

Big Data Management Platforms

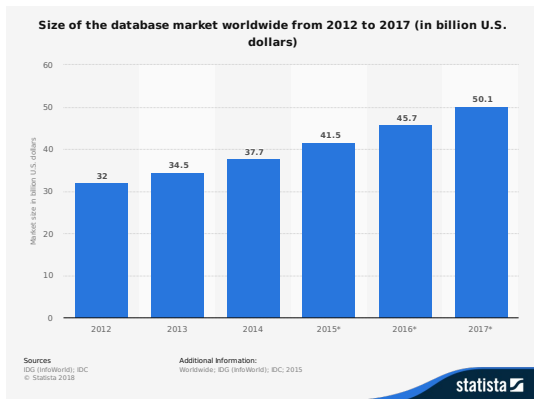
M. Tamer Özsu

University of Waterloo
David R. Cheriton School of Computer Science

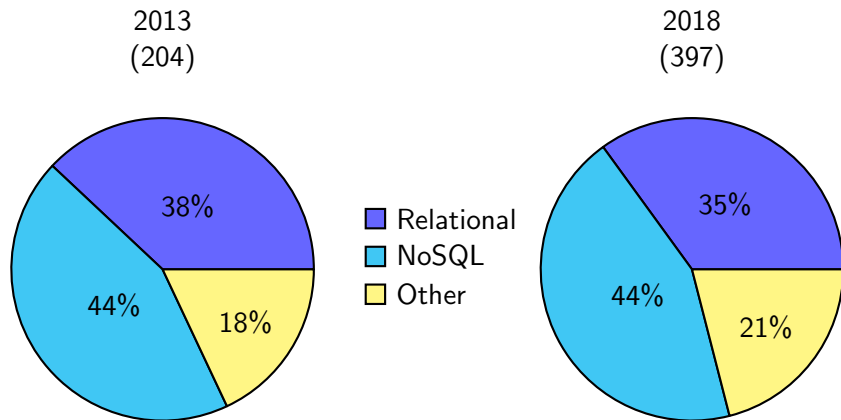


Relational DBMS market (IDC/Gartner)

- 2011: US\$24B
- 1994: US\$ 7B
- 2010-2011 growth 16.3%
- 2010-2017 growth 143.2%

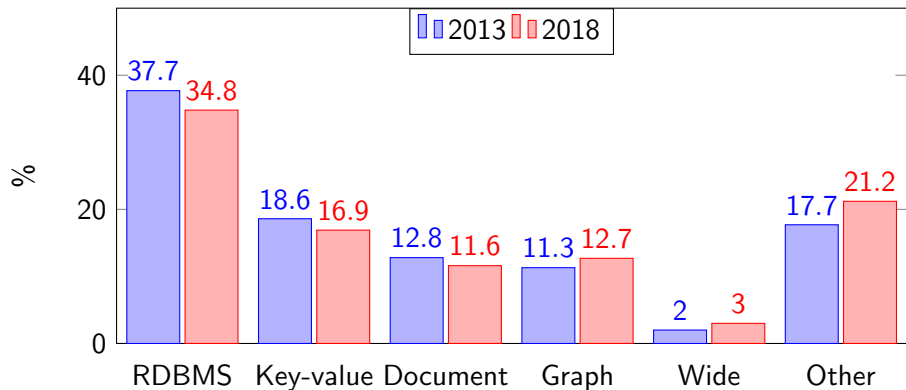


But...



Source: <https://db-engines.com>

But...



Source: <https://db-engines.com>

What is Happening?

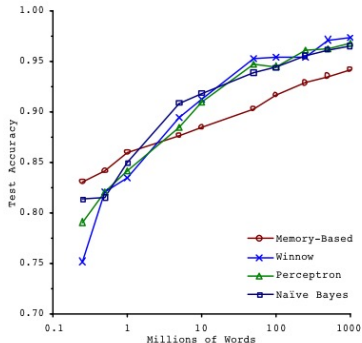
Technology push/Application Pull

- Many technological changes
 - Networks
 - Parallelism (Cluster computing & Multicore)
 - Storage systems
 - Processor technology
 - ...
- New applications
 - “Big data” applications
 - IoT
 - Recommender systems
 - Personalized health systems
 - ...

Data-driven approaches to solving big problems

Example

- Voice recognition
 - Google voice accuracy 90
 - Model-driven approach 60
- Question-Answering systems
- Confusion set disambiguation (e.g., to, two, too) [Banko & Brill, 2001]



Big Data Characteristics – Four Vs

40 ZETTABYTES

(43 TRILLION GIGABYTES)
of data will be created by 2020, an increase of 300 times from 2005



Volume
SCALE OF DATA

It's estimated that
2.5 QUINTILLION BYTES

(2.3 TRILLION GIGABYTES)
of data are created each day



Most companies in the U.S. have at least
100 TERABYTES
(100,000 GIGABYTES)
of data stored

The FOUR V's of Big Data

From traffic patterns and music downloads to web history and medical records, data is recorded, stored, and analyzed to enable the technology and services that the world relies on every day. But what exactly is big data, and how can these massive amounts of data be used?

As a leader in the sector, IBM data scientists break big data into four dimensions: **Volume, Variety, Velocity and Veracity**

Depending on the industry and organization, big data encompasses information from multiple internal and external sources such as transactions, social media, enterprise content, sensors and mobile devices. Companies can leverage data to adapt their products and services to better meet customer needs, optimize operations and infrastructure, and find new sources of revenue.

By 2015
4.4 MILLION IT JOBS
will be created globally to support big data, with 1.9 million in the United States



As of 2013, the global size of data in healthcare was estimated to be

150 EXABYTES
(151 BILLION GIGABYTES)



30 BILLION
PIECES OF CONTENT
are shared on Facebook every month



Variety
DIFFERENT FORMS OF DATA



By 2014, it's anticipated there will be

420 MILLION
WEARABLE, WIRELESS
HEALTH MONITORS

4 BILLION+
HOURS OF VIDEO
are watched on YouTube each month



400 MILLION TWEETS
are sent per day by about 200 million monthly active users



The New York Stock Exchange captures

1 TB OF TRADE
INFORMATION
during each trading session



Velocity
ANALYSIS OF
STREAMING DATA



Modern cars have close to
100 SENSORS
that monitor items such as fuel level and tire pressure

By 2016, it is projected there will be
18.9 BILLION
NETWORK
CONNECTIONS

~ almost 2.5 connections per person on earth



1 IN 3 BUSINESS
LEADERS

don't trust the information they use to make decisions



in one survey were unsure of how much of their data was inaccurate



Veracity
UNCERTAINTY
OF DATA

Poor data quality costs the US economy around

\$3.1 TRILLION A YEAR

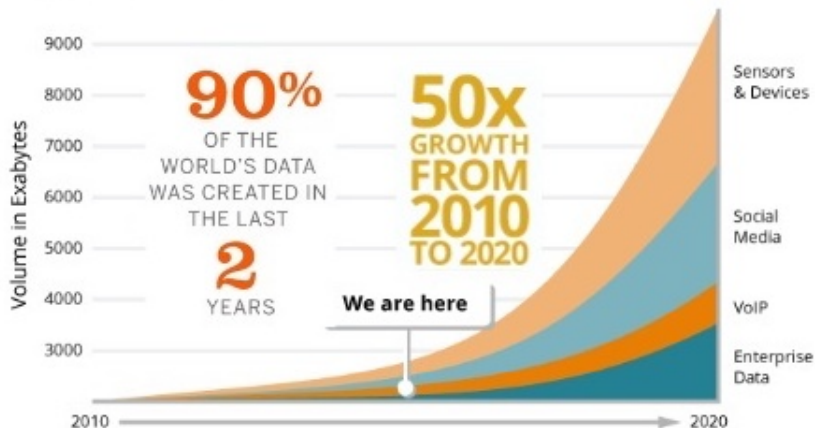


Sources: McKinsey Global Institute, Twitter, Cisco, Gartner, EMC, SAS, IBM, MEPEEC, QAS

Big Data Characteristics – Four Vs

- **Volume**

- Almost every dataset we use is high volume and volume is increasing



Big Data Characteristics – Four Vs

- **Volume**

- Almost every dataset we use is high volume and volume is increasing, but consider these
- Google: in 2016, user uploads to YouTube required 1PB of *new* storage capacity *per day*
- Alibaba: in 2017, 320 PB of log data was generated in a six hour period as a result of customer purchase activity

Big Data Characteristics – Four Vs

● Volume

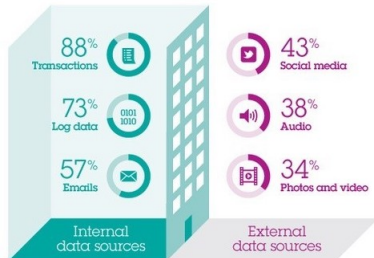
- Almost every dataset we use is high volume and volume is increasing, but consider these
- Google: in 2016, user uploads to YouTube required 1PB of *new* storage capacity *per day*
- Alibaba: in 2017, 320 PB of log data was generated in a six hour period as a result of customer purchase activity

● Variety

- Data are multi-modal

Where does big data come from?

Most big data efforts are currently focused on analyzing internal data to extract insights. Fewer organizations are looking at data outside their firewalls, such as social media.



Big Data Characteristics – Four Vs

- **Volume**

- Almost every dataset we use is high volume and volume is increasing, but consider these
- Google: in 2016, user uploads to YouTube required 1PB of *new* storage capacity *per day*
- Alibaba: in 2017, 320 PB of log data was generated in a six hour period as a result of customer purchase activity

- **Variety**

- Data are multi-modal

- **Velocity**

- Streaming data
- Facebook: 900 million photo uploads per day
- Alibaba: During 2017/11/11 470 million event logs per second

Big Data Characteristics – Four Vs

- **Volume**

- Almost every dataset we use is high volume and volume is increasing, but consider these
- Google: in 2016, user uploads to YouTube required 1PB of *new* storage capacity *per day*
- Alibaba: in 2017, 320 PB of log data was generated in a six hour period as a result of customer purchase activity

- **Variety**

- Data are multi-modal

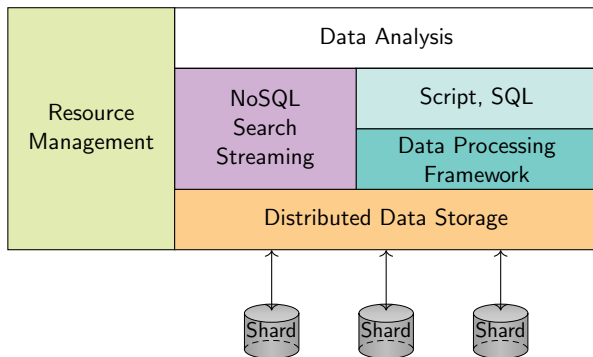
- **Velocity**

- Streaming data
- Facebook: 900 million photo uploads per day
- Alibaba: During 2017/11/11 470 million event logs per second

- **Veracity**

- Data quality & cleaning
- Managing uncertain data
- Search under uncertainty

Big Data Software Stack



- 1 Distributed Storage Systems
- 2 Main Memory Systems
- 3 Big Data Processing
 - MapReduce [Li et al., 2014]
 - Spark [Zaharia, 2016]
- 4 Stream Data Processing
- 5 Graph Processing
- 6 RDF Data Management

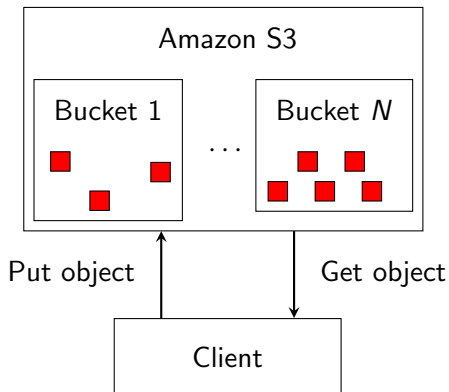
- 1 Distributed Storage Systems
- 2 Main Memory Systems
- 3 Big Data Processing
 - MapReduce [Li et al., 2014]
 - Spark [Zaharia, 2016]
- 4 Stream Data Processing
- 5 Graph Processing
- 6 RDF Data Management

- Two basic solutions to store big data
 - Objects
 - Files
- Different goals
- Complementary solutions

- Objectives
 - Very large amounts of unstructured data
 - Object transfer over a long-distance network
 - Access with REST Internet protocols
- Approach
 - An object includes its data, metadata and an identifier
 - Flat object space: no hierarchy of objects to manage
- OSD (object storage device): to store objects on disk

- Object = triple (oid, data, metadata)
- Oid: unique, permanent identifier
 - Once created, an object can be directly accessed by its oid
- Data: unstructured data (text, binary, ...)
 - Easy to move over the network
- Metadata: understand the data
 - Much flexibility in defining an object's properties
 - Protection, replication, lifetime, etc.

Amazon Simple Storage Service (S3)



- Online storage service
 - Simple and easy
 - Put/get object in a bucket (an online directory) using a unique key given by the user
 - Very big objects
 - Max size: 5TB
 - Get/put: 5GB
 - Access via REST or SOAP
- Use cases: images, videos

- Cloud services
 - Amazon S3
 - Rackspace Files
 - Microsoft Azure Vault Storage
 - Google Cloud Storage
- Enterprise systems
 - Swift, OpenStack.org
 - IBM Spectrum Scale Object
 - Quantum Lattus
 - Caringo Swarm
 - EMC Atmos
 - Hitachi HCP
 - Scality RING

- Objectives

- Storage of unstructured data (byte strings)
 - Can also be organized as fixed or variable size records
- Managed in a directory hierarchy
- Sharing in a data center
 - NAS (network attached storage) or SAN (storage area network)

- Approach

- Separation of data (file content) from metadata
 - First read the metadata (in the directory) to locate the content

- Objectives
 - Very big files, with many records
- Capabilities
 - Scalability
 - Fault-tolerance
- Approaches
 - Block-based: GFS
 - Object-based: Lustre
 - Hybrid: Ceph

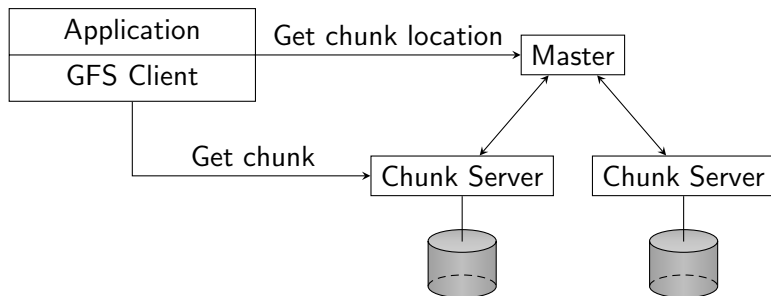
Google File System (GFS)

- Used by many Google applications
 - Search engine, Bigtable, Mapreduce, etc.
 - Proprietary
- The basis for open source solutions
 - Hadoop Distributed File System (HDFS)
- Objectives
 - Very big files, of typically several Terabytes, containing many elements such as web documents
 - Shared-nothing cluster of thousands of nodes, built from inexpensive hardware
 - Node failure is the norm, not the exception
 - Performance, fault-tolerance and high-availability
 - Using partitioning and replication

- Files are divided in fixed-size partitions, called **chunks**, of large size, i.e. 64 MB
 - Partitions are distributed across multiple nodes
 - Each partition is replicated at several nodes (2 by default)
- Optimized for read and append
 - Random updates are rare
 - Large reads of bulk data (e.g. 1 MB) and small random reads (e.g. 1 KB)
 - Append operations are also large and there may be many concurrent clients that append the same file
 - High throughput (for bulk data) more important than low latency

- Traditional file system interface (create, open, read, write, close, and delete file)
 - Two additional operations: snapshot and record append
 - No update: need to do read and write
- Relaxed consistency, with atomic record append
 - No need for distributed lock management
 - Up to the application to use techniques such as checkpointing and writing self-validating records
- Single GFS master
 - Maintains file metadata such as namespace, access control information, and data placement information
 - Simple, lightly loaded, fault-tolerant
- Fast recovery and replication strategies

GFS: Capabilities



- Object-based
- Used a lot in High Performance Computing (HPC) and scientific applications in the cloud
 - Intel Cloud Edition for Lustre
- 2 types of servers
 - Metadata servers
 - Object servers: store file contents in **object storage targets** (OSTs)
- Shared-disk cluster

Distributed File Systems

- Block-based
 - Google File System (GFS)
 - Hadoop Distributed File System (HDFS)
 - GlusterFS, Red Hat, for shared-nothing cluster
 - Global File System 2 (GFS2), Red Hat, for shared-disk cluster
- Object-based
 - Lustre, Intel
 - XtreamFS, Xtream European Project
- Hybrid
 - Ceph, Red Hat
 - Microsoft Azure HDInsight: integration of HDFS within Azure Blob Storage
 - Ozone, Hortonworks: extension of HDFS with object-based level

Outline

- 1 Distributed Storage Systems
- 2 Main Memory Systems
- 3 Big Data Processing
 - MapReduce [Li et al., 2014]
 - Spark [Zaharia, 2016]
- 4 Stream Data Processing
- 5 Graph Processing
- 6 RDF Data Management

Database Systems and Technological Changes

- Database systems were architected for the following typical environment:
 - 1-7 MIPS CPU with about 1KB cache
 - Less than 10MB RAM
 - About 80MB hard disk with 1MB/sec transfer rate
- But the technology has changed enormously. Each large server on our cluster:
 - Intel Xeon E5-4600 (Ivy Bridge) CPU: $> 80,000$ MIPS with 8 cores, 20MB L3 cache shared among cores, 256KB and 64KB per core of L2 and L1 caches
 - CPU speeds are now $\sim 300,000$ MIPS
 - 32GB RAM
 - Big servers can now have >1 TB of RAM
 - 1TB of hard disk with 6GB/s transfer rate
 - Hard disks of 6TB are coming onto market
- Database architectures have not changed since their introduction at the beginning of 1980s (until recently)

Typical DBMS Architecture

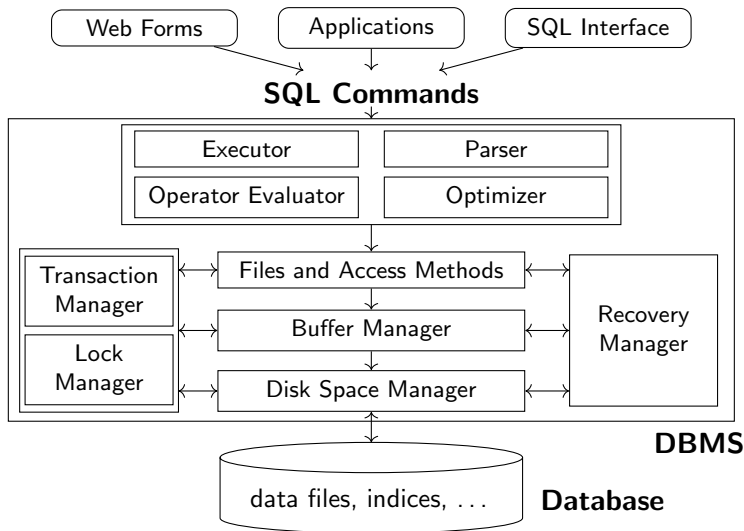
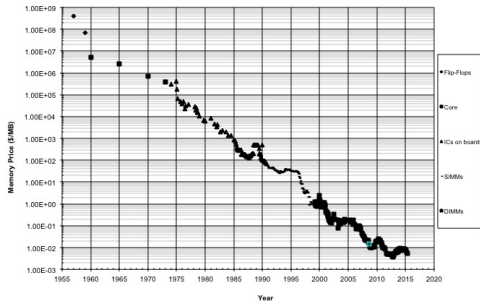
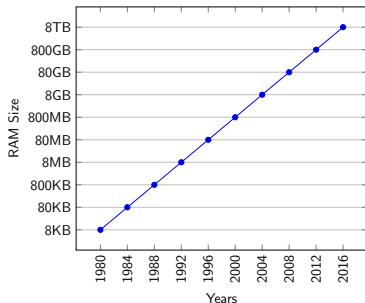
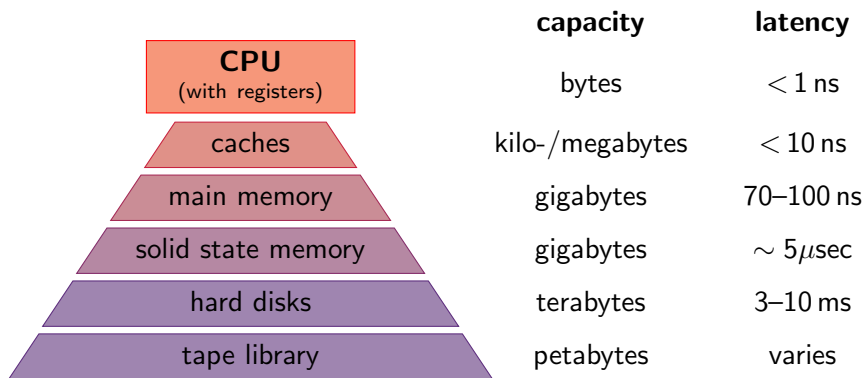


Figure inspired by Ramakrishnan/Gehrke: "Database Management Systems", McGraw-Hill 2003.

Recall the Developments in RAM



Memory Hierarchy



- Fast, but expensive and small, memory close to CPU
- Larger, slower memory at the periphery
- Increasing gap between CPU and memory speeds
- Latency is hidden by using the fast memory as a **cache**.

Main Memory (In-Memory) DBMS

Fundamental characteristic

Store the entire database in main memory for processing, possibly using the hard disk for persistence.

- Not a new idea – but increasing memory sizes allow accommodating bigger databases
- Usually goes with column-oriented storage, but not always (e.g., SQL Server Hekaton works on rows)
- Can target OLTP or OLAP workloads or both
- Requires rethinking of many functions, but there are simplifications

IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 4, NO. 6, DECEMBER 1992

391

Main Memory Database Systems: An Overview

Hector Garcia-Molina, Member, IEEE, and Kenneth Salem, Member, IEEE

Invited Paper

Abstract—Memory resident database systems (MMDBS) store their data in main physical memory and provide very high-speed access. Conventional database systems are optimized for the particular characteristics of disk storage mechanisms. Memory resident systems, on the other hand, use different optimizations to structure and organize data, as well as to make it reliable. This paper surveys the main memory resident optimizations and briefly discusses some of the memory resident systems that have been designed or implemented.

