform updates only if tuple ID $>$ last seen ID

     ⇒ ignore all tuples with tuple ID $<$ failure
   - Requirement: Don’t use random grouping
3. Use Storm’s transactional topology [57]
   - Separate execution into processing phase and commit phase
     - Processing does not need exactly-once semantics
     - Commit phase requires strong ordering

Spark: Processing of Streams

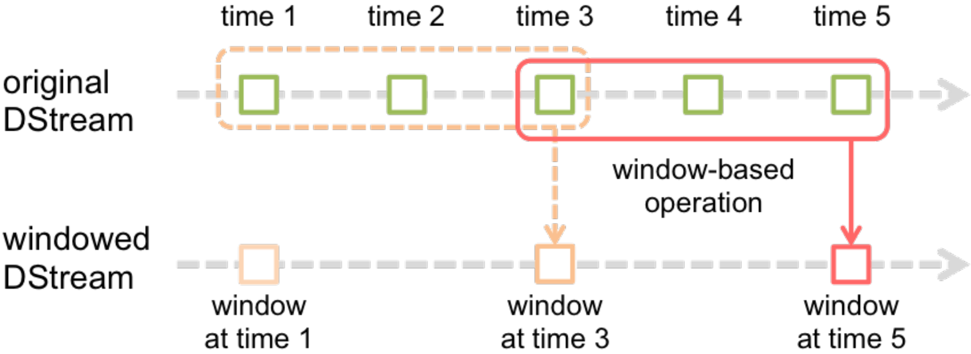
Basic processing concept is the same as for RDDs, example:

```python
words = lines.flatMap(lambda l: l.split(" "))
```



Window-Based Operations

```
# Reduce a window of 30 seconds of data every 10 seconds
rdd = words.reduceByKeyAndWindow(lambda x, y: x + y, 30, 10)
```



Outline

9 In-Memory Computation

10 Stream Processing

11 Designing Distributed Systems

12 Performance Modelling

13 Visual Analytics

14 Large Scale Data Analytics

15 Data Storage

Semantics of a Service

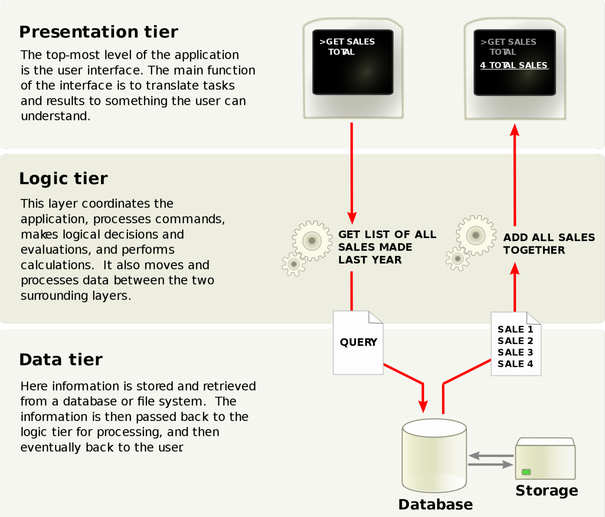
Semantics describe operations and their behavior, i.e., the property of the service

- Application programming interface (API)
- **Consistency**: Behavior of simultaneously executed operations
  - Atomicity: Are partial modifications visible to other clients
  - Visibility: When are changes visible to other clients
  - Isolation: Are operations influencing other ongoing operations
- **Availability**: Readiness to serve operations
  - Robustness of the system for typical (hardware and software) errors
  - Scalability: availability and performance behaviour depending on the number of clients, concurrent requests, request size, etc.
  - Partition tolerance: Continue to operate even if the network breaks partially
- **Durability**: Modifications should be stored on persistent storage
  - Consistency: Any operation leaves a consistent (correct) system state

Wishlist for Distributed Software

- High-availability, i.e., you can use the service all the time
- Fault-tolerance, i.e., can tolerate errors
- Scalable, i.e., the ability to be used in a range of capabilities
  - Linear scalability with the data volume (or number of users served)
    - i.e., 2n servers handle 2n the data volume + same processing time
- Extensible, i.e., easy to introduce new features and data
- Usability: high user productivity - i.e., simple programming models
- Ready for the cloud
- Debuggability
  - In respect to coding errors and performance issues
- High Performance
  - Real-time/interactive capabilities - user interact with the system without noticing delay
- High efficiency, i.e., make good use of resources (compute and storage)

Multitier architecture [25]



Presentation tier

The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

Logic tier

This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

Data tier

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.