Index Terms—Access methods, application programming interface, concurrent processing, concurrency control, data clustering, data representation, main memory database systems (MMDBS), query processing, recovery.

1. INTRODUCTION

IN a main memory database system (MMDBS) data resides increasingly in main physical memory; in a conventional database system (CDBS) it is disk resident. In a CDBS, disk data may be cached into memory for access; in a MMDBS the memory resident data may have a backup copy on disk. So in both cases, a given object can have copies both in memory and on disk. The key difference is that in MMDBS the primary copy lives permanently in memory, and this has important implications (to be discussed) as to how it is structured and accessed.

As semiconductor memory becomes cheaper and chip densities increase, it becomes feasible to store larger and larger databases in memory, making MMDBS a reality. Because data can be accessed directly in memory, MMDBS can provide much better response times and transaction throughput, as compared to CDBS's. This is especially important for real-time applications where transactions have to be completed by their specified deadlines.

A computer's main memory clearly has different properties from that of magnetic disks, and these differences have profound implications on the design and performance of the database system. Although these differences are well known, it is worthwhile reviewing them briefly.

1) The access time for main memory is orders of magnitude less than for disk storage.

2) Main memory is normally volatile, while disk storage is not. However, it is possible (at some cost) to construct nonvolatile main memory.

3) Disks have a high fixed cost per access that does not depend on the amount of data that is retrieved during the access. For this reason, disks are effective external storage devices. Main memory is not disk oriented.

4) The layout of data on a disk is much more critical than the layout of data in main memory, since sequential access to a disk is faster than random access. Sequential access is not as important in main memory.

5) Main memory is normally directly accessible by the processor(s), while disks are not. This may make data in main memory more vulnerable than disk resident data to software errors.

These differences have effects on almost every aspect of database management, from concurrency control to application interfaces. In this paper we will discuss these effects and will briefly survey some MMDBS that have been recently designed or implemented. However, before getting started, it is important to address three questions that are commonly asked about MMDBS's.

It is reasonable to assume that the entire database fits in main memory? Yes, for some applications. In some cases, the database is of limited size, growing at a slower rate than memory capacities are growing. For example, the database size may be proportional to the number of employees or customers in a company, and so make slow successful the company is, it is reasonable to expect that memory can hold a few hundred or thousand bytes per employee or customer. In some real-time applications, the data must be memory resident to meet the real-time constraints, so the database will necessarily be smaller than the amount of available memory. Examples of such real-time applications include telecommunications (e.g., 800 telephone numbers need to be translated to real numbers), radar tracking (e.g., signatures of objects need to be matched against a database of known aircraft), and securities trading (e.g., trading opportunities must be discovered and executed before they vanish).

However, there are clearly cases where the database does not and will never fit in memory, e.g., an application with satellite image data. For such cases, CDBS will continue to be important. Nevertheless, even in these very large applications, it is common to find different classes of data: hot data that is accessed frequently, usually low volume and with stringent timing requirements; cold data that is accessed rarely and is more voluminous; and various intermediate categories. If this is

Manuscript received December 1, 1990; revised July 27, 1992. The work of H. Garcia-Molina was supported by the National Science Foundation under Grant EEC-880466 and by CR-88046. H. Garcia-Molina is with the Department of Computer Science, Stanford University, Stanford, CA 94305. K. Salem is with the Department of Computer Science, University of Maryland, College Park, MD 20742. IEEE Log Number 9200050.

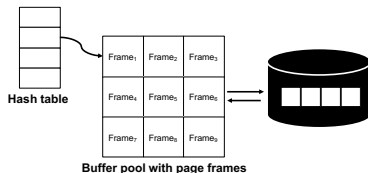
0893-4063/92/0004-0391\$04.00/0

Design Decisions

- Access patterns
 - Random access in memory is still more expensive than sequential access
 - Partly due to cache lines capacity and cache misses
- Main memory optimized storage
 - No page structure necessary
 - No secondary indexes, only main memory index
 - Try to use latch-free data structures
- Data organization
 - Partitioned vs non-partitioned vs multi-versioned vs row/column
- Concurrency
 - Lock managers are performance bottlenecks
 - Very lightweight locking or optimistic
- Durability
 - Checkpointing gives “light” durability
 - Transaction logging either to disk or non-volatile memory
 - Replication

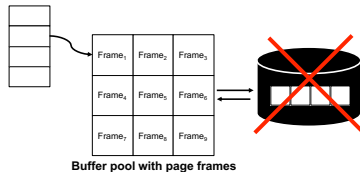
Data Organization in Disk-based Systems

- Classical disk-based relational systems page data to and from disk on demand (fixed-size blocks, e.g., 8KB)
- Storage engine manages buffer pool
 - Page frames are in-memory representation of page
 - Hash table maps page id to location in pool
 - If page is not in memory, disk I/O needed to populate a target page frame
- Logical record identifiers: (page id, offset)
- Elegant solution for abstracting disk away from other layers of the database stack
- Too much overhead for main-memory systems



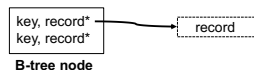
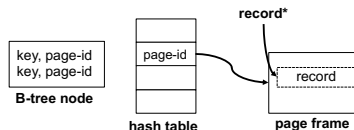
Data Organization in Main Memory Systems

- Data lives in RAM in main-memory databases; no need to page from disk



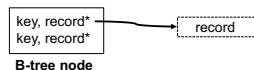
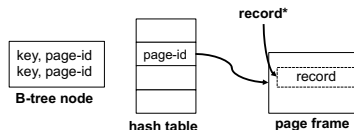
Data Organization in Main Memory Systems

- Data lives in RAM in main-memory databases; no need to page from disk
- Modern systems avoid page indirection through buffer pool
 - Avoid logical record identifiers: (page id, offset)
 - Avoid two-level indirection: Hash table access to resolve page frame in buffer pool and Calculate pointer to record using offset within page
 - Common practice to use in-memory pointers for direct record access



Data Organization in Main Memory Systems

- Data lives in RAM in main-memory databases; no need to page from disk
- Modern systems avoid page indirection through buffer pool
 - Avoid logical record identifiers: (page id, offset)
 - Avoid two-level indirection: Hash table access to resolve page frame in buffer pool and Calculate pointer to record using offset within page
 - Common practice to use in-memory pointers for direct record access
- Can result in order of magnitude performance improvements
 - Avoids page-based indirection to resolve record pointer
 - Avoids page latch overhead



Partitioned

Disjoint partitioning of the database, assign partition to node, core, etc.

- ▶ Simple per-node implementation (serial execution, no concurrent data structures)
- ▶ Partition management burden: rebalance for hotspots
- ▶ Unpredictable behavior if too many cross-partition transactions

Non-partitioned

Any thread/core can access any record in the database

- ▶ Self-balancing system: no need to worry about partition management
- ▶ Increased implementation complexity (e.g., concurrent data structures)

Multi-Versioned

- ▶ Allows for high concurrency; important in multi-core environments
- ▶ Readers allow to execute uncontested, do not block writers

Row/Columnar Layout

- ▶ Reasonable OLTP performance on in-memory columnar layout; not true for disk-based column stores

System Organization Choices

	Partitioned	Multi-Versioned	Row/Columnar
Hekaton	No	Yes	Row
HyPer	No	Yes	Hybrid
SAP HANA	No	Yes	Hybrid
H-Store/VoltDB	Yes	No	Row

Outline

- 1 Distributed Storage Systems
- 2 Main Memory Systems
- 3 Big Data Processing
 - MapReduce [Li et al., 2014]
 - Spark [Zaharia, 2016]
- 4 Stream Data Processing
- 5 Graph Processing
- 6 RDF Data Management

- For data analysis of very large data sets
 - Highly dynamic, irregular, schemaless, etc.
 - SQL too heavy
- “Embarrassingly parallel problems”
- New, simple parallel programming model
 - Data structured as (key, value) pairs
 - E.g. (doc-id, content), (word, count), etc.
 - Functional programming style with two functions to be given:
 - $\text{Map}(k1, v1) \rightarrow \text{list}(k2, v2)$
 - $\text{Reduce}(k2, \text{list}(v2)) \rightarrow \text{list}(v3)$
- Implemented on a distributed file system (e.g., Google File System) on very large clusters

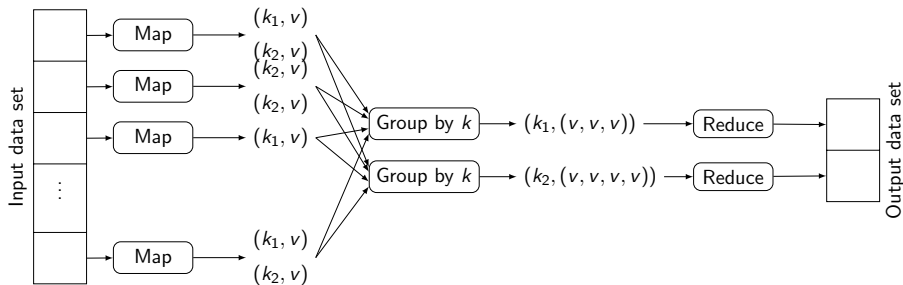
Map Function

- User-defined function
 - Processes input key/value pairs
 - Produces a set of *intermediate* key/value pairs
- Map function I/O
 - **Input:** read a *chunk* from distributed file system (DFS)
 - **Output:** Write to intermediate file on local disk
- MapReduce library
 - Executes map function
 - Groups together all intermediate values with the same key (i.e., generates a set of lists)
 - Passes these lists to reduce functions
- Effect of map function
 - Processes and partitions input data
 - Builds a distributed map (transparent to user)
 - Similar to “group by” operator in SQL

Reduce Function

- User-defined function
 - Accepts one intermediate key and a set of values for that key (i.e., a list)
 - Merges these values together to form a (possibly) smaller set
 - Typically, zero or one output value is generated per invocation
- Reduce function I/O
 - **Input:** read from intermediate files using remote reads on local files of corresponding mapper nodes
 - **Output:** Each reducee writes its output as a file back to DFS
- Effect of map function
 - Similar to aggregation operation in SQL

MapReduce Processing



Example 1

Assume you are reading the monthly average temperatures for each of the 12 months of a year for a bunch of cities, i.e., each input is the name of the city (key) and the average monthly temperature. Compute the average annual temperature for each city.

- Map:

Input: $\langle \text{City}, \text{Month}, \text{MonthAvgTemp} \rangle$

- ① Create key/value pairs

Output: $\langle \text{City}, \text{MonthAvgTemp} \rangle$

- Reduce:

Input: $\langle \text{City}, \text{MonthAvgTemp} \rangle$

- ① Sort to get $\langle \text{City}, \text{list}(\text{MonthAvgTemp}) \rangle$ (i.e., it combines in a list the monthly average temperatures for a given city)
- ② Compute average over $\text{list}(\text{MonthAvgTemp})$

Output: $\langle \text{City}, \text{AnnualAvgTemp} \rangle$

Example 2

- Consider EMP (ENAME, TITLE, CITY)
- Query:

```
SELECT      CITY, COUNT(*)  
FROM        EMP  
WHERE       ENAME LIKE "\%Smith"  
GROUP BY    CITY
```

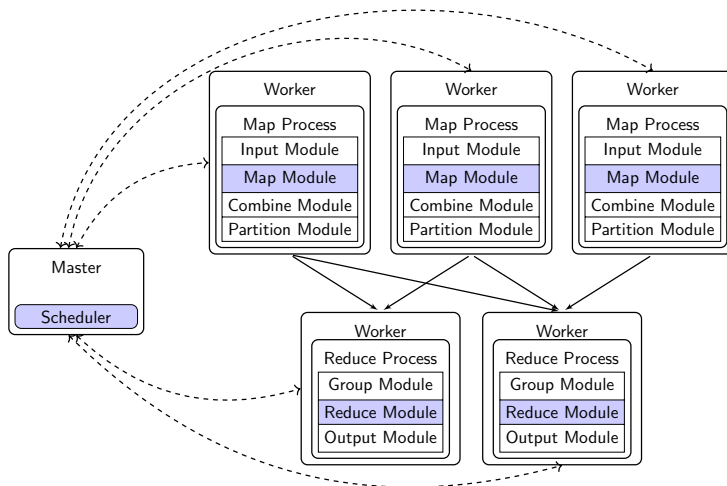
- Map:

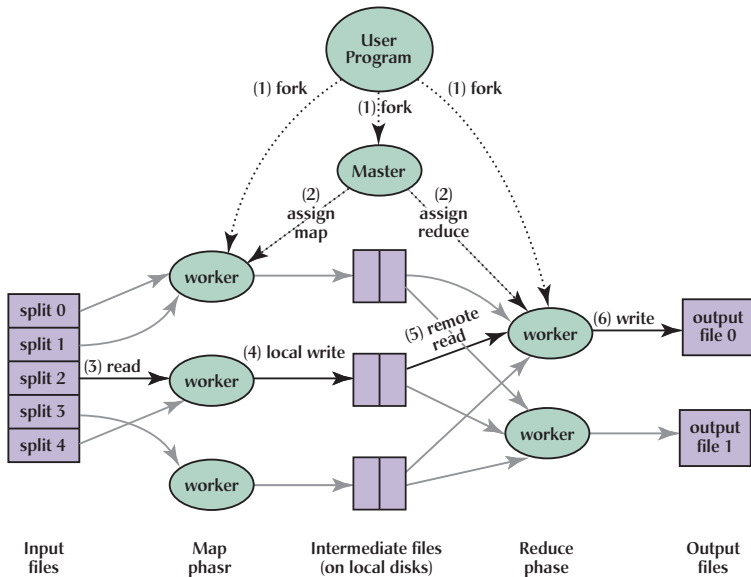
```
Input: (TID, emp), Output: (CITY, 1)  
    if emp.ENAME like "\%Smith" return (CITY, 1)
```

- Reduce:

```
Input: (CITY, list(1)), Output: (CITY, SUM(list(1)))  
    return (CITY, SUM(1*))
```

MapReduce Architecture



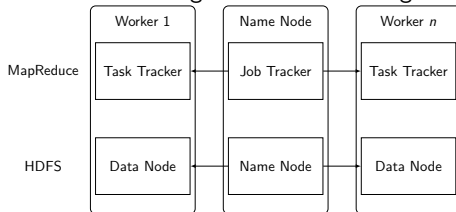


Characteristics

- Flexibility
 - User can write any `map` and `reduce` function code
 - No need to know how to parallelize
- Scalability
 - Elastic scalability
 - Automatic load balancing
- Efficiency
 - Simple: parallel scan
 - No database loading
- Fault tolerance
 - Worker failure
 - Master pings workers periodically; assumes failure if no response
 - Tasks (both `map` and `reduce`) on failed workers scheduled on a different worker node
 - Master failure
 - Checkpoints of master state
 - Recovery after failure → progress can halt
 - Replication on distributed data store

Hadoop

- Most popular MapReduce implementation – developed by Yahoo!
- Two components
 - Processing engine
 - HDFS: Hadoop Distributed Storage System – others possible
 - Can be deployed on the same machine or on different machines
- Processes
 - **Job tracker**: hosted on the master node and implements the schedule
 - **Task tracker**: hosted on the worker nodes and accepts tasks from job tracker and executes them
- HDFS
 - **Name node**: stores how data are partitioned, monitors the status of data nodes, and data dictionary
 - **Data node**: Stores and manages *data chunks* assigned to it



Hadoop UDF Functions

Phase	Name	Function
Map	InputFormat::getSplit	Partition the input data into different splits. Each split is processed by a mapper and may consist of several chunks.
	RecordReader::next	Define how a split is divided into items. Each item is a key/value pair and used as the input for the <code>map</code> function.
	Mapper::map	Users can customize the <code>map</code> function to process the input data. The input data are transformed into some intermediate key/value pairs.
	WritableComparable::compareTo	The comparison function for the key/-value pairs.
	Job::setCombinerClass	Specify how the key/value pair are aggregated locally.
Shuffle	Job::setPartitionerClass	Specify how the intermediate key/value pairs are shuffled to different reducers.
Reduce	Job::setGroupingComparatorClass	Specify how the key/value pairs are grouped in the reduce phase.
	Reducer::reduce	Users write their own <code>reduce</code> functions to perform the corresponding jobs.

MapReduce Implementations

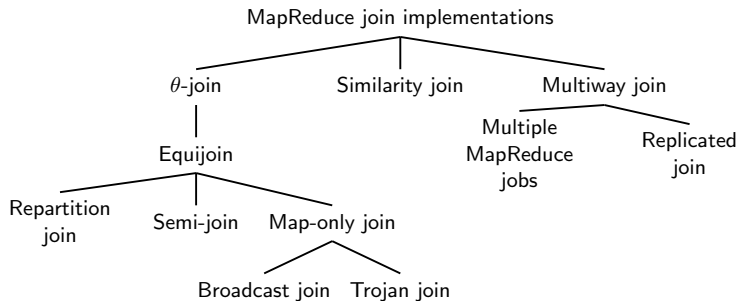
Name	Language	File System	Index	Master Server	Multiple Job Support
Hadoop	Java	HDFS	No	Name Node and Job Tracker	Yes
Disco	Python and Erlang	Distributed Index	Disco Server	No	No
Skynet	Ruby	MySQL or Unix File System	No	Any node in the cluster	No
FileMap	Shell and Perl Scripts	Unix File System	No	Any node in the cluster	No
Twister	Java	Unix File System	No	One master node in broker network	Yes
Cascading	Java	HDFS	No	Name Node and Job Tracker	Yes

MapReduce Languages

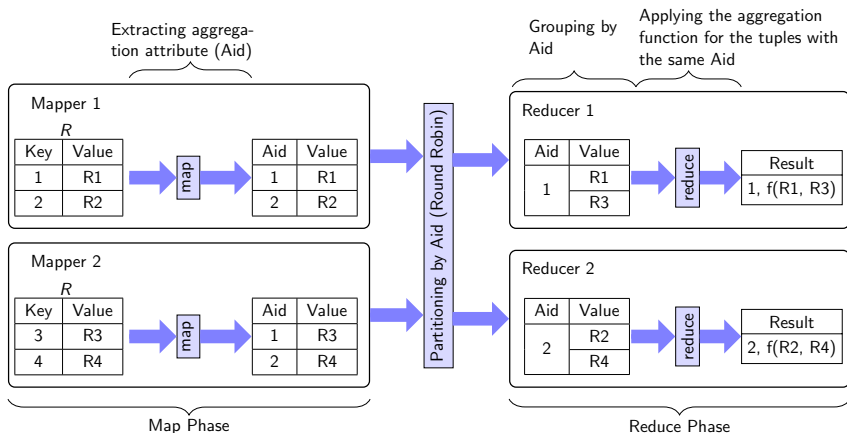
	Hive	Pig
Language	Declarative SQL-like	Dataflow
Data model	Nested	Nested
UDF	Supported	Supported
Data partition	Supported	Not supported
Interface	Command line, web, JDBC/ODBC server	Command line
Query optimization	Rule based	Rule based
Metastore	Supported	Not supported

MapReduce Implementations of Database Operators

- Select and Project can be easily implemented in the map function
- Aggregation is not difficult (see next slide)
- Join requires more work



Aggregation



Baseline implementation of $R(A, B) \bowtie S(B, C)$

- 1 Partition R and assign each partition to mappers
- 2 Each mapper takes $\langle a, b \rangle$ tuples and converts them to a list of key/value pairs of the form $(b, \langle a, R \rangle)$
- 3 Each reducer pulls the pairs with the same key
- 4 Each reducer joins tuples of R with tuples of S

DBMS on MapReduce

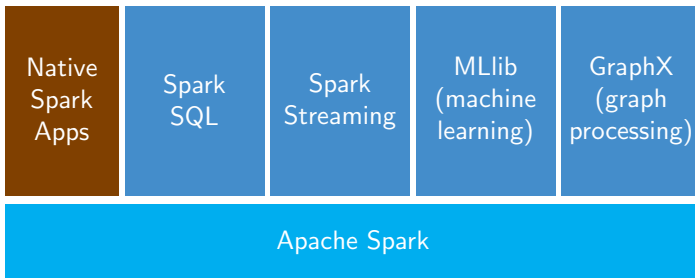
	HadoopDB	Llama	Cheetah
Language	SQL-like	Simple interface	SQL
Storage	Row store	Column store	Hybrid store
Data compression	No	Yes	Yes
Data partition	Horizontally partitioned	Vertically partitioned	Horizontally partitioned at chunk level
Indexing	Local index in each database instance	No index	Local index for each data chunk
Query optimization	Rule based optimization plus local optimization by PostgreSQL	Column-based optimization, late materialization and processing multiway join in one job	Multi-query optimization, materialized views

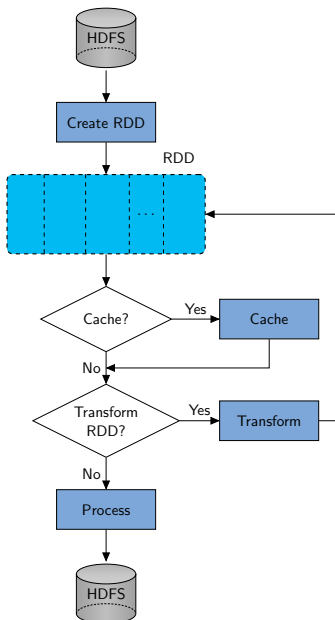
- MapReduce does not perform well in **iterative** computations
 - Workflow model is acyclic
 - Have to write to HDFS after each iteration and have to read from HDFS at the beginning of next iteration

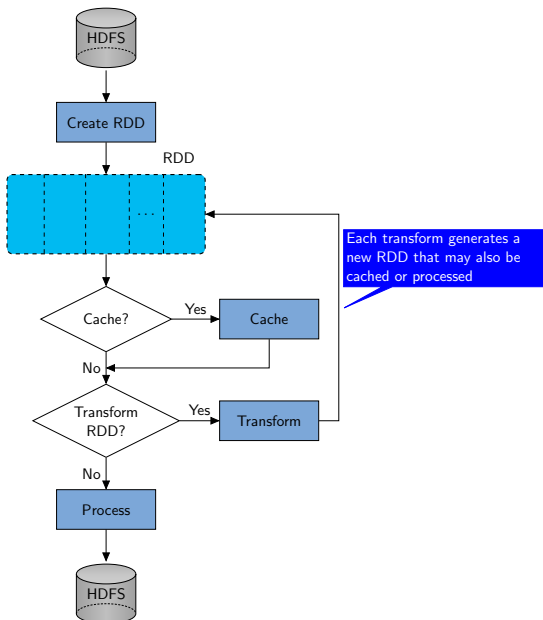
- MapReduce does not perform well in **iterative** computations
 - Workflow model is acyclic
 - Have to write to HDFS after each iteration and have to read from HDFS at the beginning of next iteration
- Spark objectives
 - Better support for iterative programs
 - Provide a complete ecosystem
 - Similar abstraction (to MapReduce) for programming
 - Maintain MapReduce fault-tolerance and scalability