Figure: Source: [25]

Two-Phase Commit Protocol (2PC) [18]

- Idea: one process coordinates commit and checks that all agree on the decision

Sketch of the algorithm

1. Prepare phase
   1. Coordinator sends message with transaction to all participants
   2. Participant executes transaction until commit is needed.
      Replies yes (commit) or no (e.g. conflict). Records changes in undo/redo logs
   3. Coordinator checks decision by all replies, if all reply yes, decide commit
2. Commit phase
   1. Coordinator sends message to all processes with decision
   2. Processes commit or rollback the transactions, send acknowledgment
   3. Coordinator sends reply to requester

- Think about: What should happen if the coordinator fails?
- What should a "participant" do upon such failures, how to detect them?

Consistent Hashing (2)

- In this example, server IP addresses are hashed to the ring
  - They could be hashed several times for fault tolerance
- The items are strings, the hash determines where they are located
- The arrow shows the server responsible for the items

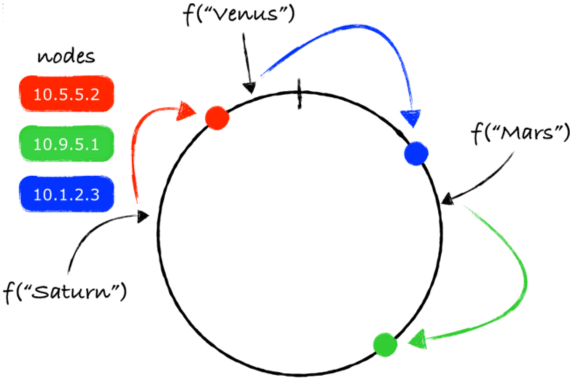


Figure: Source: [22]

- For more info, see `https://www.youtube.com/watch?v=juxlRh4ZhoI` and [22], [23]

REST [31]

- Advantages of REST due to HTTP
  - Simplicity of the interfaces
  - Portability: Independent of client and server platform
  - Cachable: Read requests can be cached close to the user
  - Tracable: Communication can be inspected

Semantics of HTTP request verbs [33]

- GET: retrieve a representation of a resource (no updates)
- PUT: store the enclosed data under the given URI
- POST: transfer an entity/data as a subordinate of the web resource
- DELETE: remove the given URI
- PUT and DELETE are idempotent
  - GET also w/o concurrent updates

Outline

9 In-Memory Computation

10 Stream Processing

11 Designing Distributed Systems

12 Performance Modelling

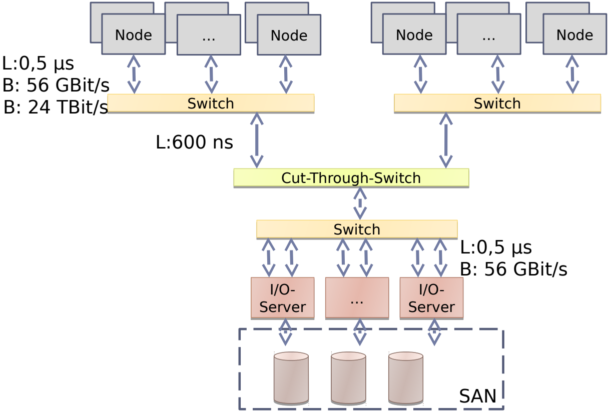
13 Visual Analytics

14 Large Scale Data Analytics

15 Data Storage

HPC Cluster Characteristics

- High-end components
- Extra fast interconnect, global/shared storage with dedicated servers
- Network provides high (near-full) bisection bandwidth. Various topologies are possible.