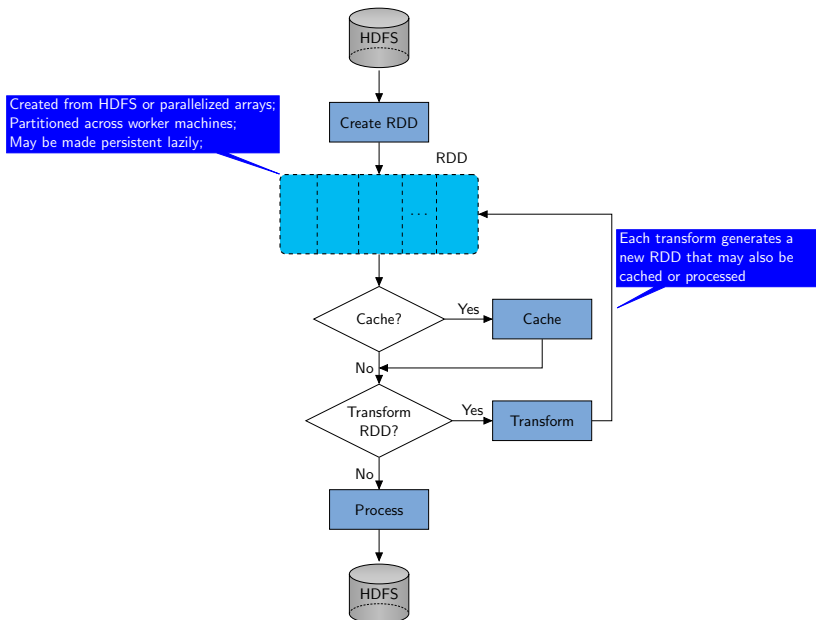
- MapReduce does not perform well in **iterative** computations
 - Workflow model is acyclic
 - Have to write to HDFS after each iteration and have to read from HDFS at the beginning of next iteration
- Spark objectives
 - Better support for iterative programs
 - Provide a complete ecosystem
 - Similar abstraction (to MapReduce) for programming
 - Maintain MapReduce fault-tolerance and scalability
- Fundamental concepts
 - RDD: Reliable Distributed Datasets
 - Caching of working set
 - Maintaining lineage for fault-tolerance

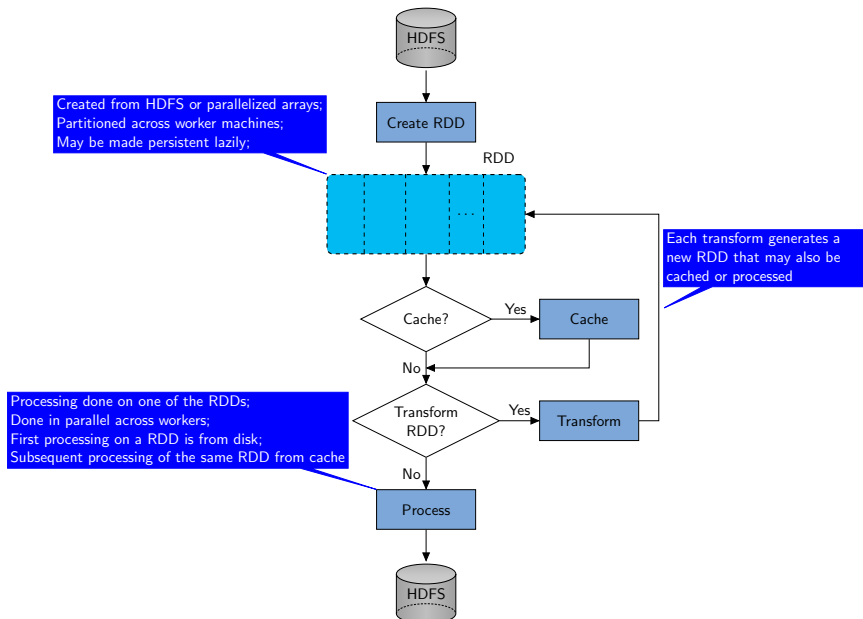
General Batching	Specialized Systems			
	Streaming	Iterative	Ad hoc/SQL	Graph
MapReduce	Storm	Mahout	Pig	Giraph
	S4		Hive	
	Samza		Drill	
			Impala	



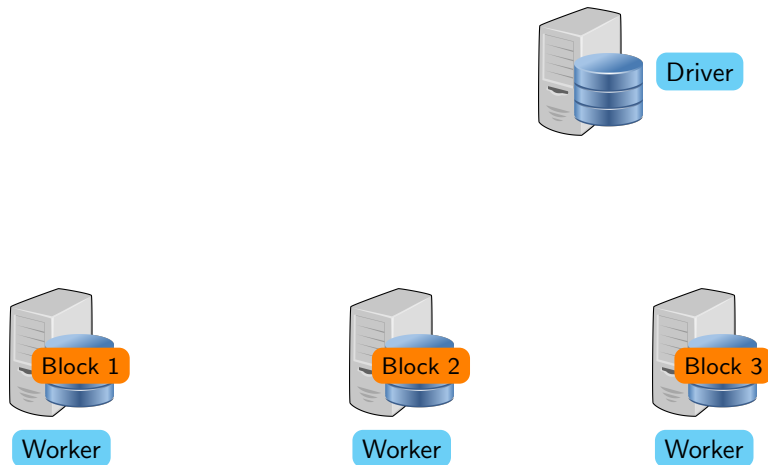








Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns



Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns

```
lines = spark.textFile(hdfs://...)
```

CreateRDD



Worker



Worker



Worker

Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns

```
lines = spark.textFile(hdfs://...)
errors = lines.filter(_.startsWith("ERROR"))
```

Transform RDD



Driver



Worker



Worker



Worker

Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns

```
lines = spark.textFile(hdfs://...)
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('t ')(2))
```

Another transform



Driver



Worker



Worker



Worker

Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns

```
lines = spark.textFile(hdfs://...)
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split("\t", 2))
cachedMsgs = messages.cache()
```

Cache results



Worker



Worker



Worker

Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns

```
lines = spark.textFile(hdfs://...)
errors = lines.filter(_.startsWith(\ERROR"))
messages = errors.map(_.split('\t ')(2))
cachedMsgs = messages.cache()

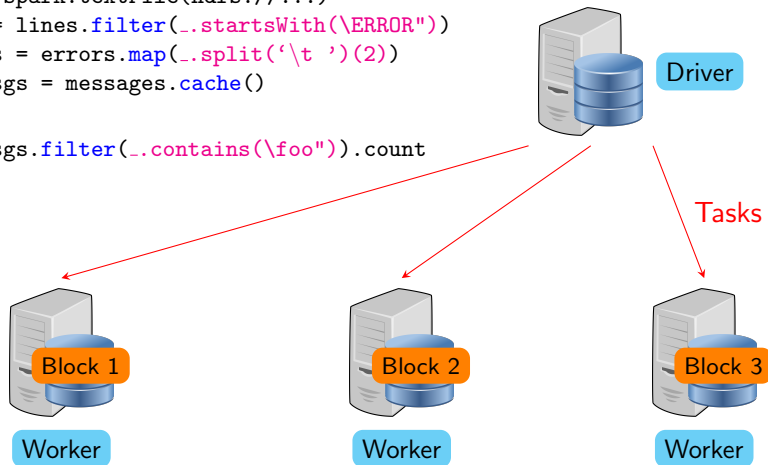
cachedMsgs.filter(_.contains(\foo")).count
```



Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns

```
lines = spark.textFile(hdfs://...)
errors = lines.filter(_.startsWith(ERROR))
messages = errors.map(_.split('t ')(2))
cachedMsgs = messages.cache()
```

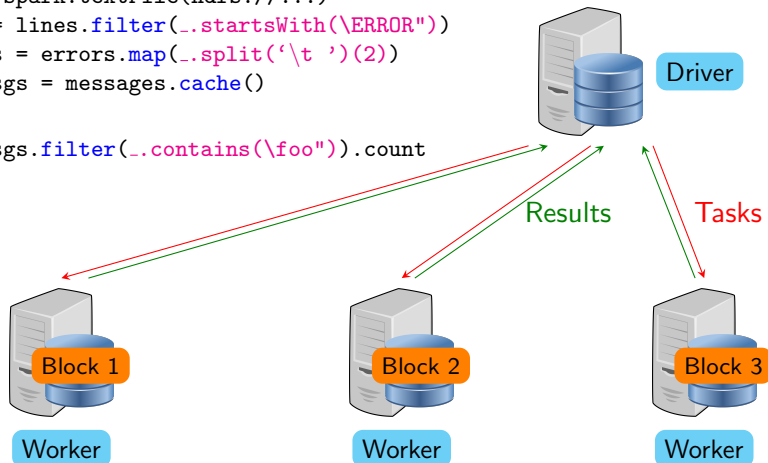
```
cachedMsgs.filter(_.contains(foo)).count
```



Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns

```
lines = spark.textFile(hdfs://...)
errors = lines.filter(_.startsWith(ERROR))
messages = errors.map(_.split('t ')(2))
cachedMsgs = messages.cache()
```

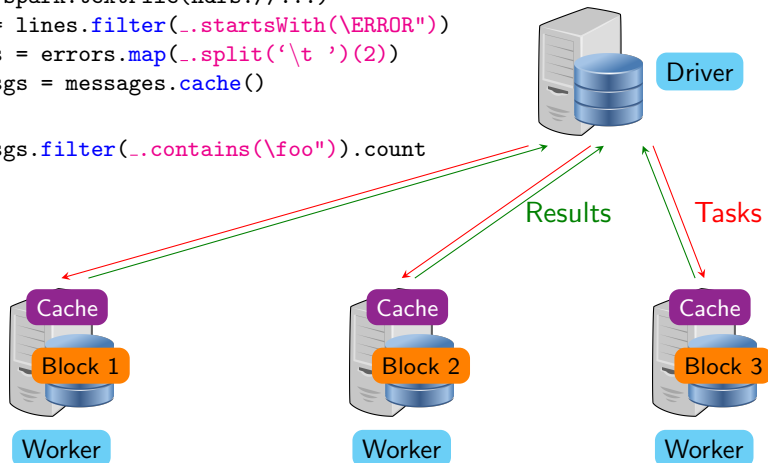
```
cachedMsgs.filter(_.contains(foo)).count
```



Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns

```
lines = spark.textFile(hdfs://...)
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('t ')(2))
cachedMsgs = messages.cache()
```

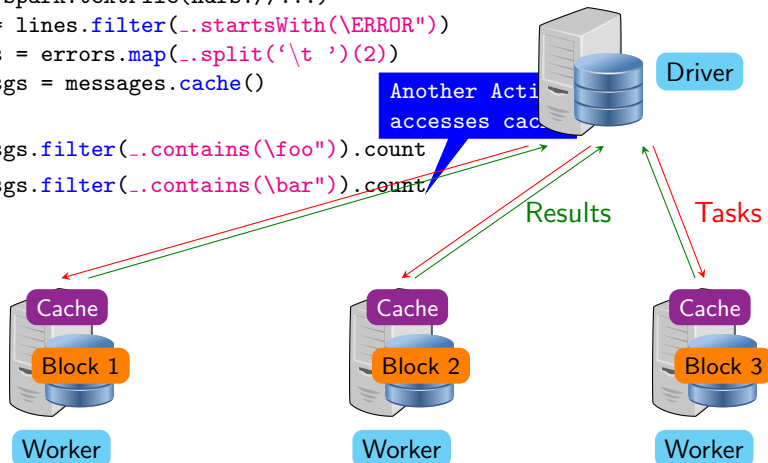
```
cachedMsgs.filter(_.contains("foo")).count
```



Load log messages from a file system, create a new file by filtering the error messages, read this file into memory, then interactively search for various patterns

```
lines = spark.textFile(hdfs://...)
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('t ')(2))
cachedMsgs = messages.cache()
```

```
cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
```



Example Transform Functions

Function	Meaning
<code>map(func)</code>	Return a new RDD formed by passing each element of the source through a function <i>func</i>
<code>filter(func)</code>	Return a new RDD formed by selecting those elements of the source on which <i>func</i> returns true
<code>flatMap(func)</code>	Similar to map, but each input item can be mapped to 0 or more output items (<i>func</i> should return a Seq not a single item)
<code>mapPartitions(func)</code>	Similar to map, but runs separately on each partition (block) of the RDD, so <i>func</i> must be of type Iterator
<code>sample(repl, fraction, seed)</code>	Sample a fraction <i>fraction</i> of the data, with or without replacement (set <i>repl</i> accordingly), using a given random number generator seed
<code>union(otherDataset)</code> <code>intersection()</code>	Return a new RDD that contains the union/intersection of the elements in the source RDD and the argument
<code>groupByKey()</code>	Operates on a RDD of (K, V) pairs, returns a RDD of (K, Iterable<V>) pairs
<code>reduceByKey(func, ...)</code>	Operates on a RDD of (K, V) pairs, returns a RDD of (K, V) pairs where the values for each key are aggregated using the given reduce function <i>func</i>

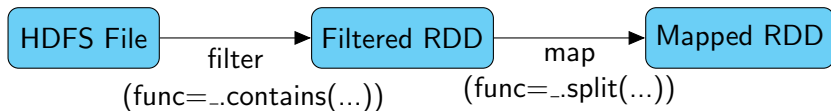
Example Actions

Action	Meaning
<code>collect()</code>	Return all the elements of the dataset as an array at the driver program.
<code>count()</code>	
<code>take(<i>n</i>)</code>	Return an array with the first <i>n</i> elements of the dataset
<code>first(<i>func</i>)</code>	Return the first element of the dataset (similar to <code>take(1)</code>)
<code>takeSample(<i>repl</i>, <i>num</i>, [<i>seed</i>])</code>	Return an array with a random sample of <i>num</i> elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed
<code>takeOrdered(<i>n</i>, [<i>ordering</i>])</code>	Return the first <i>n</i> elements of the RDD using either their natural order or a custom comparator
<code>saveAsTextFile(<i>path</i>)</code>	Write the elements of the RDD as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system
<code>countByKey()</code>	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key
<code>foreach(<i>func</i>)</code>	Run a function <i>func</i> on each element of the dataset

RDD Fault Tolerance

- RDDs maintain lineage information that can be used to reconstruct lost partitions
- Lineage is constructed as an object and stored persistently for recovery
- Example

```
messages = spark.textFile(\hdfs://...")  
    .lines.filter(_.startsWith(\ERROR"))  
    .errors.map(_split('\t ')(2))
```



Outline

- 1 Distributed Storage Systems
- 2 Main Memory Systems
- 3 Big Data Processing
 - MapReduce [Li et al., 2014]
 - Spark [Zaharia, 2016]
- 4 Stream Data Processing
- 5 Graph Processing
- 6 RDF Data Management

Stream System Inputs & Outputs

Inputs

One or more sources generate data continuously, in real time, and in fixed order (by timestamp)

- ▶ Sensor networks – weather monitoring, road traffic monitoring
- ▶ Web data – financial trading, news/sports tickers
- ▶ Scientific data – experiments in particle physics
- ▶ Transaction logs – point-of-sale purchases
- ▶ Network traffic analysis – IP packet headers

Stream System Inputs & Outputs

Inputs

One or more sources generate data continuously, in real time, and in fixed order (by timestamp)

- ▶ Sensor networks – weather monitoring, road traffic monitoring
- ▶ Web data – financial trading, news/sports tickers
- ▶ Scientific data – experiments in particle physics
- ▶ Transaction logs – point-of-sale purchases
- ▶ Network traffic analysis – IP packet headers

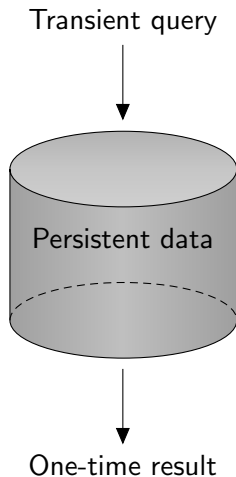
Outputs

Want to collect and process data in real-time; up-to-date answers generated continuously or periodically

- ▶ Environment monitoring
- ▶ Location monitoring
- ▶ Correlations across stock prices
- ▶ Denial-of-service attack detection

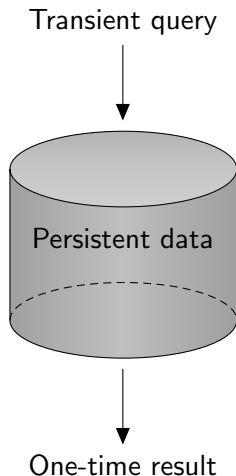
DBMS versus DSS

Traditional DBMS:

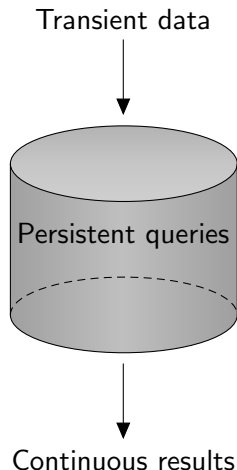


DBMS versus DSS

Traditional DBMS:

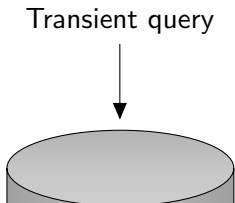


Data Stream System (DSS):

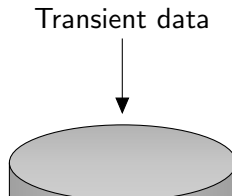


DBMS versus DSS

Traditional DBMS:



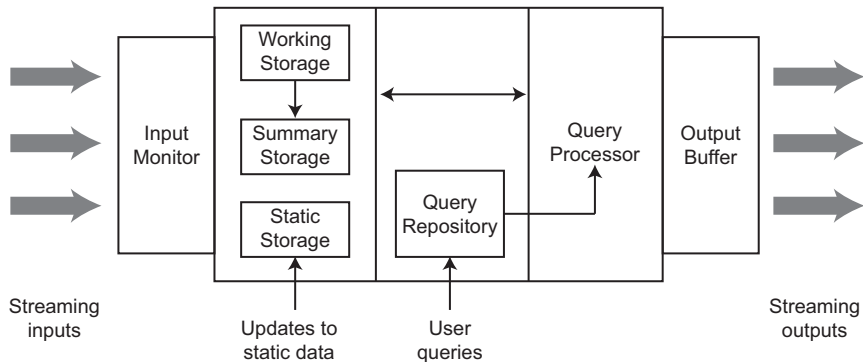
Data Stream System (DSS):



Other differences of DSS

- ▶ Push-based (data-driven)
- ▶ Unbounded stream; query execution as data arrives at the system – one look
- ▶ System conditions may not be stable – arrival rates fluctuate, workload may change

DSS System Architecture



Continuous processing

Each new data item is processed as soon as it arrives in the system.

Examples: Apache Storm, Heron

Processing Models

Continuous processing

Each new data item is processed as soon as it arrives in the system.

Examples: Apache Storm, Heron

Windowed processing

Incoming data items are batched in windows and executed as a batch.

For user, recently arrived data may be more interesting and useful.

Examples: Aurora, STREAM, Spark Streaming

Append-only sequence of timestamped items that arrive in some order.

$\langle \text{timestamp}, \text{payload} \rangle$

What is the payload?

- Relational tuples
- Revision tuples
- Sequence of events (as in publish/subscribe systems)
- Sequence of sets (or bags) of elements with each set storing elements that have arrived during the same unit of time

Window Definition

According to the direction of endpoint movement

- ▶ Fixed window: both endpoints are fixed
- ▶ Sliding window: both endpoints can slide (backward or forward)
- ▶ Landmark window: one endpoint fixed, the other sliding

Window Definition

According to the direction of endpoint movement

- ▶ Fixed window: both endpoints are fixed
- ▶ Sliding window: both endpoints can slide (backward or forward)
- ▶ Landmark window: one endpoint fixed, the other sliding

According to definition of window size

- ▶ Logical window: time-based – window length measured in time units
- ▶ Physical window: count-based – window length measured in number of data items
- ▶ Partitioned window: split a window into multiple count-based windows
- ▶ Predicate window: arbitrary predicate defines the contents of the window

Window Definition

According to the direction of endpoint movement

- ▶ Fixed window: both endpoints are fixed
- ▶ Sliding window: both endpoints can slide (backward or forward)
- ▶ Landmark window: one endpoint fixed, the other sliding

According to definition of window size

- ▶ Logical window: time-based – window length measured in time units
- ▶ Physical window: count-based – window length measured in number of data items
- ▶ Partitioned window: split a window into multiple count-based windows
- ▶ Predicate window: arbitrary predicate defines the contents of the window

Most common ones

- ▶ Time-based sliding window
- ▶ Count-based sliding window

Stream Query Models

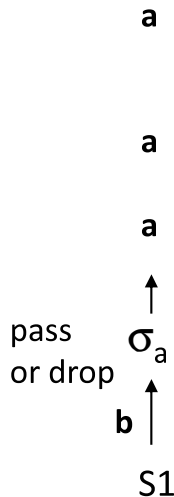
- Recall that queries are persistent
- They may be **monotonic** or **non-monotonic**
- Monotonic: result set always grows

If $Q(t)$ is the result of a query at time t , given two executions at time t_i and t_j , $Q(t_i) \subseteq Q(t_j)$ for all $t_i > t_j$

- Non-monotonic: deletions from the result set is possible

Streaming Operators

- Stateless operators are no problem:
e.g., selection, projection



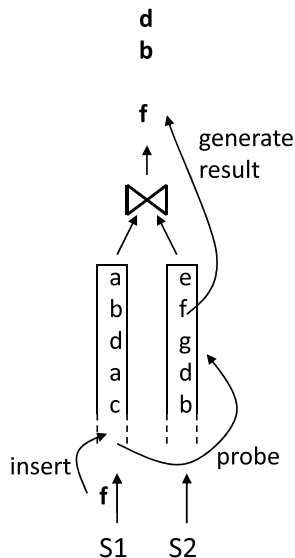
Streaming Operators

- Stateless operators are no problem:
e.g., selection, projection
- Stateful operators (e.g., nested loop join) are **blocking**
 - You need to see the entire inner operand

```
for each tuple  $r \in R$  do  
  for each tuple  $s \in S$  do  
    if  $r == s$  then  
      add  $\langle r, s \rangle$  to result  
    end for  
  end for
```

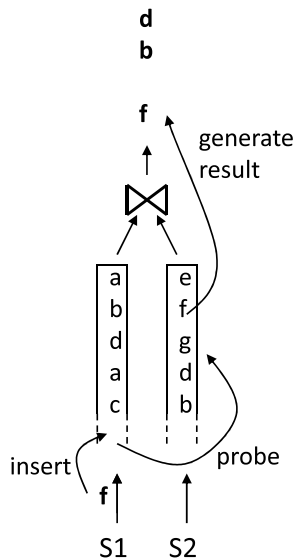
Streaming Operators

- Stateless operators are no problem:
e.g., selection, projection
- Stateful operators (e.g., nested loop join) are **blocking**
 - You need to see the entire inner operand
- For some blocking operators, non-blocking versions exist
 - Symmetric hash join



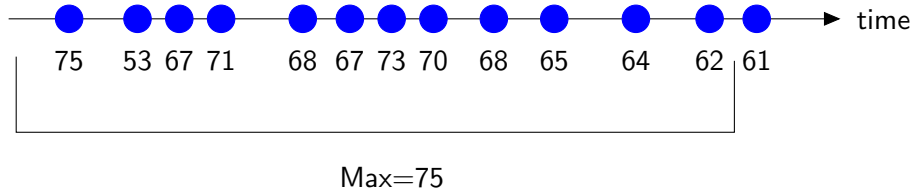
Streaming Operators

- Stateless operators are no problem:
e.g., selection, projection
- Stateful operators (e.g., nested loop join) are **blocking**
 - You need to see the entire inner operand
- For some blocking operators, non-blocking versions exist
 - Symmetric hash join
- Windowed execution otherwise



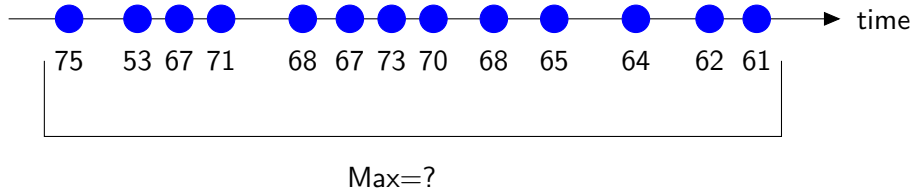
Operators over Sliding Windows

- Joins and aggregation may require unbounded state, so they typically operate over sliding windows
- Example: track the maximum value in an on-line sequence over a sliding window of the last N time units



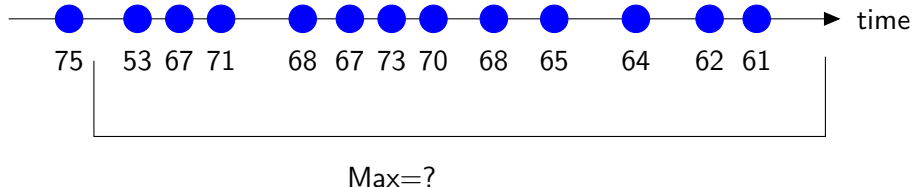
Operators over Sliding Windows

- Joins and aggregation may require unbounded state, so they typically operate over sliding windows
- Example: track the maximum value in an on-line sequence over a sliding window of the last N time units



Operators over Sliding Windows

- Joins and aggregation may require unbounded state, so they typically operate over sliding windows
- Example: track the maximum value in an on-line sequence over a sliding window of the last N time units



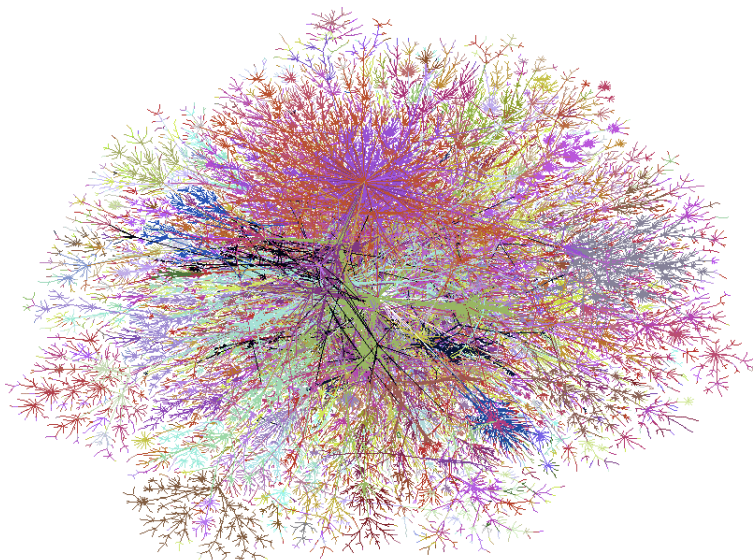
What to do when the data arrival rate is higher than the system's processing capacity?

- Random load shedding
 - Randomly drop a fraction of arriving data items
- Semantic load shedding
 - Examine the payload (contents) of a data item before deciding whether or not to drop it
 - Some data items may have more value than others
- Or, rather than dropping tuples:
 - Spill to disk and process during idle times
 - Shorten the windows
 - Update the answer less often

Outline

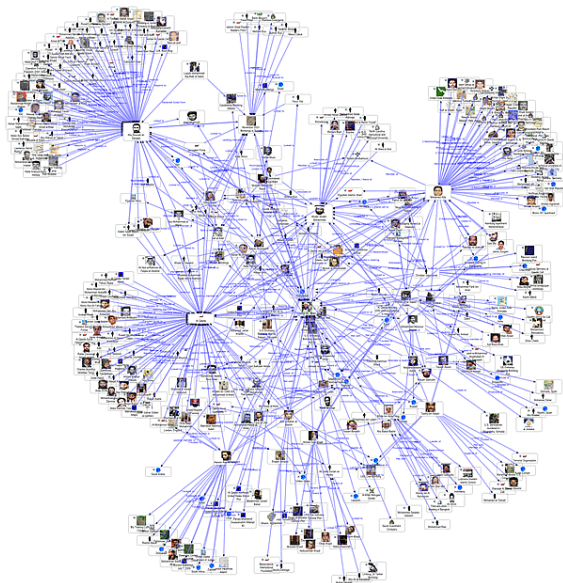
- 1 Distributed Storage Systems
- 2 Main Memory Systems
- 3 Big Data Processing
 - MapReduce [Li et al., 2014]
 - Spark [Zaharia, 2016]
- 4 Stream Data Processing
- 5 Graph Processing
- 6 RDF Data Management

Graph Data are Very Common



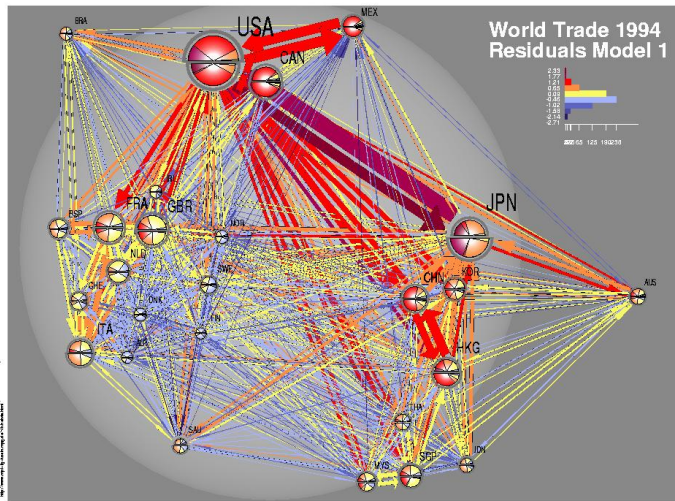
Internet

Graph Data are Very Common

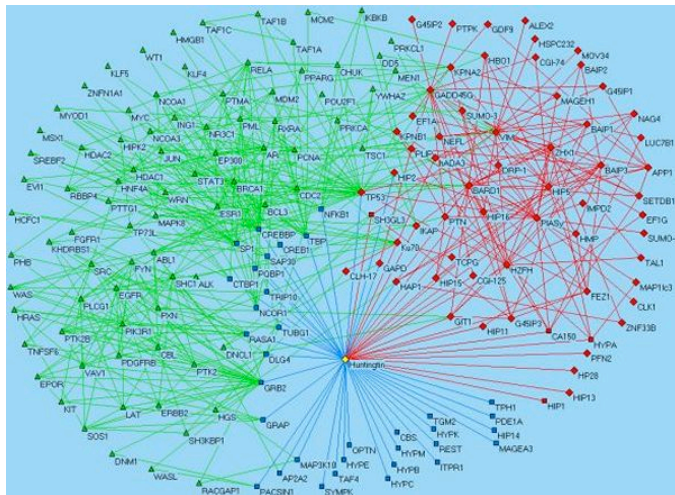


Social
networks

Graph Data are Very Common

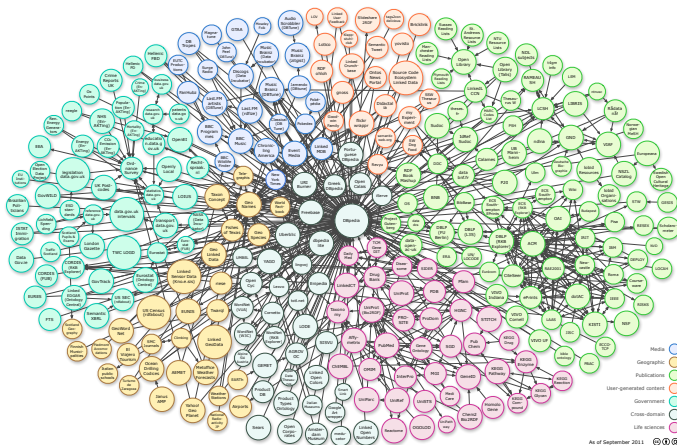


Graph Data are Very Common



Biological
networks

Graph Data are Very Common

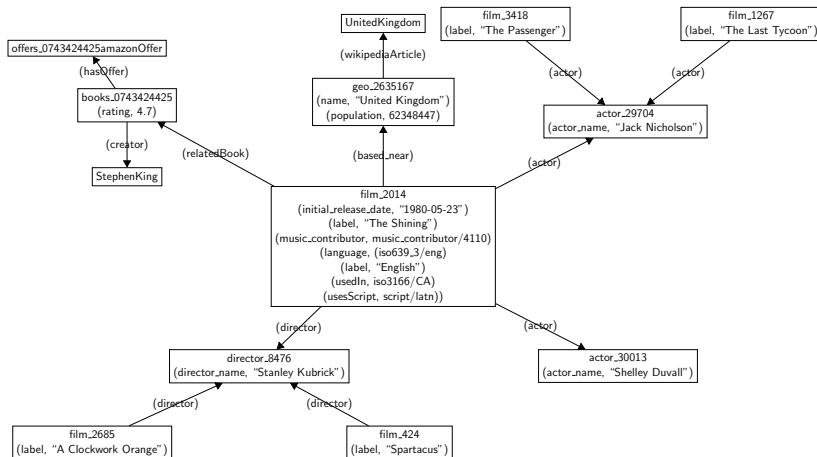


Linked data

Linking Open Data cloud diagram, by Richard Cyganiak and Anja Jentzsch.
<http://lod-cloud.net/>

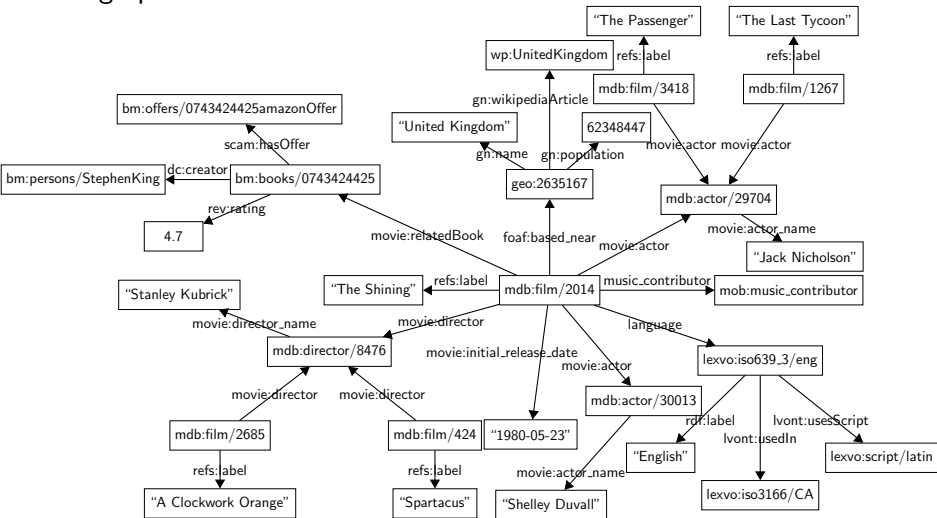
Graph Types

Property graph



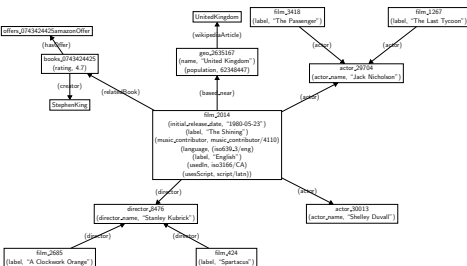
Graph Types

RDF graph



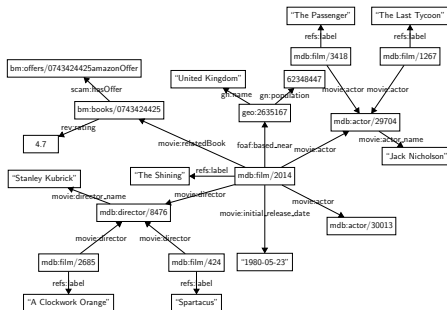
Graph Types

Property graph



- Workload: Online queries and analytic workloads
- Query execution: Varies

RDF graph



- Workload: SPARQL queries
- Query execution: subgraph matching by homomorphism

Property Graph Processing Systems

System	Memory/ Disk	Architecture	Computing paradigm	Supported Workloads
Hadoop	Disk	Parallel/Distributed	MapReduce	Analytical
Haloop	Disk	Parallel/Distributed	MapReduce	Analytical
Pegasus	Disk	Parallel/Distributed	MapReduce	Analytical
GraphX	Disk	Parallel/Distributed	MapReduce (Spark)	Analytical
Pregel/Giraph	Memory	Parallel/Distributed	Vertex-Centric	Analytical
GraphLab	Memory	Parallel/Distributed	Vertex-Centric	Analytical
GraphChi	Disk	Single machine	Vertex-Centric	Analytical
Stream	Disk	Single machine	Edge-Centric	Analytical
Trinity	Memory	Parallel/Distributed	Flexible using K-V store on DSM	Online & Analytical
Titan	Disk	Parallel/Distributed	K-V store (Cassandra)	Online
Neo4J	Disk	Single machine	Procedural/ Linked-list	Online

- Scale-up: Single machine execution
 - Graph datasets are small and can fit in a single machine – even in main memory
 - Single machine avoids parallel execution complexities

- Scale-up: Single machine execution
 - Graph datasets are small and can fit in a single machine – even in main memory
 - Single machine avoids parallel execution complexities
- Scale-out: Parallel execution
 - Graph data sets grow when they are expanded to their storage formats

Dataset	$ V $	$ E $	Regular size	Single Machine*
Live Journal	4,847,571	68,993,773	1.08GB	6.3GB
USA Road	23,947,347	58,333,344	951MB	9.09GB
Twitter	41,652,230	1,468,365,182	26GB	128 GB
UK0705	82,240,700	2,829,101,180	48GB	247GB
World Road	682,496,072	717,016,716	15GB	194GB
CommonCrawl2014	1,727,000,000	64,422,000,000	1.3TB	Out of memory

* Using (PowerLyra)

- Scale-up: Single machine execution
 - Graph datasets are small and can fit in a single machine – even in main memory
 - Single machine avoids parallel execution complexities
- Scale-out: Parallel execution
 - Graph data sets grow when they are expanded to their storage formats
 - Workstations big enough to handle even smaller datasets are still expensive
 - Some graphs are **very large**: Alibaba: several billion vertices, > 100 million edges
 - Dataset size may not be the determinant $\Rightarrow$ parallelizing computation is important

- Scale-up: Single machine execution
 - Graph datasets are small and can fit in a single machine – even in main memory
 - Single machine avoids parallel execution complexities
- Scale-out: Parallel execution
 - Graph data sets grow when they are expanded to their storage formats
 - Workstations big enough to handle even smaller datasets are still expensive
 - Some graphs are **very large**: Alibaba: several billion vertices, > 100 million edges
 - Dataset size may not be the determinant $\Rightarrow$ parallelizing computation is important

We focus on parallel graph analytics systems

Graph Partitioning

- Edge-cut (vertex-disjoint)
 - Achieve disjoint partitions by allocating each **vertex** to a partition
 - Objective 1: Partitions should be balanced
 - Objective 2: Minimize edge-cuts (to reduce communication)
 - Good for graphs with low-degree vertices, not for power-law graphs
 - Examples: Hashing, METIS [Karypis and Kumar, 1995], label propagation algorithms [Ugander and Backstrom, 2013]

Graph Partitioning

- Edge-cut (vertex-disjoint)

- Achieve disjoint partitions by allocating each **vertex** to a partition
- Objective 1: Partitions should be balanced
- Objective 2: Minimize edge-cuts (to reduce communication)
- Good for graphs with low-degree vertices, not for power-law graphs
- Examples: Hashing, METIS [Karypis and Kumar, 1995], label propagation algorithms [Ugander and Backstrom, 2013]

- Vertex-cut (edge-disjoint)

- Achieve disjoint partitions by allocating each **edge** to a partition and replicating vertices as necessary
- Objective 1: Partitions should be balanced
- Objective 2: Minimize vertex-cuts (to reduce replica cost)
- Perform better on power-law graphs
- Examples: Hashing, greedy algorithms [Gonzalez et al., 2012]

Graph Partitioning

- Edge-cut (vertex-disjoint)

- Achieve disjoint partitions by allocating each **vertex** to a partition
- Objective 1: Partitions should be balanced
- Objective 2: Minimize edge-cuts (to reduce communication)
- Good for graphs with low-degree vertices, not for power-law graphs
- Examples: Hashing, METIS [Karypis and Kumar, 1995], label propagation algorithms [Ugander and Backstrom, 2013]

- Vertex-cut (edge-disjoint)

- Achieve disjoint partitions by allocating each **edge** to a partition and replicating vertices as necessary
- Objective 1: Partitions should be balanced
- Objective 2: Minimize vertex-cuts (to reduce replica cost)
- Perform better on power-law graphs
- Examples: Hashing, greedy algorithms [Gonzalez et al., 2012]

- Hybrid

- Edge-cut for low-degree vertices/vertex-cut for high-degree ones
- PowerLyra [Chen et al., 2015]

Graph Workloads

Online graph querying

- Reachability
- Single source shortest-path
- Subgraph matching
- SPARQL queries

Offline graph analytics

- PageRank
- Clustering
- Connected components
- Diameter finding
- Graph colouring
- All pairs shortest path
- Graph pattern mining
- Machine learning algorithms
(Belief propagation, Gaussian non-negative matrix factorization)

Graph Workloads

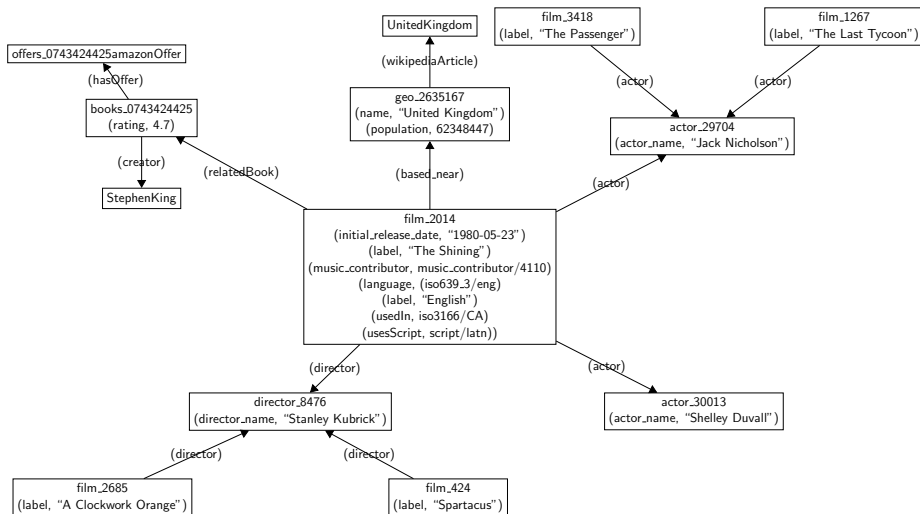
Online graph querying

- Reachability
- Single source shortest-path
- Subgraph matching
- SPARQL queries

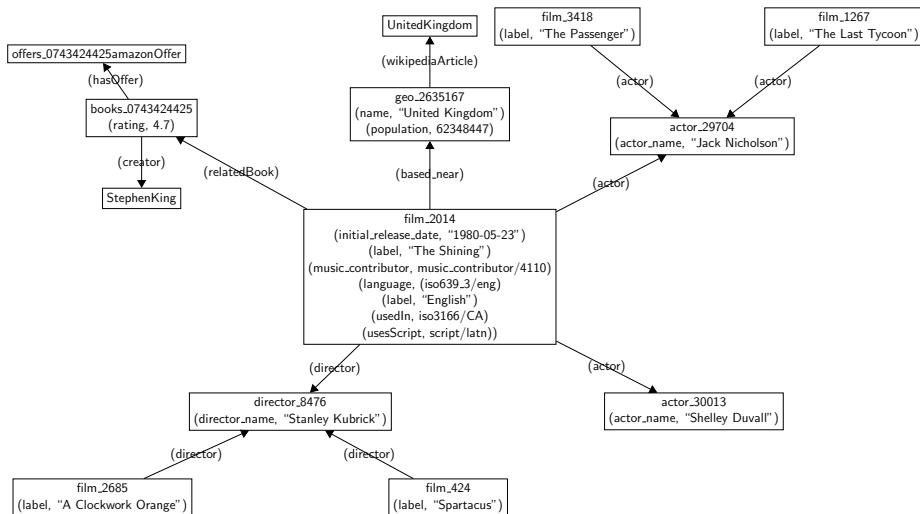
Offline graph analytics

- PageRank
- Clustering
- Connected components
- Diameter finding
- Graph colouring
- All pairs shortest path
- Graph pattern mining
- Machine learning algorithms
(Belief propagation, Gaussian non-negative matrix factorization)

Reachability Queries

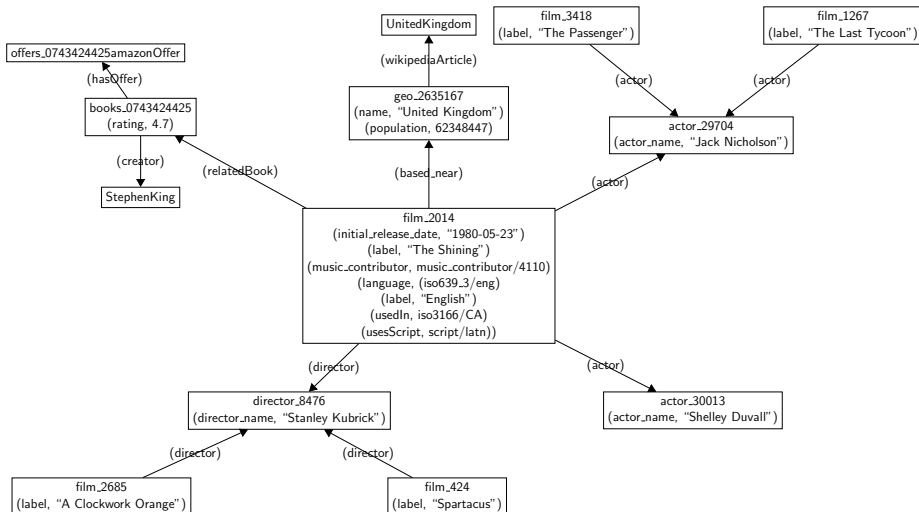


Reachability Queries



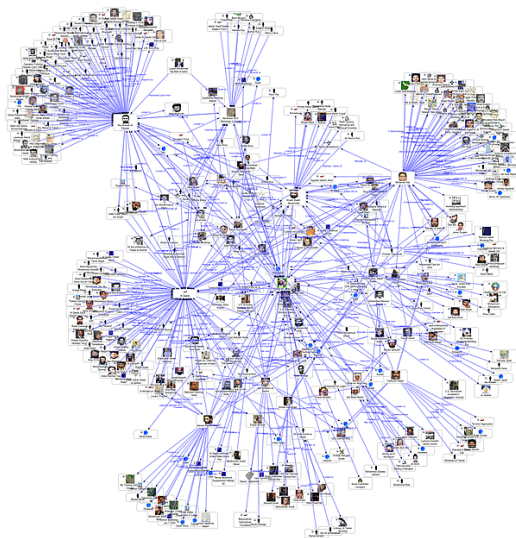
Can you reach film_1267 from film_2014?