Basic Approach

Question

Is the observed performance acceptable?

Basic Approach

Start with a simple model

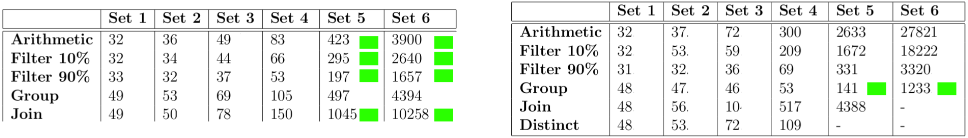
1. Measure time for the execution of your workload
2. Quantify the workload with some metrics
   - E.g., amount of tuples or data processed, computational operations needed
   - E.g., you may use the statistics output for each Hadoop job
3. Compute W, the workload you process per time
4. Compute the expected performance P based on the system's hardware characteristics
5. Compare W with P, the efficiency is $E = \frac{W}{P}$
   - If $E << 1$, e.g., 0.01, you are using only 1% of the potential!

Refine the model as needed, e.g., include details about intermediate steps

Discussion: Comparing Pig and Hive Big Data Solutions

Benchmark by IBM [16], similar to Apache Benchmark

- Tests several operations, data set increases 10x in size
  - Set 1: 772 KB; 2: 6.4 MB; 3: 63 MB; 4: 628 MB; 5: 6.2 GB; 6: 62 GB
- Five data/compute nodes, configured to run eight reduce and 11 map tasks



	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Arithmetic	32	36	49	83	423 🟩	3900 🟩
Filter 10%	32	34	44	66	295 🟩	2640 🟩
Filter 90%	33	32	37	53	197 🟩	1657 🟩
Group	49	53	69	105	497	4394
Join	49	50	78	150	1045 🟩	10258 🟩

	Set 1	Set 2	Set 3	Set 4	Set 5	Set 6
Arithmetic	32.	37.	72.	300	2633	27821
Filter 10%	32.	53.	59.	209	1672	18222
Filter 90%	31.	32.	36.	69	331	3320.
Group	48.	47.	46.	53	141 🟩	1233. 🟩
Join	48.	56.	10	517	4388	-
Distinct	48.	53.	72.	109	-	-

Figure: Time for **Pig (left) and Hive**. Source: B. Jakobus (modified), "Table 2: Averaged performance" [16]

Assessing performance

- How could we model performance here?
- How would you judge the runtime here?

Outline

9 In-Memory Computation

10 Stream Processing

11 Designing Distributed Systems

12 Performance Modelling

13 Visual Analytics

14 Large Scale Data Analytics

15 Data Storage

Visual Analytics Workflow

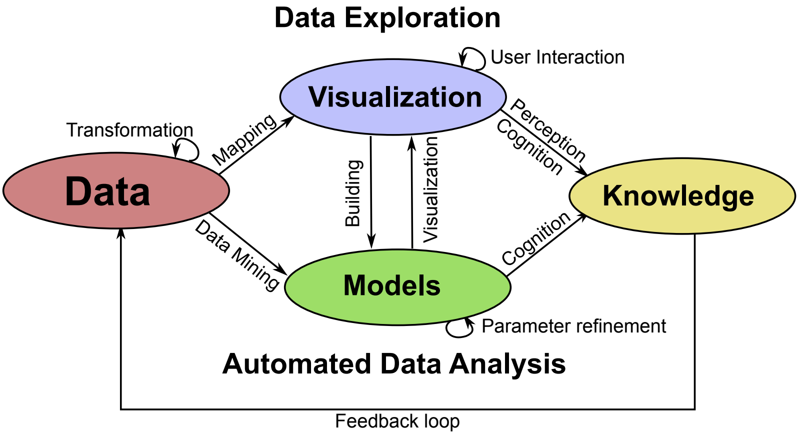


Figure: Figure based on [48]

Motto: Analyse First – Show the Important; Zoom, Filter and Analyse Further – Details on