Reachability Queries



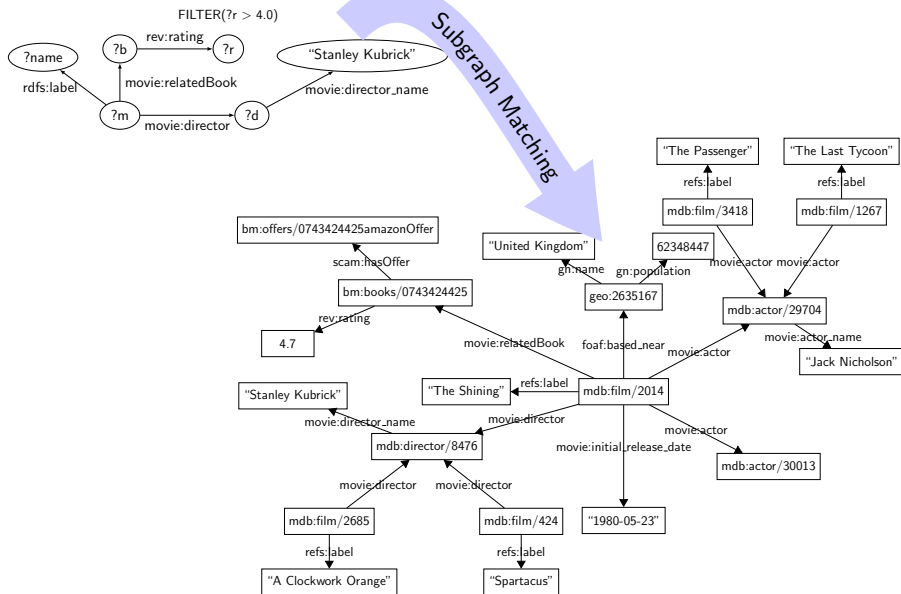
Is there a book whose rating is > 4.0 associated with a film that was directed by Stanley Kubrick?

Reachability Queries



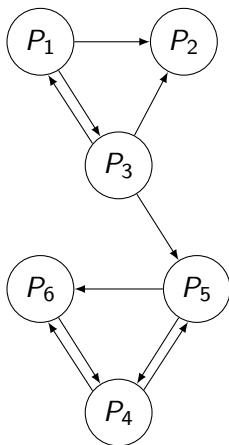
Think of Facebook graph and finding friends of friends.

Subgraph Matching



PageRank Computation

A web page is important if it is pointed to by other important pages.



$$r(P_i) = (1 - d) + d \sum_{P_j \in B_{P_i}} \frac{r(P_j)}{|F_{P_j}|}$$

(let $d = 1$)

$$r(P_2) = \frac{r(P_1)}{2} + \frac{r(P_3)}{3}$$

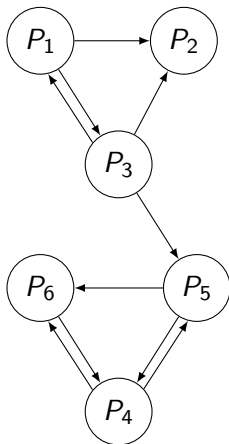
$$r_{k+1}(P_i) = \sum_{P_j \in B_{P_i}} \frac{r_k(P_j)}{|F_{P_j}|}$$

B_{P_i} : in-neighbours of P_i

F_{P_i} : out-neighbours of P_i

PageRank Computation

A web page is important if it is pointed to by other important pages.



$$r_{k+1}(P_i) = \sum_{P_j \in B_{P_i}} \frac{r_k(P_j)}{|F_{P_j}|}$$

Iteration 0	Iteration 1	Iteration 2	Rank at Iter. 2
$r_0(P_1) = 1/6$	$r_1(P_1) = 1/18$	$r_2(P_1) = 1/36$	5
$r_0(P_2) = 1/6$	$r_1(P_2) = 5/36$	$r_2(P_2) = 1/18$	4
$r_0(P_3) = 1/6$	$r_1(P_3) = 1/12$	$r_2(P_3) = 1/36$	5
$r_0(P_4) = 1/6$	$r_1(P_4) = 1/4$	$r_2(P_4) = 17/72$	1
$r_0(P_5) = 1/6$	$r_1(P_5) = 5/36$	$r_2(P_5) = 11/72$	3
$r_0(P_6) = 1/6$	$r_1(P_6) = 1/6$	$r_2(P_6) = 14/72$	2

Iterative processing
Touch each vertex

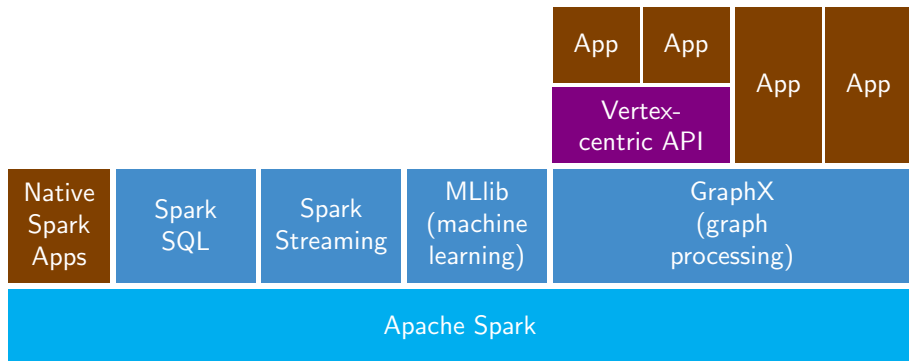
Can MapReduce be Used for Graph Analytics?

- Yes; map and reduce functions can be written for graph analytics workloads
 - Scalable Graph processing Class *SGC* [Qin et al., 2014]
 - Connected component computation [Kiveris et al., 2014; Rastogi et al., 2013]
- Not suitable for iterative processing due to data movement at each stage
 - No guarantee that computation will be assigned to the same worker nodes in the next round
- High I/O cost
 - Need to save in storage system (HDFS) intermediate results of each iteration

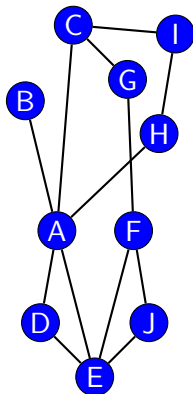
Can MapReduce be Used for Graph Analytics?

- Yes; map and reduce functions can be written for graph analytics workloads
 - Scalable Graph processing Class *SGC* [Qin et al., 2014]
 - Connected component computation [Kiveris et al., 2014; Rastogi et al., 2013]
- Not suitable for iterative processing due to data movement at each stage
 - No guarantee that computation will be assigned to the same worker nodes in the next round
- High I/O cost
 - Need to save in storage system (HDFS) intermediate results of each iteration
- There are systems that address these concerns
 - HaLoop [Bu et al., 2010, 2012]
 - GraphX over Spark [Gonzalez et al., 2014]

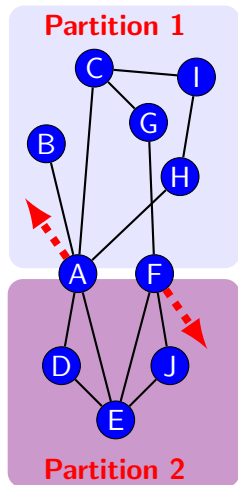
- Built on top of Spark
- Objective is to combine data analytics with graph processing
 - Unify computation on tables and graphs
- Carefully convert graph to tabular representation
- Native GraphX API or can accommodate vertex-centric computation



GraphX: Representation of Graphs as Tables

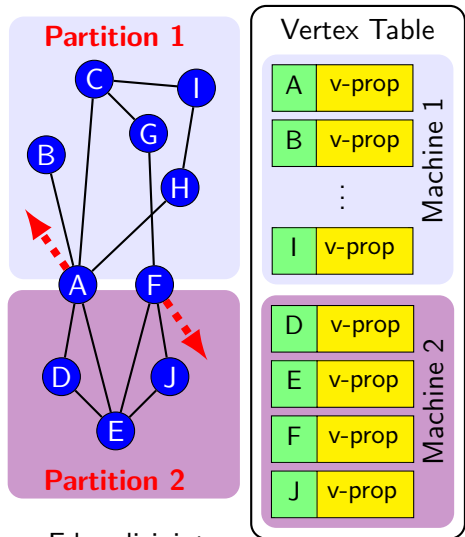


GraphX: Representation of Graphs as Tables



Edge-disjoint
partitioning

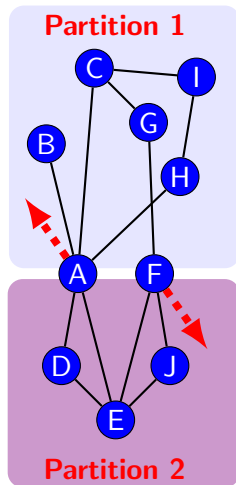
GraphX: Representation of Graphs as Tables



Edge-disjoint
partitioning

(RDD)
v-prop: vertex prop.

GraphX: Representation of Graphs as Tables



Edge-disjoint
partitioning

Vertex Table	
A	v-prop
B	v-prop
⋮	
I	v-prop

Machine 1

D	v-prop
E	v-prop
F	v-prop
J	v-prop

Machine 2

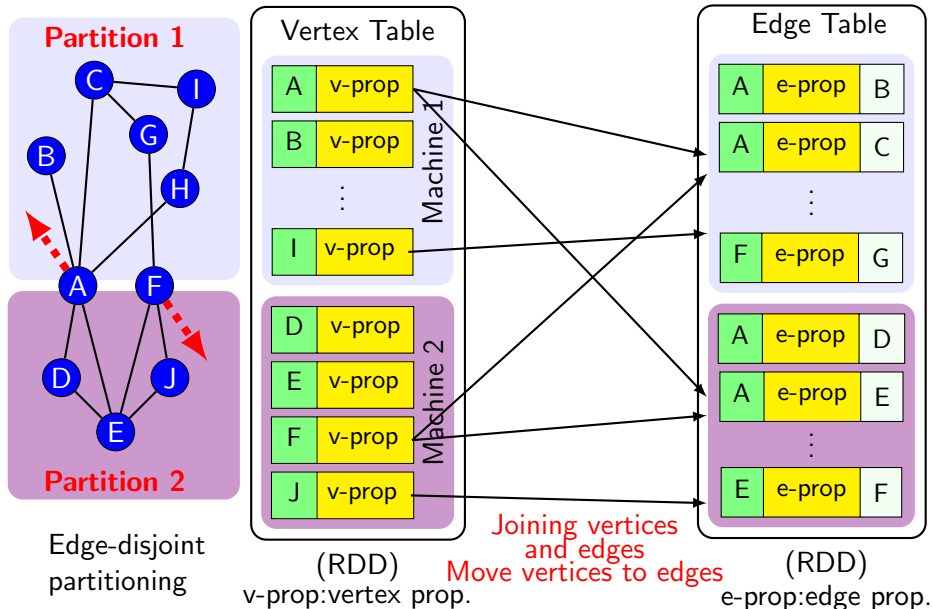
(RDD)
v-prop:vertex prop.

Edge Table		
A	e-prop	B
A	e-prop	C
⋮		
F	e-prop	G

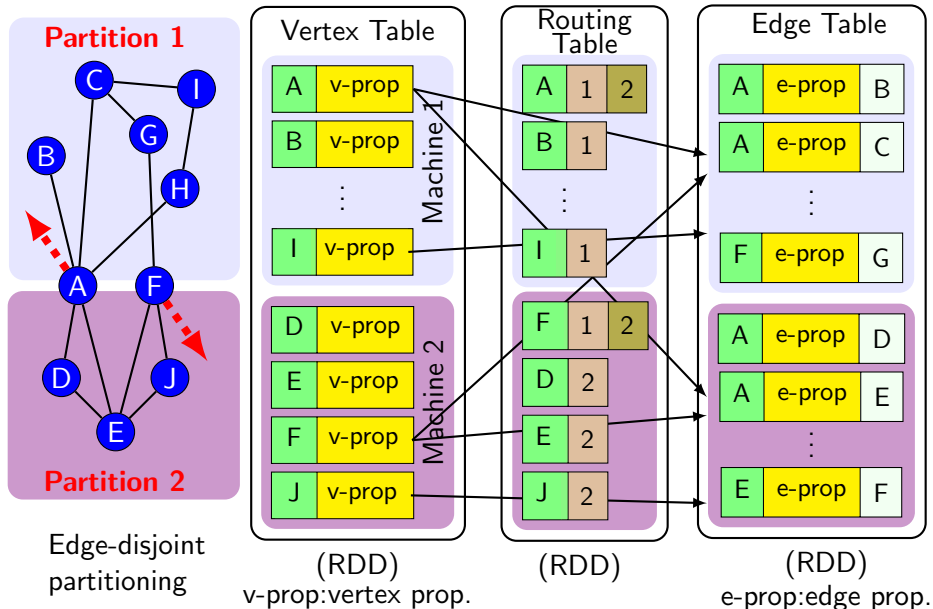
A	e-prop	D
A	e-prop	E
⋮		
E	e-prop	F

(RDD)
e-prop:edge prop.

GraphX: Representation of Graphs as Tables

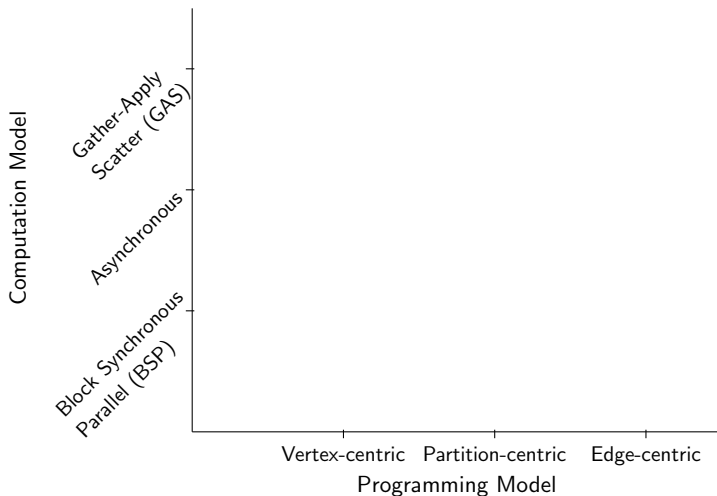


GraphX: Representation of Graphs as Tables

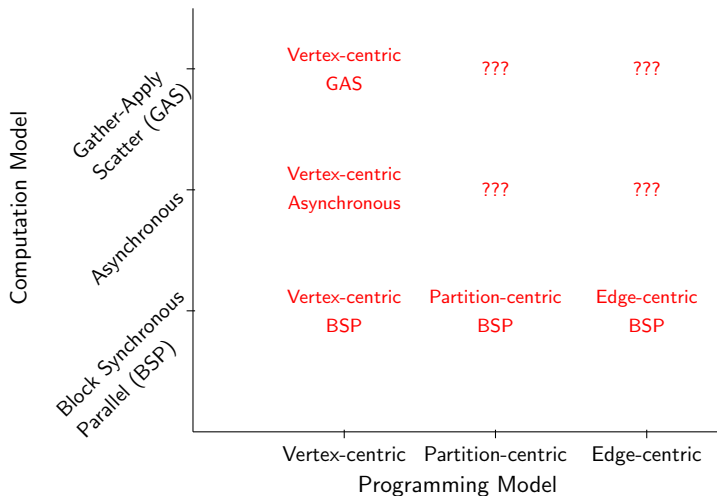


- Programming model
- Computation model

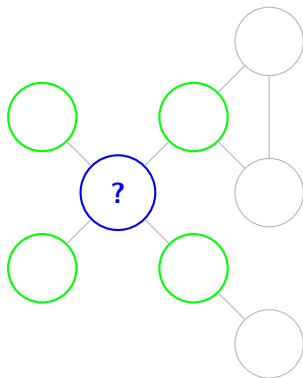
- Programming model
- Computation model



- Programming model
- Computation model

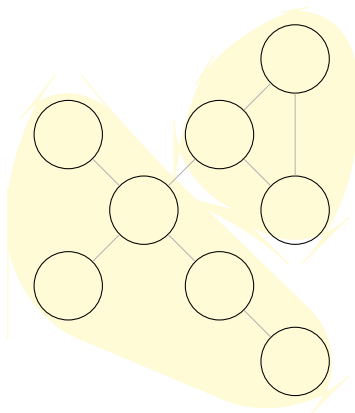


- Vertex-centric
 - Computation on a vertex is the focus
 - “Think like a vertex”
 - Vertex computation depends on its own state + states of its neighbors
 - `Compute(vertex v)`
 - `GetValue()`, `WriteValue()`

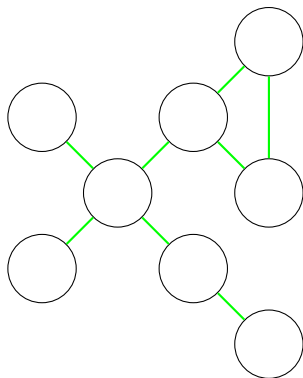


Programming Models

- Vertex-centric
- Partition-centric (Block-centric)
 - Computation on an entire partition is specified
 - “Think like a block” or “Think like a graph”
 - Aim is to reduce the communication cost among vertices



- Vertex-centric
- Partition-centric (Block-centric)
- Edge-centric
 - Computation is specified on each edge rather than on each vertex or block
 - `Compute(edge e)`



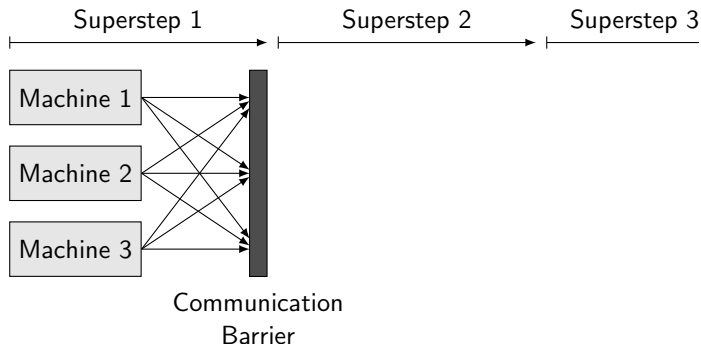
- Block Synchronous Parallel (BSP) [Valiant, 1990]

- Block Synchronous Parallel (BSP) [Valiant, 1990]

Computation

Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]

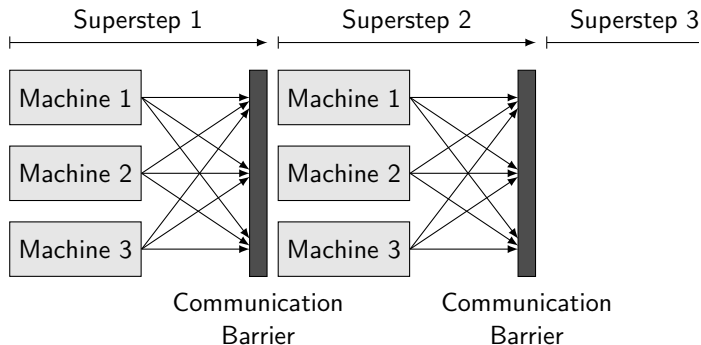


Each machine performs computation on its graph partition

At the end of each superstep results are **pushed** to other workers

Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]

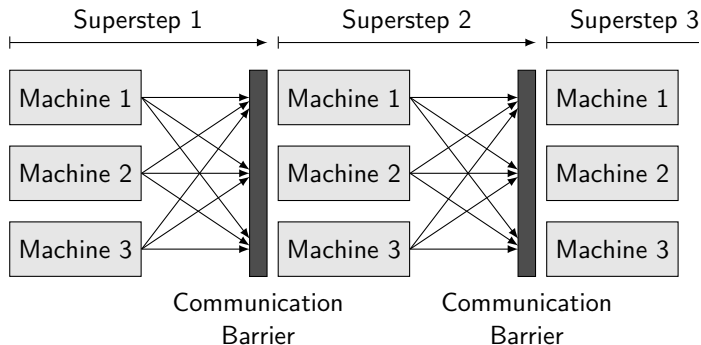


Each machine performs computation on its graph partition

At the end of each superstep results are **pushed** to other workers

Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]