Guidelines for Graphical Displays

Goals of **graphical displays** according to [42]

- show the data
- induce the viewer to **think about the substance** rather than about methodology, graphic design, the technology of graphic production, or something else
- avoid distorting what the data have to say
- present many numbers in a small space
- make large data sets coherent
- encourage the eye to compare different pieces of data
- reveal the data at several levels of detail, from a broad overview to the fine structure
- serve a reasonably clear purpose: description, exploration, tabulation, or decoration
- be closely integrated with the statistical and verbal descriptions of a data set

Guidelines

Simple rules

- Use the right visualization for the for data types
- Use building blocks for graphics (known plot styles)
- Reduce information to the essential part to be communicated
- Consistent use of building blocks and themes (retinal properties)

Outline

9 In-Memory Computation

10 Stream Processing

11 Designing Distributed Systems

12 Performance Modelling

13 Visual Analytics

14 Large Scale Data Analytics

15 Data Storage

Large Scale Data Analytics for Scientific Computing

Scientific Computing

- Large-scale computing on the frontier of science
- Traditional workflow: execute scientific application, store results, analyze results

Challenges

- Large data volumes and velocities
  - How can we analyze 1 PByte of data?
  - How can we manage 100 M files?
- Complex system (and storage) topologies
- Understanding/optimization of system behavior is difficult
- Data movement between CPU and even memory storage is costly
  - 5000x more than a DP FLOP[9]
  - 10 pJ per Flop (2018), 2000 pJ for DRAM access

[9] http://www.fatih.edu.tr/ esma.yildirim/DIDC2014-workshop/DIDC-parashar.pdf

In-situ and in-transit Analysis/Processing

- **In-situ**: analyze results while the application is still computing
  - How: define computation (e.g. data flow graph) of data a-priori
  - Runtime deploys them with application execution
  - Typically on either the same nodes as the application or dedicated servers
- **In-transit**: analyze/post-process data while it is on the I/O path
  - Extend in-situ idea with means to deploy parts of the processing across system
- **Computational steering**: interact with the application while it runs
  - e.g., modify simulation parameters, modify objects
- Example solutions that support analysis
  - DataSpaces[10]
  - ADIOS[11]
  - Paraview (with Catalyst)

[10] http://www.fatih.edu.tr/ esma.yildirim/DIDC2014-workshop/DIDC-parashar.pdf

[11] Paper: Combining in-situ and in-transit processing to enable extreme-scale scientific analysis, 2012

Outline

9 In-Memory Computation

10 Stream Processing

11 Designing Distributed Systems

12 Performance Modelling

13 Visual Analytics

14 Large Scale Data Analytics

15 Data Storage

The I/O Stack

- Parallel application
  - ▶ Is distributed across many nodes
  - ▶ Has a specific access pattern for I/O
  - ▶ May use several interfaces
    File (POSIX, ADIOS, HDF5), SQL, NoSQL
- Middleware provides high-level access
- POSIX: ultimately file system access
- Parallel file system: Lustre, GPFS, PVFS2
- File system: EXT4, XFS, NTFS
- Block device: utilizes storage media to export a block API
- Operating system: (orthogonal aspect)

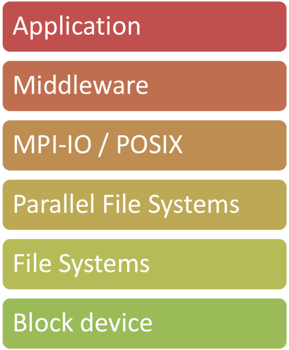
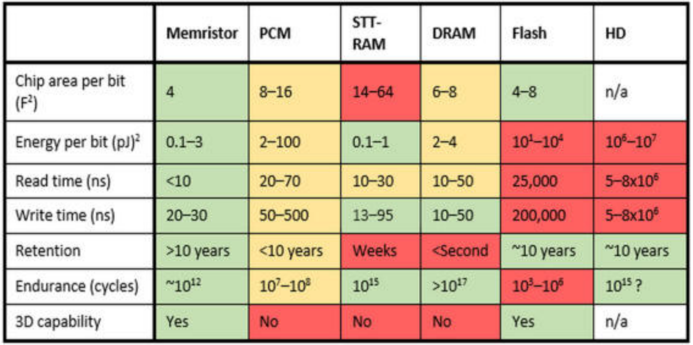