Each machine performs computation on its graph partition

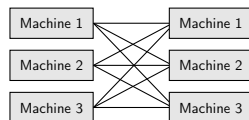
At the end of each superstep results are **pushed** to other workers

Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel

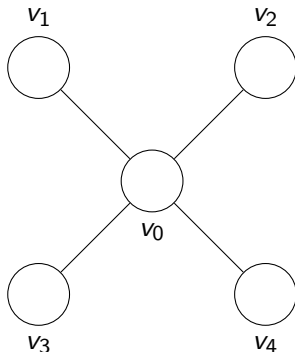
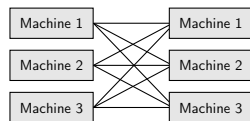
Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel
 - No communication barriers. ✓
 - Uses the *most recent* values. ✓
 - Implemented via distributed locking



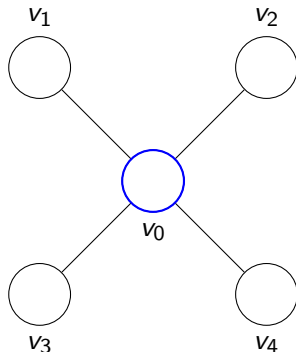
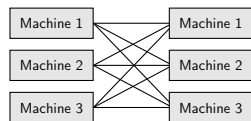
Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel
 - No communication barriers. ✓
 - Uses the *most recent* values. ✓
 - Implemented via distributed locking
 - Consider vertex-centric program



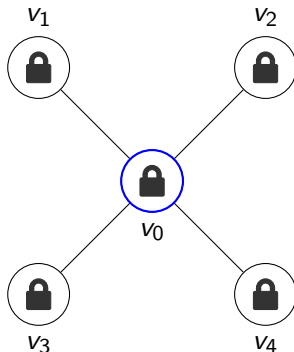
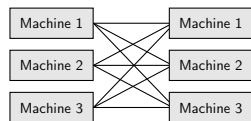
Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel
 - No communication barriers. ✓
 - Uses the *most recent* values. ✓
 - Implemented via distributed locking
 - Consider vertex-centric program



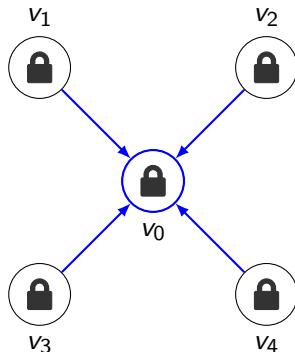
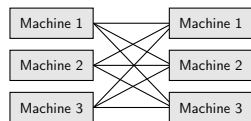
Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel
 - No communication barriers. ✓
 - Uses the *most recent* values. ✓
 - Implemented via distributed locking
 - Consider vertex-centric program



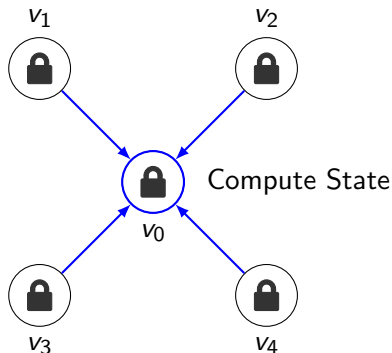
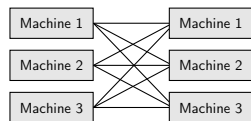
Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel
 - No communication barriers. ✓
 - Uses the *most recent* values. ✓
 - Implemented via distributed locking
 - Consider vertex-centric program



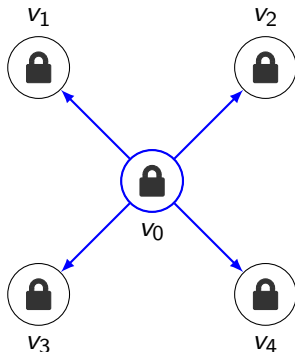
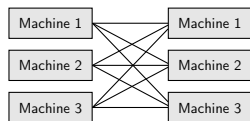
Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel
 - No communication barriers. ✓
 - Uses the *most recent* values. ✓
 - Implemented via distributed locking
 - Consider vertex-centric program



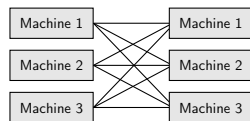
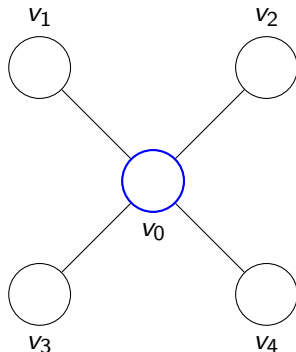
Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel
 - No communication barriers. ✓
 - Uses the *most recent* values. ✓
 - Implemented via distributed locking
 - Consider vertex-centric program



Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel
 - No communication barriers. ✓
 - Uses the *most recent* values. ✓
 - Implemented via distributed locking
 - Consider vertex-centric program

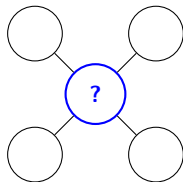


Computational Models

- Block Synchronous Parallel (BSP) [Valiant, 1990]
- Asynchronous Parallel
- Gather-Apply-Scatter (GAS)
 - Similar to BSP, but pull-based
 - Gather: pull state
 - Apply: Compute function
 - Scatter: Update state
 - Updates of states separated from scheduling

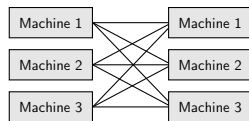
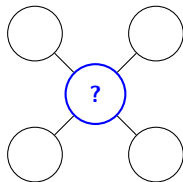
Vertex-Centric Asynchronous Systems

- “Think like a vertex”
- `Compute(vertex v)`



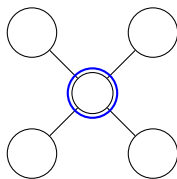
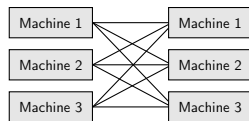
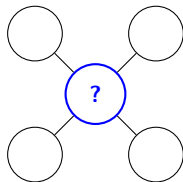
Vertex-Centric Asynchronous Systems

- “Think like a vertex”
- `Compute(vertex v)`
- Supersteps exist along with synchronization barriers, but ...
- `Compute(vertex v)` function can see messages it was sent in the same superstep as well as those that come at the end of the previous superstep



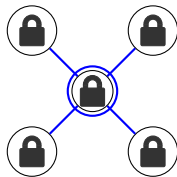
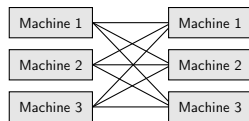
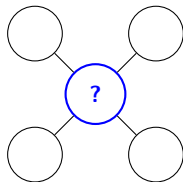
Vertex-Centric Asynchronous Systems

- “Think like a vertex”
- `Compute(vertex v)`
- Supersteps exist along with synchronization barriers, but ...
- `Compute(vertex v)` function can see messages it was sent in the same superstep as well as those that come at the end of the previous superstep
- Consistency of vertex states: distributed locking



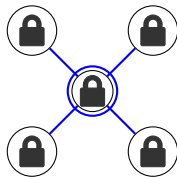
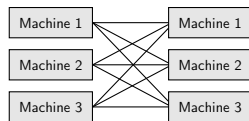
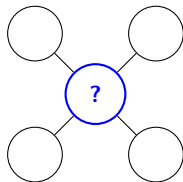
Vertex-Centric Asynchronous Systems

- “Think like a vertex”
- `Compute(vertex v)`
- Supersteps exist along with synchronization barriers, but ...
- `Compute(vertex v)` function can see messages it was sent in the same superstep as well as those that come at the end of the previous superstep
- Consistency of vertex states: distributed locking



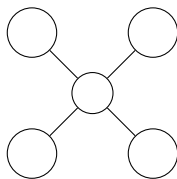
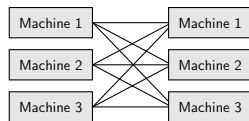
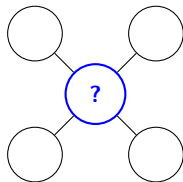
Vertex-Centric Asynchronous Systems

- “Think like a vertex”
- `Compute(vertex v)`
- Supersteps exist along with synchronization barriers, but ...
- `Compute(vertex v)` function can see messages it was sent in the same superstep as well as those that come at the end of the previous superstep
- Consistency of vertex states: distributed locking



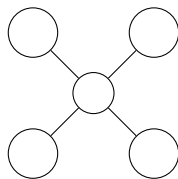
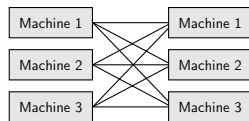
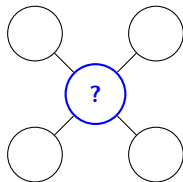
Vertex-Centric Asynchronous Systems

- “Think like a vertex”
- `Compute(vertex v)`
- Supersteps exist along with synchronization barriers, but ...
- `Compute(vertex v)` function can see messages it was sent in the same superstep as well as those that come at the end of the previous superstep
- Consistency of vertex states: distributed locking



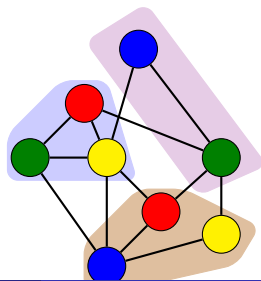
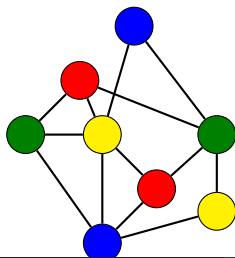
Vertex-Centric Asynchronous Systems

- “Think like a vertex”
- `Compute(vertex v)`
- Supersteps exist along with synchronization barriers, but ...
- `Compute(vertex v)` function can see messages it was sent in the same superstep as well as those that come at the end of the previous superstep
- Consistency of vertex states: distributed locking
- Consistency issues: no guarantee about input to `Compute()`
- Example systems: GRACE [Wang et al., 2013], GiraphCU [Han and Daudjee, 2015]



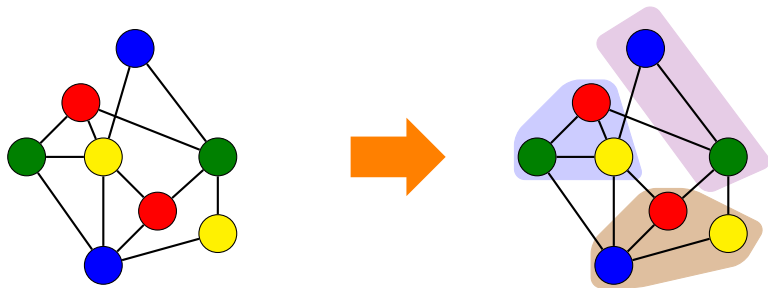
Partition- (Block-)Centric BSP Systems

- Blogel [Yan et al., 2014]: “Think like a block”; also “think like a graph” [Tian et al., 2013]
- Better handles the characteristics of real-world graphs by reducing communication
- Exploit the partitioning of the graph
- Message exchanges only among **blocks**
- Within a block, run a serial in-memory algorithm; BSP between partitions



Blogel Principles

- Exploit the partitioning of the graph
- Message exchanges only among **blocks**
- Block: a **connected** subgraph of the graph
- Within a block, run a serial in-memory algorithm; no need to follow a BSP model



Benefits of Partition- (Block-)Centric BSP

- High-degree vertices inside a block send no messages
- Fewer number of supersteps
- Fewer number of blocks than vertices



- “Think like an edge”
- $\text{Compute}(\text{edge } e)$
- Number of edges $\gg$ number of vertices
 - More computation but perhaps fewer messages
 - Operate on unsorted sequence of edges $\Rightarrow$ no random access
- X-Stream [Roy et al., 2013]

- OLAP in RDBMS
 - Usage: Data Warehousing + Business Intelligence
 - Model: Multidimensional cube
 - Operations: Roll-up, drill-down, and slice and dice
- Analytics that we discussed over graphs is much different
- Can we do OLAP-style analytics over graphs?
 - There is some work
 - Graph summarization [Tian et al., 2008]
 - Snapshot-based Aggregation [Chen et al., 2008]
 - Graph Cube [Zhao et al., 2011]
 - Pagrol [Wang et al., 2014]
 - Gagg Model [Maali et al., 2015]

Outline

- 1 Distributed Storage Systems
- 2 Main Memory Systems
- 3 Big Data Processing
 - MapReduce [Li et al., 2014]
 - Spark [Zaharia, 2016]
- 4 Stream Data Processing
- 5 Graph Processing
- 6 RDF Data Management

- Everything is an **uniquely** named **resource**



RDF Introduction

`xmlns:y=http://data.linkedmdb.org/resource/actor/
y:JN29704`

- Everything is an **uniquely** named **resource**
- Prefixes can be used to shorten the names



RDF Introduction

`xmlns:y=http://data.linkedmdb.org/resource/actor/
y:JN29704`

- Everything is an **uniquely** named **resource**
- Prefixes can be used to shorten the names
- Properties of resources can be defined



`y:JN29704 hasName "Jack Nicholson"
y:JN29704 BornOnDate "1937-04-22"`

RDF Introduction

- Everything is an **uniquely** named **resource**
- Prefixes can be used to shorten the names
- Properties of resources can be defined
- Relationships with other resources can be defined

xmlns:y=http://data.linkedmdb.org/resource/actor/
y:JN29704



y:JN29704 hasName "Jack Nicholson"
y:JN29704 BornOnDate "1937-04-22"

movieActor

y:TS2014



y:TS2014 title "The Shining"
y:TS2014 releaseDate "1980-05-23"

RDF Introduction

- Everything is an **uniquely** named **resource**
- Prefixes can be used to shorten the names
- Properties of resources can be defined
- Relationships with other resources can be defined
- Resource descriptions can be contributed by different people/groups and can be located anywhere in the web
- Integrated web “database”

xmlns:y=http://data.linkedmdb.org/resource/actor/
y:JN29704



y:JN29704 hasName “Jack Nicholson”
y:JN29704 BornOnDate “1937-04-22”

movieActor

y:TS2014



y:TS2014 title “The Shining”
y:TS2014 releaseDate “1980-05-23”

RDF Data Model

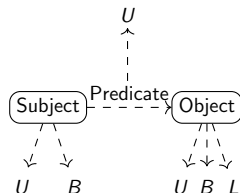
- Triple: Subject, Predicate (Property), Object
(s, p, o)

Subject: the entity that is described (URI or blank node)

Predicate: a feature of the entity (URI)

Object: value of the feature (URI, blank node or literal)

- $(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$
- Set of RDF triples is called an **RDF graph**



U : set of URIs

B : set of blank nodes

L : set of literals

Subject	Predicate	Object
<code>http://...imdb.../film/2014</code>	<code>rdfs:label</code>	"The Shining"
<code>http://...imdb.../film/2014</code>	<code>movie:releaseDate</code>	"1980-05-23"
<code>http://...imdb.../29704</code>	<code>movie:actor_name</code>	"Jack Nicholson"
...	...	...

RDF Example Instance

Prefixes: mdb=http://data.linkedmdb.org/resource/; geo=http://sws.geonames.org/
bm=http://wifo5-03.informatik.uni-mannheim.de/bookmashup/
lexvo=http://lexvo.org/id/; wp=http://en.wikipedia.org/wiki/

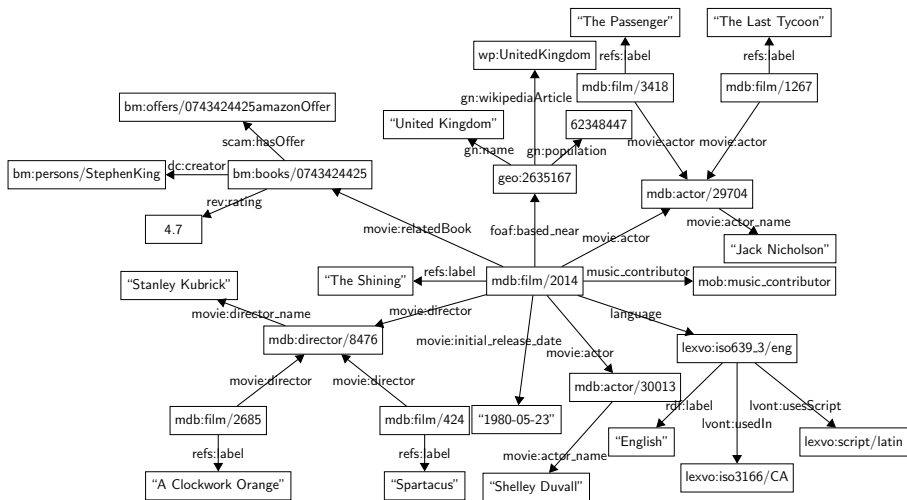
Subject	Predicate	Object
<u>mdb: film/2014</u>	rdfs:label	<u>"The Shining"</u>
mdb:film/2014	movie:initial_release_date	"1980-05-23"
mdb:film/2014	movie:director	mdb:director/8476
mdb:film/2014	movie:actor	mdb:actor/29704
mdb:film/2014	movie:actor	<u>mdb: actor/30013</u>
mdb:film/2014	movie:music_contributor	mdb: music_contributor/4110
mdb:film/2014	foaf:based_near	geo:2635167
mdb:film/2014	movie:relatedBook	bm:0743424425
mdb:film/2014	movie:language	lexvo:iso639-3/eng
mdb:director/8476	movie:director_name	"Stanley Kubrick"
mdb:film/2685	movie:director	mdb:director/8476
mdb:film/2685	rdfs:label	"A Clockwork Orange"
mdb:film/424	movie:director	<u>mdb:director/8476</u>
mdb:film/424	rdfs:label	"Spartacus"
mdb:actor/29704	movie:actor_name	"Jack Nicholson"
mdb:film/1267	movie:actor	mdb:actor/29704
mdb:film/1267	rdfs:label	"The Last Tycoon"
mdb:film/3418	movie:actor	mdb:actor/29704
mdb:film/3418	rdfs:label	"The Passenger"
geo:2635167	gn:name	"United Kingdom"
geo:2635167	gn:population	62348447
geo:2635167	gn:wikipediaArticle	wp:United_Kingdom
bm:books/0743424425	dc:creator	bm:persons/Stephen+King
bm:books/0743424425	rev:rating	4.7
bm:books/0743424425	scom:hasOffer	bm:offers/0743424425amazonOffer
lexvo:iso639-3/eng	rdfs:label	"English"
lexvo:iso639-3/eng	lvont:usedIn	lexvo:iso3166/CA
lexvo:iso639-3/eng	lvont:usesScript	lexvo:script/Latn

URI

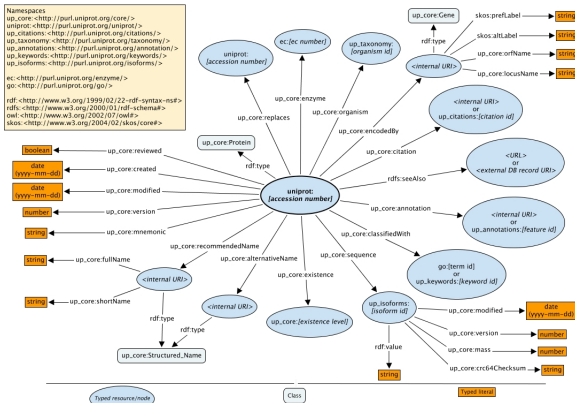
Literal

URI

RDF Graph



- UniProt collects data from >150 biological resources
- Claim: “lack of a common standard to represent and link information makes data integration an expensive business” $\Rightarrow$ RDF can help



UniProt in RDF – What does the data look like?

- UniProt accession for the human CYP51 protein – Q16850
- Encode it as RDF: <http://purl.uniprot.org/uniprot/Q16850.rdf>

UniProt in RDF – What does the data look like?

- UniProt accession for the human CYP51 protein – Q16850
- Encode it as RDF: <http://purl.uniprot.org/uniprot/Q16850.rdf>
- XML/RDF format

```
<rdf:Description rdf:about="http://purl.uniprot.org/citations/8619637">  
<rdf:type rdf:resource="http://purl.uniprot.org/core/Journal_Citation"/>  
<title>The ubiquitously expressed human CYP51 encodes lanosterol 14 alpha-demethylase, a cytochrome  
P450 whose expression is regulated by oxysterols.</title>  
<author>Stroemstedt M.</author>  
<author>Rozman D.</author>  
<author>Waterman M.R.</author>  
<skos:exactMatch rdf:resource="http://purl.uniprot.org/pubmed/8619637"/>  
<foaf:primaryTopicOf rdf:resource="https://www.ncbi.nlm.nih.gov/pubmed/8619637"/>  
<dcterms:identifier>doi:10.1006/abbi.1996.0193</dcterms:identifier>  
<date rdf:datatype="http://www.w3.org/2001/XMLSchema#gYear">1996</date>  
<name>Arch. Biochem. Biophys.</name>  
<volume>329</volume>  
<pages>73-81</pages>  
</rdf:Description>
```

Predicate

Subject

Object

UniProt in RDF – What does the data look like?