Figure: Example I/O stack

Storage Media

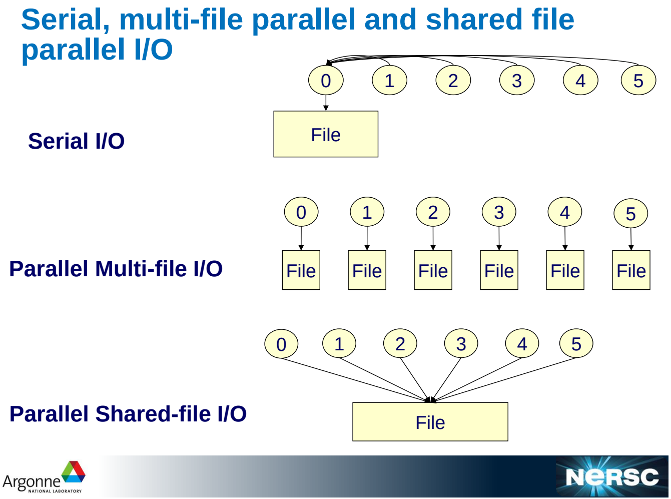
- Many technologies are available with different characteristics
- Block-accessible or byte-adressable (NVRAM)



	Memristor	PCM	STT-RAM	DRAM	Flash	HD
Chip area per bit (F^2)	4	8–16	14–64	6–8	4–8	n/a
Energy per bit (pJ)[2]	0.1–3	2–100	0.1–1	2–4	10^1–10^4	10^6–10^7
Read time (ns)	<10	20–70	10–30	10–50	25,000	5–8x10^6
Write time (ns)	20–30	50–500	13–95	10–50	200,000	5–8x10^6
Retention	>10 years	<10 years	Weeks	<Second	~10 years	~10 years
Endurance (cycles)	~10^{12}	10^7–10^8	10^{15}	>10^{17}	10^3–10^6	10^{15} ?
3D capability	Yes	No	No	No	Yes	n/a

Figure: Source: ZDNet [100]

Application I/O Types



Serial, multi-file parallel and shared file parallel I/O

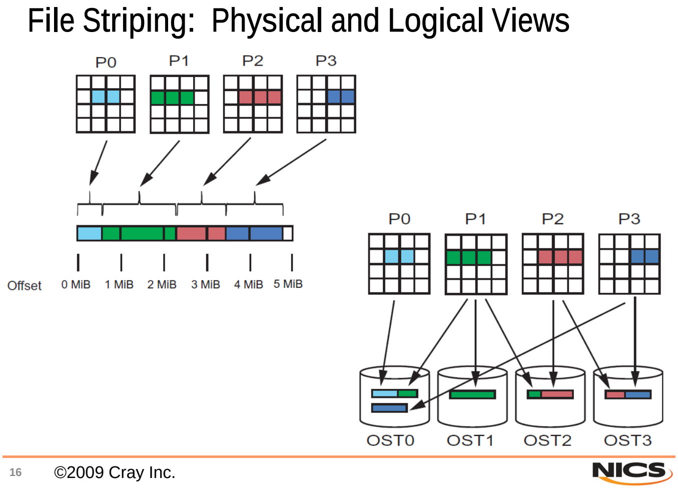
Serial I/O

Parallel Multi-file I/O

Parallel Shared-file I/O

Figure: Source: Lonnie Crosby [101]

File Striping: Distributing Data Across Devices



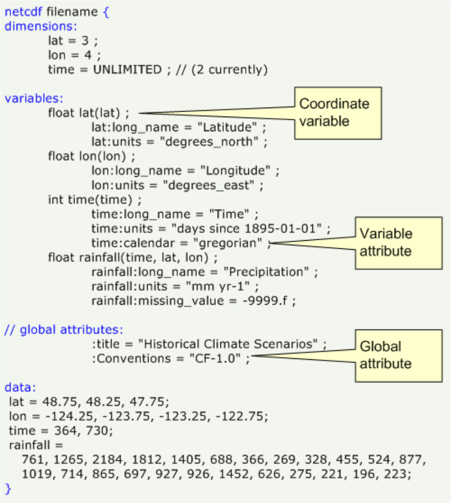
File Striping: Physical and Logical Views

16 ©2009 Cray Inc.