- UniProt accession for the human CYP51 protein – Q16850
- Encode it as RDF: <http://purl.uniprot.org/uniprot/Q16850.rdf>
- XML/RDF format

```
<rdf:Description rdf:about="http://purl.uniprot.org/citations/8619637" >  
<rdf:type rdf:resource="http://purl.uniprot.org/core/Journal_Citation" />  
<title>The ubiquitously expressed human CYP51 encodes lanosterol 14 alpha-demethylase, a cytochrome  
P450 whose expression is regulated by oxysterols.</title>  
<author>Stroemstedt M.</author>  
<author>Rozman D.</author>  
<author>Waterman M.R.</author>  
<skos:exactMatch rdf:resource="http://purl.uniprot.org/pubmed/8619637" />  
<foaf:primaryTopicOf rdf:resource="https://www.ncbi.nlm.nih.gov/pubmed/8619637" />  
<dcterms:identifier>doi:10.1006/abbi.1996.0193</dcterms:identifier>  
<date rdf:datatype="http://www.w3.org/2001/XMLSchema#gYear">1996</date>  
<name>Arch. Biochem. Biophys.</name>  
<volume>329</volume>  
<pages>73-81</pages>  
</rdf:Description>
```

- This can be shown as a table <Subject, Predicate, Object >

RDF Query Model – SPARQL

- Query Model - **SPARQL Protocol and RDF Query Language**
- Given U (set of URIs), L (set of literals), and V (set of variables), a SPARQL expression is defined recursively:

- an atomic triple pattern, which is an element of

$$(U \cup V) \times (U \cup V) \times (U \cup V \cup L)$$

- $?x$ rdfs:label "The Shining"
 - P FILTER R , where P is a graph pattern expression and R is a built-in SPARQL condition (i.e., analogous to a SQL predicate)
 - $?x$ rev:rating ?p FILTER(?p > 3.0)
 - $P1$ AND/OPT/UNION $P2$, where $P1$ and $P2$ are graph pattern expressions

RDF Query Model – SPARQL

- Query Model - **SPARQL Protocol and RDF Query Language**
- Given U (set of URIs), L (set of literals), and V (set of variables), a SPARQL expression is defined recursively:
 - an atomic triple pattern, which is an element of

$$(U \cup V) \times (U \cup V) \times (U \cup V \cup L)$$

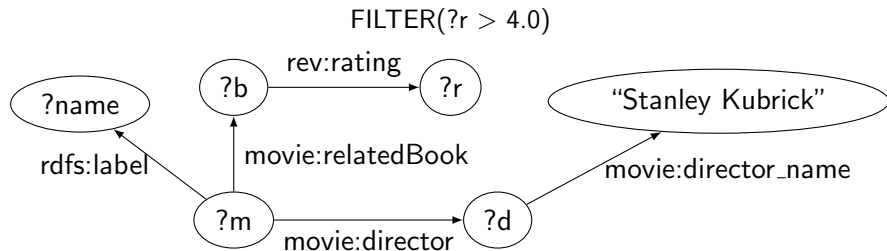
- $?x$ `rdfs:label` "The Shining"
- P `FILTER` R , where P is a graph pattern expression and R is a built-in SPARQL condition (i.e., analogous to a SQL predicate)
 - $?x$ `rev:rating` $?p$ `FILTER`($?p > 3.0$)
- $P1$ `AND/OPT/UNION` $P2$, where $P1$ and $P2$ are graph pattern expressions

- Example:

```
SELECT ?name
WHERE {
  ?m rdfs:label ?name. ?m movie:director ?d.
  ?d movie:director_name "Stanley Kubrick".
  ?m movie:relatedBook ?b. ?b rev:rating ?r.
  FILTER(?r > 4.0)
}
```

SPARQL Queries

```
SELECT ?name
WHERE {
  ?m rdfs:label ?name. ?m movie:director ?d.
  ?d movie:director_name "Stanley Kubrick".
  ?m movie:relatedBook ?b. ?b rev:rating ?r.
  FILTER(?r > 4.0)
}
```



UniProt in RDF – The Data Can Be Queried

- RDF encoded UniProt data can be queried using SPARQL:
<http://sparql.uniprot.org/sparql>

UniProt in RDF – The Data Can Be Queried

- RDF encoded UniProt data can be queried using SPARQL:

<http://sparql.uniprot.org/sparql>

- Get the GO function for Q16850 (from UniProt SPARQL endpoint)

```
PREFIX upc:<http://purl.uniprot.org/core/>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?goid ?golabel
WHERE {
  <http://purl.uniprot.org/uniprot/Q16850> a upc:Protein ;
  upc:classifiedWith ?keyword .
  ?keyword rdfs:seeAlso ?goid .
  ?goid rdfs:label ?golabel .
}
```

- Find the differential expression of probes and the p value that map to Q16850 (from Expression Atlas SPARQL endpoint)

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX atlasterms:<http://rdf.ebi.ac.uk/terms/atlas/>
SELECT distinct ?valueLabel ?pvalue
WHERE {
  ?value rdfs:label ?valueLabel .
  ?value atlasterms:pValue ?pvalue .
  ?value atlasterms:isMeasurementOf ?probe .
  ?probe atlasterms:dbXref <http://purl.uniprot.org/uniprot/Q16850> .
}
ORDER BY ASC(?pvalue)
```

Naïve Triple Store Design

```
SELECT ?name
WHERE {
  ?m rdfs:label ?name. ?m movie:director ?d.
  ?d movie:director_name "Stanley Kubrick".
  ?m movie:relatedBook ?b. ?b rev:rating ?r.
  FILTER(?r > 4.0)
}
```

Subject	Property	Object
mdb:film/2014	rdfs:label	"The Shining"
mdb:film/2014	movie:initial_release_date	"1980-05-23"
mdb:film/2014	movie:director	mdb:director/8476
mdb:film/2014	movie:actor	mdb:actor/29704
mdb:film/2014	movie:actor	mdb:actor/30013
mdb:film/2014	movie:music_contributor	mdb:music_contributor/4110
mdb:film/2014	foaf:based_near	geo:2635167
mdb:film/2014	movie:relatedBook	bm:0743424425
mdb:film/2014	movie:language	lexvo:iso639-3/eng
mdb:director/8476	movie:director_name	"Stanley Kubrick"
mdb:film/2685	movie:director	mdb:director/8476
mdb:film/2685	rdfs:label	"A Clockwork Orange"
mdb:film/424	movie:director	mdb:director/8476
mdb:film/424	rdfs:label	"Spartacus"
mdb:actor/29704	movie:actor_name	"Jack Nicholson"
mdb:film/1267	movie:actor	mdb:actor/29704
mdb:film/1267	rdfs:label	"The Last Tycoon"
mdb:film/3418	movie:actor	mdb:actor/29704
mdb:film/3418	rdfs:label	"The Passenger"
geo:2635167	gn:name	"United Kingdom"
geo:2635167	gn:population	62348447
geo:2635167	gn:wikipediaArticle	wp:United_Kingdom
bm:books/0743424425	dc:creator	bm:persons/Stephen+King
bm:books/0743424425	rev:rating	4.7
bm:books/0743424425	scom:hasOffer	bm:offers/0743424425amazonOffer
lexvo:iso639-3/eng	rdfs:label	"English"
lexvo:iso639-3/eng	lvont:usedIn	lexvo:iso3166/CA
lexvo:iso639-3/eng	lvont:usesScript	lexvo:script/Latn

Naïve Triple Store Design

```
SELECT ?name
WHERE {
  ?m rdfs:label ?name. ?m movie:director ?d.
  ?d movie:director_name "Stanley Kubrick".
  ?m movie:relatedBook ?b. ?b rev:rating ?r.
  FILTER(?r > 4.0)
}
```

Subject	Property	Object
mdb:film/2014	rdfs:label	"The Shining"
mdb:film/2014	movie:initial_release_date	"1980-05-23"
mdb:film/2014	movie:director	mdb:director/8476
mdb:film/2014	movie:actor	mdb:actor/29704
mdb:film/2014	movie:actor	mdb:actor/30013
mdb:film/2014	movie:music_contributor	mdb:music_contributor/4110
mdb:film/2014	foaf:based_near	geo:2635167
mdb:film/2014	movie:relatedBook	bm:0743424425
mdb:film/2014	movie:language	lexvo:iso639-3/eng
mdb:director/8476	movie:director_name	"Stanley Kubrick"
mdb:film/2685	movie:director	mdb:director/8476
mdb:film/2685	rdfs:label	"A Clockwork Orange"
mdb:film/424	movie:director	mdb:director/8476
mdb:film/424	rdfs:label	"Spartacus"
mdb:actor/29704	movie:actor_name	"Jack Nicholson"
mdb:film/1267	movie:actor	mdb:actor/29704
mdb:film/1267	rdfs:label	"The Last Tycoon"
mdb:film/3418	movie:actor	mdb:actor/29704
mdb:film/3418	rdfs:label	"The Passenger"
geo:2635167	gn:name	"United Kingdom"
geo:2635167	gn:population	62348447
geo:2635167	gn:wikipediaArticle	wp:United.Kingdom
bm:books/0743424425	dc:creator	bm:persons/Stephen+King
bm:books/0743424425	rev:rating	4.7
bm:books/0743424425	scom:hasOffer	bm:offers/0743424425amazonOffer
lexvo:iso639-3/eng	rdfs:label	"English"
lexvo:iso639-3/eng	lvont:usedIn	lexvo:iso3166/CA
lexvo:iso639-3/eng	lvont:usesScript	lexvo:script/Latn



```
SELECT T1.object
FROM T as T1, T as T2, T as T3,
      T as T4, T as T5
WHERE T1.p="rdfs:label"
AND T2.p="movie:relatedBook"
AND T3.p="movie:director"
AND T4.p="rev:rating"
AND T5.p="movie:director_name"
AND T1.s=T2.s
AND T1.s=T3.s
AND T2.o=T4.s
AND T3.o=T5.s
AND T4.o > 4.0
AND T5.o="Stanley Kubrick"
```

Naïve Triple Store Design

```
SELECT ?name
WHERE {
    ?m rdfs:label ?name.
    ?m movie:director ?d.
    ?d movie:director_name "Stanley Kubrick".
    ?m movie:relatedBook ?b.
    ?b rev:rating ?r.
    FILTER(?r > 4.0)
}
```

Subject	Property	Object
mdb:film/2014	rdfs:label	"The Shining"
mdb:film/2014	movie:initial_release_date	"1980-05-23"
mdb:film/2014	movie:director	mdb:director/8476
mdb:film/2014	movie:actor	mdb:actor/29704
mdb:film/2014	movie:actor	mdb:actor/30013
mdb:film/2014	movie:music_contributor	mdb:music_contributor/4110
mdb:film/2014	foaf:based_near	geo:2635167
mdb:film/2014	movie:relatedBook	bm:0743424425
mdb:film/2014	movie:language	lexvo:iso639-3/eng
mdb:director/8476	movie:director_name	"Stanley Kubrick"
mdb:film/2685	movie:director	mdb:director/8476
mdb:film/2685	rdfs:label	"A Clockwork Orange"
mdb:film/424	movie:director	mdb:director/8476
mdb:film/424	rdfs:label	"Spartacus"
mdb:actor/29704	movie:actor_name	"Jack Nicholson"
mdb:film/1267	movie:actor	mdb:actor/29704
mdb:film/1267	rdfs:label	"The Last Tycoon"
mdb:film/3418	movie:actor	mdb:actor/29704
mdb:film/3418	rdfs:label	"The Passenger"
geo:2635167	gn:name	"United Kingdom"
geo:2635167	gn:population	62348447
geo:2635167	gn:wikipediaArticle	wp:United.Kingdom
bm:books/0743424425	dc:creator	bm:persons/Stephen+King
bm:books/0743424425	rev:rating	4.7
bm:books/0743424425	scom:hasOffer	bm:offers/0743424425amazonOffer
lexvo:iso639-3/eng	rdfs:label	"English"
lexvo:iso639-3/eng	lvont:usedIn	lexvo:iso3166/CA
lexvo:iso639-3/eng	lvont:usesScript	lexvo:script/Latn

Easy to implement
but
too many self-joins!