NICS

NetCDF: Common Data form Language

- Notation used to describe NetCDF object is called CDL (network Common Data form Language)
  - Provides a convenient way of describing NetCDF datasets
- Utilities allow producing CDL text files from binary NetCDF datasets and vice-versa
- File contains dimensions, variables, and attributes
- Components are used together to capture the meaning of data and relations among data fields



```
netcdf filename {
dimensions:
        lat = 3 ;
        lon = 4 ;
        time = UNLIMITED ; // (2 currently)

variables:
        float lat(lat) ;
                lat:long_name = "Latitude" ;
                lat:units = "degrees_north" ;
        float lon(lon) ;
                lon:long_name = "Longitude" ;
                lon:units = "degrees_east" ;
        int time(time) ;
                time:long_name = "Time" ;
                time:units = "days since 1895-01-01" ;
                time:calendar = "gregorian" ;
        float rainfall(time, lat, lon) ;
                rainfall:long_name = "Precipitation" ;
                rainfall:units = "mm yr-1" ;
                rainfall:missing_value = -9999.f ;

// global attributes:
                :title = "Historical Climate Scenarios" ;
                :Conventions = "CF-1.0" ;

data:
 lat = 48.75, 48.25, 47.75;
 lon = -124.25, -123.75, -123.25, -122.75;
 time = 364, 730;
 rainfall =
   761, 1265, 2184, 1812, 1405, 688, 366, 269, 328, 455, 524, 877,
   1019, 714, 865, 697, 927, 926, 1452, 626, 275, 221, 196, 223;
}
```

Understanding of I/O Behavior and Systems

How can we understand system behavior?

- Observation
  - Measurement of runs on the system
  - Can be many cases to run
  - Slight bias since measurement perturbs behavior
  - Benchmarking: applications geared to exhibit certain system behavior
- Monitoring: system/tool-provided observation creation
- Theory: Performance models
  - Used to determine performance for a system/workload
  - Behavioral models

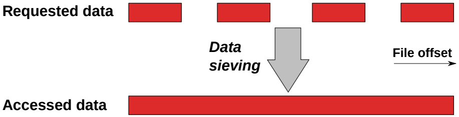
    Build models based on ensemble of observations
- System/application simulation
  - Based on system and workload models

How Can Benchmarks Help to Analyze I/O?

- Benefits of benchmarks
  - Can use simple/understandable sequence of operations
    - Ease comparison with theoretic values (that requires understandable metrics)
  - May use a pattern like a realistic workloads
    - Provides performance estimates or bounds for workloads!
  - Sometimes only possibility to understand hardware capabilities
    - Because the theoretic analysis may be infeasible
- Benefits of benchmarks vs. applications
  - Are easier to code/understand/setup/run than applications
  - Come with less restrictive "license" limitations
- Flexible testing (strategies)
  - Single-shot: e.g., acceptance test
  - Periodically: regression tests

Optimizations

- There are too many tunables and optimizations for I/O
  - Read-ahead, write-behind, async I/O
  - Distribution of data across servers (e.g., Lustre stripe size)
  - We will investigate the complexity of one example...
- Performance benefit of I/O optimizations is non-trival to predict
- Non-contiguous I/O supports data-sieving optimization
  - Transforms non-sequential I/O to large contiguous I/O
  - Tunable with MPI hints: enabled/disabled, buffer size
  - Benefit depends on system AND application



- Data sieving is difficult to parameterize
  - What should be recommended from a data center's perspective?

Summary

...