```
FROM T1, T2, T3, T4, T5
WHERE T1.p="rdfs:label"
AND T2.p="movie:relatedBook"
AND T3.p="movie:director"
AND T4.p="rev:rating"
AND T5.p="movie:director_name"
AND T1.s=T2.s
AND T1.s=T3.s
AND T2.o=T4.s
AND T3.o=T5.s
AND T4.o > 4.0
AND T5.o="Stanley Kubrick"
```

Exhaustive Indexing

- RDF-3X [Neumann and Weikum, 2008, 2009], Hexastore [Weiss et al., 2008]
- Strings are mapped to ids using a mapping table

Original triple table

Subject	Property	Object
mdb: film/2014	rdfs:label	"The Shining"
mdb:film/2014	movie:initial_release_date	"1980-05-23"
mdb:director/8476	movie:director_name	"Stanley Kubrick"
mdb:film/2685	movie:director	mdb:director/8476

Encoded triple table

Subject	Property	Object
0	1	2
0	3	4
5	6	7
8	9	5

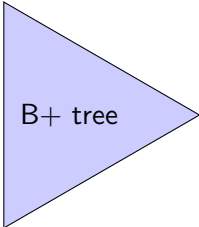
Mapping table

ID	Value
0	mdb: film/2014
1	rdfs:label
2	"The Shining"
3	movie:initial_release_date
4	"1980-05-23"
5	mdb:director/8476
6	movie:director_name
7	"Stanley Kubrick"
8	mdb:film/2685
9	movie:director

Exhaustive Indexing

- RDF-3X [Neumann and Weikum, 2008, 2009], Hexastore [Weiss et al., 2008]
- Strings are mapped to ids using a mapping table
- Triples are indexed in a clustered B+ tree in lexicographic order

Subject	Property	Object
0	1	2
0	3	4
5	6	7
8	9	5
⋮	⋮	⋮



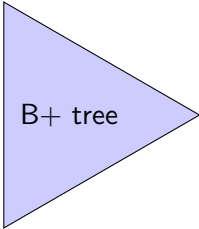
B+ tree

Easy querying
through mapping
table

Exhaustive Indexing

- RDF-3X [Neumann and Weikum, 2008, 2009], Hexastore [Weiss et al., 2008]
- Strings are mapped to ids using a mapping table
- Triples are indexed in a clustered B+ tree in lexicographic order
- Create indexes for permutations of the three columns: SPO, SOP, PSO, POS, OPS, OSP

Subject	Property	Object
0	1	2
0	3	4
5	6	7
8	9	5
⋮	⋮	⋮



B+ tree

Easy querying
through mapping
table

Exhaustive Indexing–Query Execution

- Each triple pattern can be answered by a range query
- Joins between triple patterns computed using merge join
- Join order is easy due to extensive indexing

Subject	Property	Object
0	1	2
0	3	4
5	6	7
8	9	5
⋮	⋮	⋮

ID	Value
0	mdb: film/2014
1	rdfs:label
2	"The Shining"
3	movie:initial_release_date
4	"1980-05-23"
5	mdb:director/8476
6	movie:director_name
7	"Stanley Kubrick"
8	mdb:film/2685
9	movie:director

Exhaustive Indexing–Query Execution

- Each triple pattern can be answered by a range query
- Joins between triple patterns computed using merge join
- Join order is easy due to extensive indexing

Advantages

- ▶ Eliminates some of the joins – they become range queries
- ▶ Merge join is easy and fast

8	9	5
⋮	⋮	⋮

3	movie:initial_release_date
4	"1980-05-23"
5	mdb:director/8476
6	movie:director_name
7	"Stanley Kubrick"
8	mdb:film/2685
9	movie:director

Exhaustive Indexing–Query Execution

- Each triple pattern can be answered by a range query
- Joins between triple patterns computed using merge join
- Join order is easy due to extensive indexing

Advantages

- ▶ Eliminates some of the joins – they become range queries
- ▶ Merge join is easy and fast

Disadvantages

- ▶ Space usage
- ▶ Expensive updates

8	mdb:film/2685
9	movie:director

Property Tables

- Grouping by entities; Jena [Wilkinson, 2006], DB2-RDF [Bornea et al., 2013]
- *Clustered property table*: group together the properties that tend to occur in the same (or similar) subjects
- *Property-class table*: cluster the subjects with the same type of property into one property table

Subject	Property	Object
mdb:film/2014	rdfs:label	"The Shining"
mdb:film/2014	movie:director	mdb:director/8476
mdb:film/2685	movie:director	mdb:director/8476
mdb:film/2685	rdfs:label	"A Clockwork Orange"
mdb:actor/29704	movie:actor_name	"Jack Nicholson"
...	...	...



Subject	refs:label	movie:director
mob:film/2014	"The Shining"	mob:director/8476
mob:film/2685	"The Clockwork Orange"	mob:director/8476



Subject	movie:actor_name
mdb:actor	"Jack Nicholson"

Property Tables

- Grouping by entities; Jena [Wilkinson, 2006], DB2-RDF [Bornea et al., 2013]
- *Clustered property table*: group together the properties that tend to

Advantages

- ▶ Fewer joins
- ▶ If the data is structured, we have a relational system – similar to normalized relations

mob:film/2014	movie:director	mob:director/8476
mdb:film/2685	movie:director	mdb:director/8476
mdb:film/2685	rdfs:label	"A Clockwork Orange"
mdb:actor/29704	movie:actor_name	"Jack Nicholson"
...	...	...



Subject	refs:label	movie:director
mob:film/2014	"The Shining"	mob:director/8476
mob:film/2685	"The Clockwork Orange"	mob:director/8476

Subject	movie:actor_name
mdb:actor	"Jack Nicholson"

Property Tables

- Grouping by entities; Jena [Wilkinson, 2006], DB2-RDF [Bornea et al., 2013]
- *Clustered property table*: group together the properties that tend to

Advantages

- ▶ Fewer joins
- ▶ If the data is structured, we have a relational system – similar to normalized relations

mob:film/2685	movie:director	mob:director/8476
mdb:film/2685	movie:director	mdb:director/8476

Disadvantages

- ▶ Potentially a lot of NULLs
- ▶ Clustering is not trivial
- ▶ Multi-valued properties are complicated

mob:film/2685	The Clockwork Orange	mob:director/8476
---------------	----------------------	-------------------

Vertical Partitioning

- Binary Tables [Abadi et al., 2007, 2009]:
 - Grouping by properties – for each property, build a two-column table, containing both subject and object, ordered by subjects
 - n two column tables (n is the number of unique properties in the data)

Subject	Property	Object
mdb:film/2014	rdfs:label	"The Shining"
mdb:film/2014	movie:director	mdb:director/8476
mdb:film/2685	movie:director	mdb:director/8476
mdb:film/2685	rdfs:label	"A Clockwork Orange"
mdb:actor/29704	movie:actor_name	"Jack Nicholson"
...	...	...

refs:label

Subject	Object
mdb:film/2014	"The Shining"
mdb:film/2685	"The Clockwork Orange"

movie:director

Subject	Object
mdb:film/2014	mdb:director/8476
mdb:film/2685	mdb:director/8476

movie:actor_name

Subject	Object
mdb:actor/29704	"Jack Nicholson"

Advantages

- ▶ Supports multi-valued properties
- ▶ No NULLs
- ▶ No clustering
- ▶ Read only needed attributes (i.e. less I/O)
- ▶ Good performance for subject-subject joins

Vertical Partitioning

Advantages

- ▶ Supports multi-valued properties
- ▶ No NULLs
- ▶ No clustering
- ▶ Read only needed attributes (i.e. less I/O)
- ▶ Good performance for subject-subject joins

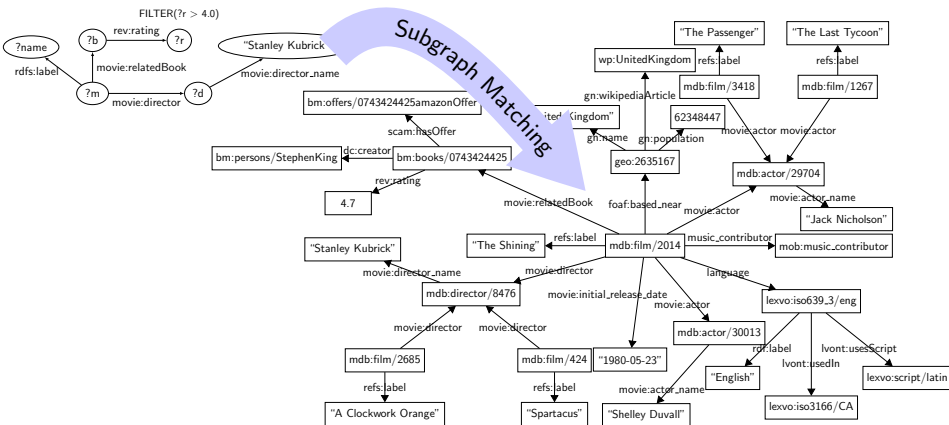
Disadvantages

- ▶ Not useful for subject-object joins
- ▶ Expensive inserts

Vertical Partitioning

- Binary Tables [Abadi et al., 2007, 2009]:
 - Grouping by properties – for each property, build a two-column table, containing both subject and object, ordered by subjects
 - n two column tables (n is the number of unique properties in the data)
- TripleBit [Yuan et al., 2013]:
 - Create a table with $|\text{triple}|$ columns, $|\text{objects}| + |\text{subjects}|$ rows with “1” if object/subject exists in triple; groups columns by predicate
 - Compress columns (since they are sparse); partition by predicate, then partition into chunks
 - (P,S,O) and (P,O,S) indexes on the chunks

- Answering SPARQL query $\equiv$ subgraph matching using homomorphism
- gStore [Zou et al., 2011, 2014], chameleon-db [Aluç et al., 2013]

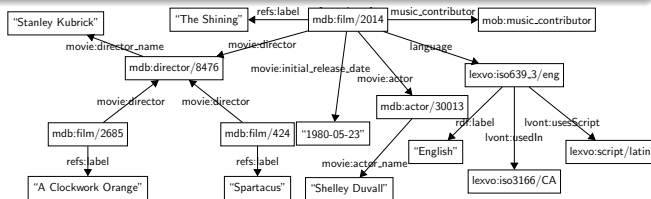


- Answering SPARQL query $\equiv$ subgraph matching using homomorphism
- gStore [Zou et al., 2011, 2014], chameleon-db [Aluç et al., 2013]



Advantages

- ▶ Maintains the graph structure
- ▶ Full set of queries can be handled



- Answering SPARQL query $\equiv$ subgraph matching using homomorphism
- gStore [Zou et al., 2011, 2014], chameleon-db [Aluç et al., 2013]

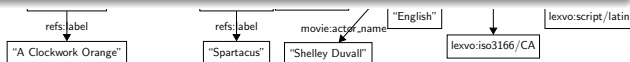
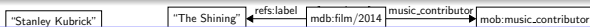


Advantages

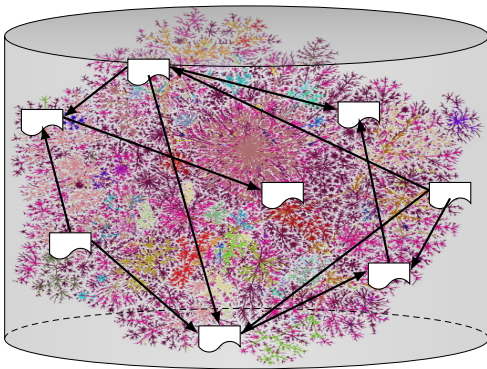
- ▶ Maintains the graph structure
- ▶ Full set of queries can be handled

Disadvantages

- ▶ Graph pattern matching is expensive

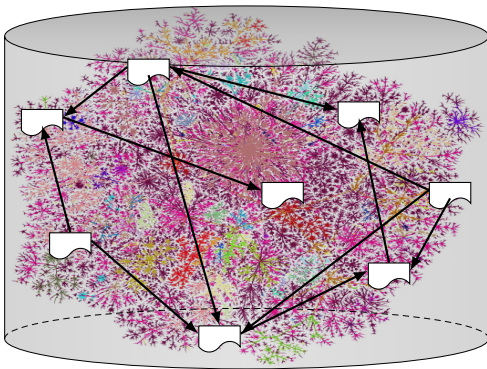


RDF Federation



- Distributed environment
- **SPARQL endpoints** can process SPARQL queries
- Non-SPARQL endpoints require additional components

RDF Federation



- Distributed environment
- **SPARQL endpoints** can process SPARQL queries
- Non-SPARQL endpoints require additional components
- Issues
 - Query decomposition
 - Localization (source selection)
 - Result composition

- RDF data warehouse D is partitioned ($\{D_1, \dots, D_n\}$) and placed on cloud platforms (such as HDFS, HBase)

- RDF data warehouse D is partitioned ($\{D_1, \dots, D_n\}$) and placed on cloud platforms (such as HDFS, HBase)
- SPARQL query is run through MapReduce jobs
- Data parallel execution

- RDF data warehouse D is partitioned ($\{D_1, \dots, D_n\}$) and placed on cloud platforms (such as HDFS, HBase)
- SPARQL query is run through MapReduce jobs
- Data parallel execution
- Examples: HARD [Rohloff and Schantz, 2010] , HadoopRDF [Husain et al., 2011] , EAGRE [Zhang et al., 2013] and JenaHBase [Khadilkar et al., 2012]

- RDF data warehouse D is partitioned ($\{D_1, \dots, D_n\}$) and placed on cloud platforms (such as HDFS, HBase)
- SPARQL query is run through MapReduce jobs
- Data parallel execution
- Examples: HARD [Rohloff and Schantz, 2010] , HadoopRDF [Husain et al., 2011] , EAGRE [Zhang et al., 2013] and JenaHBase [Khadilkar et al., 2012]

- ▶ High scalability and fault-tolerance
- ▶ Possibly low performance since MapReduce is not suitable for graph processing

Partition-based Approaches

- (Offline) Partition an RDF data warehouse (graph) into several fragments that are distributed to sites
 - RDF data $D = \{D_1, \dots, D_n\}$
 - Allocate each D_i to a site

Partition-based Approaches

- (Offline) Partition an RDF data warehouse (graph) into several fragments that are distributed to sites
 - RDF data $D = \{D_1, \dots, D_n\}$
 - Allocate each D_i to a site
- Partitioning alternatives
 - Table-based (e.g., [Husain et al., 2011])
 - Graph-based (e.g., [Huang et al., 2011; Zhang et al., 2013])
 - Unit-based (e.g., [Gurajada et al., 2014; Lee and Liu, 2013])

Partition-based Approaches

- (Offline) Partition an RDF data warehouse (graph) into several fragments that are distributed to sites
 - RDF data $D = \{D_1, \dots, D_n\}$
 - Allocate each D_i to a site
- Partitioning alternatives
 - Table-based (e.g., [Husain et al., 2011])
 - Graph-based (e.g., [Huang et al., 2011; Zhang et al., 2013])
 - Unit-based (e.g., [Gurajada et al., 2014; Lee and Liu, 2013])
- (Online) SPARQL query decomposed $Q = \{Q_1, \dots, Q_k\} \Rightarrow$ query graph is decomposed
- Distributed execution of $\{Q_1, \dots, Q_k\}$ over $\{D_1, \dots, D_n\}$

Partition-based Approaches

- (Offline) Partition an RDF data warehouse (graph) into several fragments that are distributed to sites
 - RDF data $D = \{D_1, \dots, D_n\}$
 - Allocate each D_i to a site
- Partitioning alternatives
 - Table-based (e.g., [Husain et al., 2011])
 - Graph-based (e.g., [Huang et al., 2011; Zhang et al., 2013])
 - Unit-based (e.g., [Gurajada et al., 2014; Lee and Liu, 2013])
- (Online) SPARQL query decomposed $Q = \{Q_1, \dots, Q_k\} \Rightarrow$ query graph is decomposed
- Distributed execution of $\{Q_1, \dots, Q_k\}$ over $\{D_1, \dots, D_n\}$
- Examples: GraphPartition [Huang et al., 2011], WARP [Hose and Schenkel, 2013], Partout [Galarraga et al., 2014], Vertex-block [Lee and Liu, 2013]

Partition-based Approaches

- (Offline) Partition an RDF data warehouse (graph) into several fragments that are distributed to sites
 - RDF data $D = \{D_1, \dots, D_n\}$
 - Allocate each D_i to a site
- Partitioning alternatives
 - ▶ High performance
 - ▶ Great for parallelizing centralized RDF data
 - ▶ May not be possible to re-partition and re-allocate Web data (i.e., LOD)
 - ▶ Each approach requires a specific partitioning strategy – no generic partitioning
 - ▶ Query decomposition may not be easy

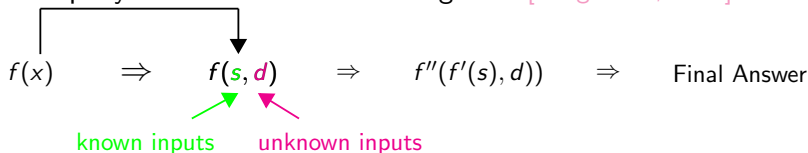
Liu, 2015]

Partial Query Evaluation (PQE)

- RDF data warehouse is partitioned and distributed as before
 - RDF data $D = \{D_1, \dots, D_n\}$
 - Allocate each D_i to a site
- SPARQL query is not decomposed
- Partial query evaluation – Distributed gStore [Peng et al., 2016]

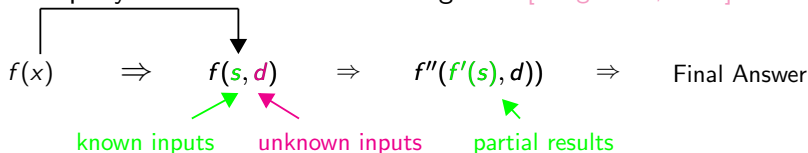
Partial Query Evaluation (PQE)

- RDF data warehouse is partitioned and distributed as before
 - RDF data $D = \{D_1, \dots, D_n\}$
 - Allocate each D_i to a site
- SPARQL query is not decomposed
- Partial query evaluation – Distributed gStore [Peng et al., 2016]



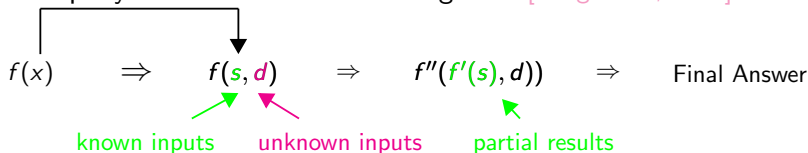
Partial Query Evaluation (PQE)

- RDF data warehouse is partitioned and distributed as before
 - RDF data $D = \{D_1, \dots, D_n\}$
 - Allocate each D_i to a site
- SPARQL query is not decomposed
- Partial query evaluation – Distributed gStore [Peng et al., 2016]



Partial Query Evaluation (PQE)

- RDF data warehouse is partitioned and distributed as before
 - RDF data $D = \{D_1, \dots, D_n\}$
 - Allocate each D_i to a site
- SPARQL query is not decomposed
- Partial query evaluation – Distributed gStore [Peng et al., 2016]



- Query is the function and each D_i is the known input

- RDF data warehouse is partitioned and distributed
 - RDF data $D = \{D_1, \dots, D_n\}$
 - Allocate each D_i to a site

- RDF data warehouse is partitioned and distributed
 - RDF data $D = \{D_1, \dots, D_n\}$
 - Allocate each D_i to a site
- Partitioning alternatives
 - Table-based (e.g., [Husain et al., 2011])
 - Graph-based (e.g., [Huang et al., 2011; Zhang et al., 2013])
 - Unit-based (e.g., [Gurajada et al., 2014; Lee and Liu, 2013])

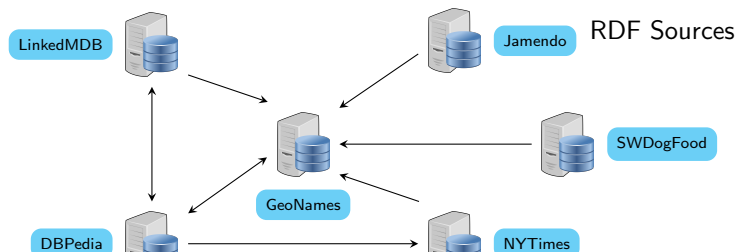
- RDF data warehouse is partitioned and distributed
 - RDF data $D = \{D_1, \dots, D_n\}$
 - Allocate each D_i to a site
- Partitioning alternatives
 - Table-based (e.g., [Husain et al., 2011])
 - Graph-based (e.g., [Huang et al., 2011; Zhang et al., 2013])
 - Unit-based (e.g., [Gurajada et al., 2014; Lee and Liu, 2013])
- SPARQL query decomposed $Q = \{Q_1, \dots, Q_k\}$
- Distributed execution of $\{Q_1, \dots, Q_k\}$ over $\{D_1, \dots, D_n\}$

- RDF data warehouse is partitioned and distributed
 - RDF data $D = \{D_1, \dots, D_n\}$
 - Allocate each D_i to a site
- Partitioning alternatives
 - Table-based (e.g., [Husain et al., 2011])
 - Graph-based (e.g., [Huang et al., 2011; Zhang et al., 2013])
 - Unit-based (e.g., [Gurajada et al., 2014; Lee and Liu, 2013])
- SPARQL query decomposed $Q = \{Q_1, \dots, Q_k\}$
- Distributed execution of $\{Q_1, \dots, Q_k\}$ over $\{D_1, \dots, D_n\}$

- ▶ High performance
- ▶ Great for parallelizing centralized RDF data
- ▶ May not be possible to re-partition and re-allocate Web data (i.e., LOD)
- ▶ Query decomposition may not be easy

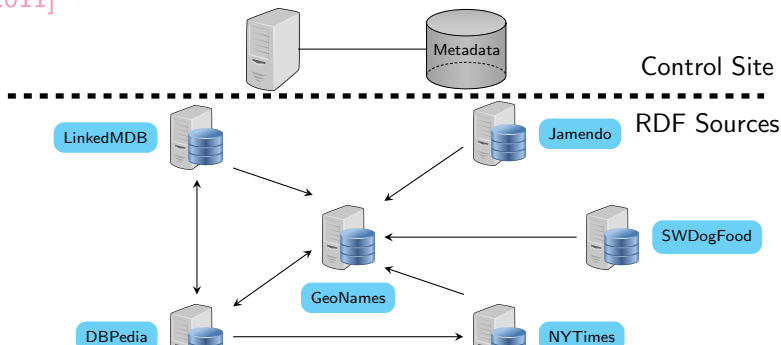
SPARQL Endpoint Federation

- No data re-partitioning/re-distribution
- Consider $D = D_1 \cup D_2 \cup \dots \cup D_n$; D_i : SPARQL endpoint
- SPARQL query decomposed $Q = \{Q_1, \dots, Q_k\}$
- Distributed execution of $\{Q_1, \dots, Q_k\}$ over $\{D_1, \dots, D_n\}$
- E.g.: SPLENDID [Görlitz and Staab, 2011], ANAPSID [Acosta et al., 2011]



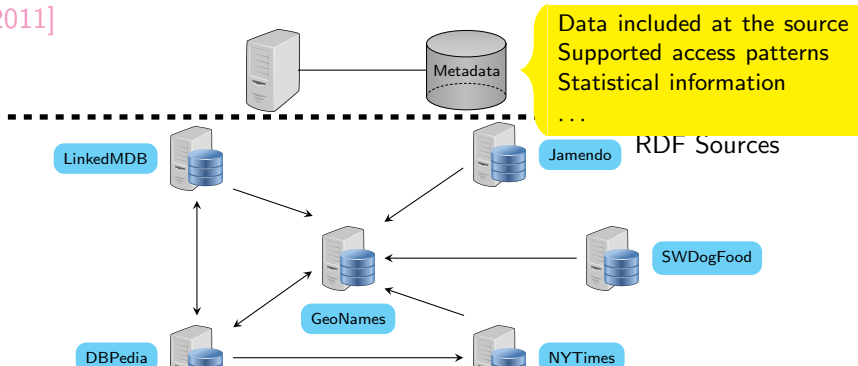
SPARQL Endpoint Federation

- No data re-partitioning/re-distribution
- Consider $D = D_1 \cup D_2 \cup \dots \cup D_n$; D_i : SPARQL endpoint
- SPARQL query decomposed $Q = \{Q_1, \dots, Q_k\}$
- Distributed execution of $\{Q_1, \dots, Q_k\}$ over $\{D_1, \dots, D_n\}$
- E.g.: SPLENDID [Görlitz and Staab, 2011], ANAPSID [Acosta et al., 2011]



SPARQL Endpoint Federation

- No data re-partitioning/re-distribution
- Consider $D = D_1 \cup D_2 \cup \dots \cup D_n$; D_i : SPARQL endpoint
- SPARQL query decomposed $Q = \{Q_1, \dots, Q_k\}$
- Distributed execution of $\{Q_1, \dots, Q_k\}$ over $\{D_1, \dots, D_n\}$
- E.g.: SPLENDID [Görlitz and Staab, 2011], ANAPSID [Acosta et al., 2011]



SPARQL Endpoint Federation

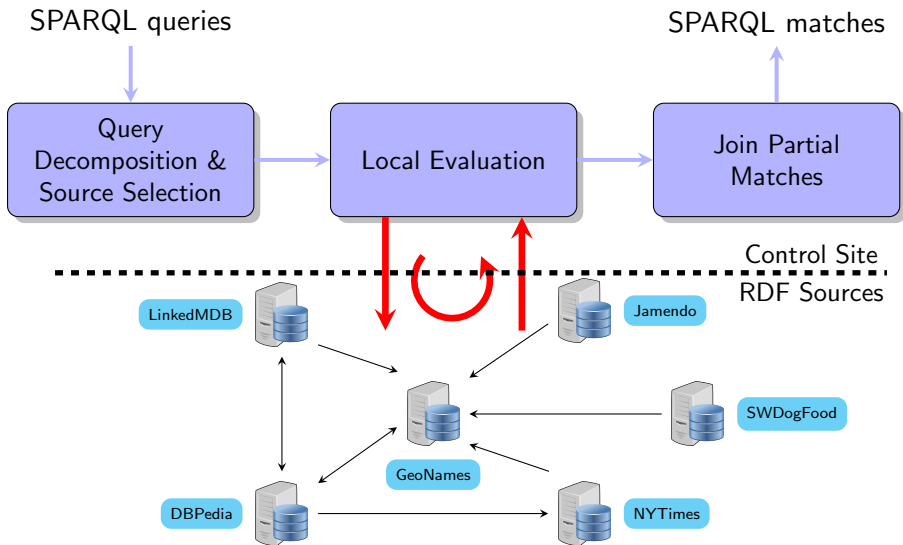
- No data re-partitioning/re-distribution
- Consider $D = D_1 \cup D_2 \cup \dots \cup D_n$; D_i : SPARQL endpoint
- SPARQL query decomposed $Q = \{Q_1, \dots, Q_k\}$
- Distributed execution of $\{Q_1, \dots, Q_k\}$ over $\{D_1, \dots, D_n\}$
- E.g.: SPLENDID [Görlitz and Staab, 2011], ANAPSID [Acosta et al., 2011]



- ▶ Data integration approach
- ▶ May be the only way to proceed if RDF data is already distributed with autonomous owners
- ▶ Not all RDF data storage points are SPARQL endpoints

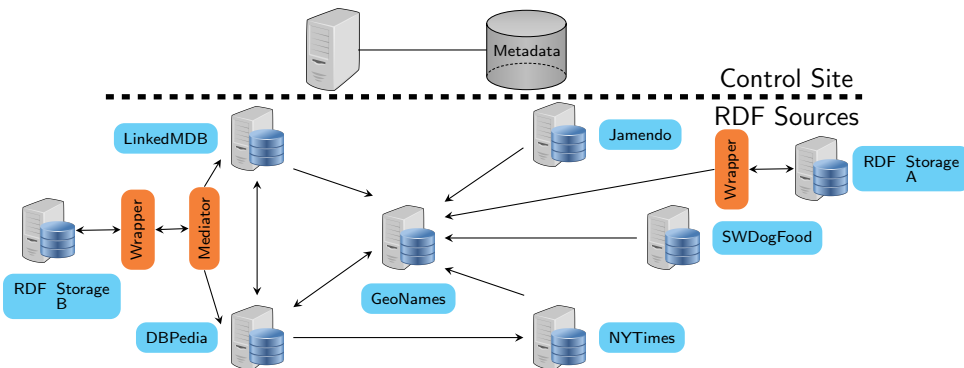


Federated Query Processing



Not All RDF Storage Sites are SPARQL Endpoints

- Use the mediator-wrapper paradigm
- Wrappers provide SPARQL endpoint functionality
- Mediators may be introduced if wrappers are thin
- E.g.: DARQ [Quilitz and Leser, 2008], FedX [Schwarte et al., 2011]



UniProt Federation – EBI RDF Platform



Curated computational models of biological processes



Sample information for reference samples and samples for which data exist in one of the EBI's assay databases



Curated chemical database of bioactive molecules with drug-like properties



Genome databases for vertebrates and other eukaryotic species



Gene expression data from the Gene Expression Atlas



Curated and peer-reviewed pathways



Federated Access to UniProt Collection

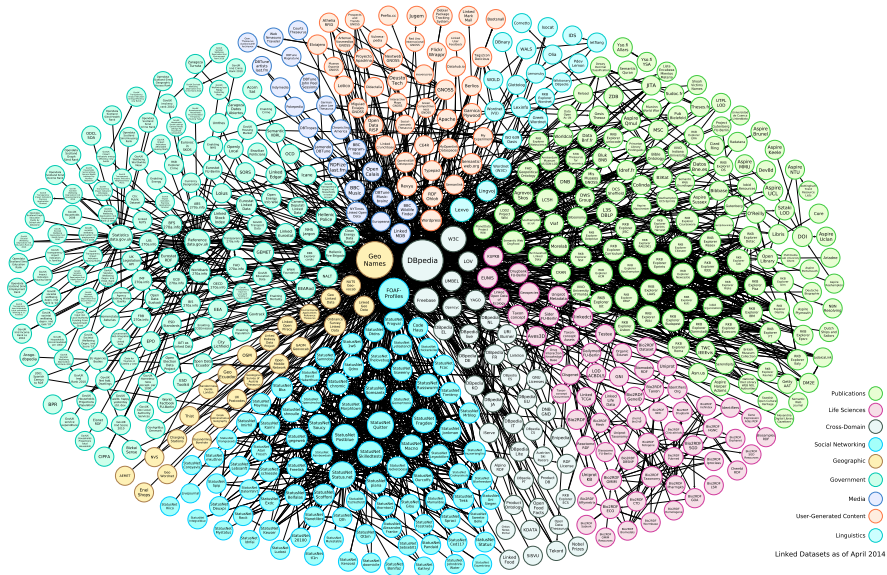
Get the Reactome pathways where Q16850 is associated, then get all the other proteins in that pathway and pull out their expression from the atlas, along with the GO annotations from UniProt

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX biopax3: <http://www.biopax.org/release/biopax-level3.owl#>
PREFIX atlasterms: <http://rdf.ebi.ac.uk/terms/atlas/>
PREFIX upc:<http://purl.uniprot.org/core/>
SELECT DISTINCT ?pathwayname ?expressionValue ?golabel
WHERE {
  # Get the pathways that reference Q16850
  ?pathway rdf:type biopax3:Pathway .
  ?pathway biopax3:displayName ?pathwayname .
  ?pathway biopax3:pathwayComponent
  [?rel [biopax3:entityReference ?dbXref]] .
  ?pathway biopax3:pathwayComponent
  [?rel [biopax3:entityReference <http://purl.uniprot.org/uniprot/Q16850>]] .

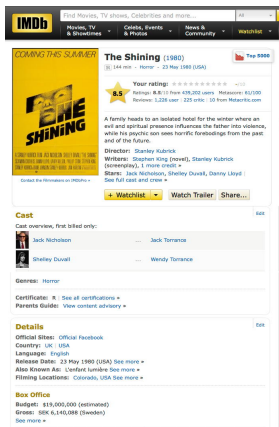
  # Get the expression for those proteins
  SERVICE <http://www.ebi.ac.uk/rdf/services/atlas/sparql> {
    ?value rdfs:label ?expressionValue .
    ?value atlasterms:pValue ?pvalue .
    ?value atlasterms:isMeasurementOf ?probe .
    ?probe atlasterms:dbXref ?dbXref .
  }

  # get the GO functions from Uniprot
  SERVICE <http://uniprot.org/sparql> {
    ?dbXref a upc:Protein ;
    upc:classifiedWith ?keyword .
    ?keyword rdfs:seeAlso ?goid .
    ?goid rdfs:label ?golabel .
  }
}
```

Linked Object Data – Closer Look



Traditional Hypertext-based Web Access



IMDb

Find Movies, TV shows, Celebrities and more...

Home, TV & Showtimes Celebs, Events & Photos News & Community Watchlist

COMING THIS SUMMER

The Shining (1980)

144 min - Horror - 23 May 1980 (USA) **Top 500**

Your rating: ***** **8.5**
Ratings: 8,610 from 438,202 users Metascore: 83/100
Reviews: 3,228 critic 225 critic 10 from Metacritic.com

A family heads to an isolated hotel for the winter where an evil and spiritual presence influences the father into violence, while his psychic son sees horrific forebodings from the past and of the future.

Director: Stanley Kubrick
Writers: Stephen King (novel), Stanley Kubrick (screenplay), L. Mowbray (screenplay)
Stars: Jack Nicholson, Shelley Duvall, Danny Lloyd
See full cast and crew »

Cast
Cast overview, first billed only:
Jack Nicholson ... Jack Torrance
Shelley Duvall ... Wendy Torrance

Genres: Horror

Certificates: R See all certifications »
Parents Guide: View content advisory »

Details
Official Sites: Official Facebook
Country: UK, USA
Language: English
Release Date: 23 May 1980 (USA) See more »
Also Known As: L'enfer lumine See more »
Filming Locations: Colorado, USA See more »

Box Office
Budget: \$15,000,000 (estimated)
Gross: 105,634,018 (Sweden)
See more »

Data exposed
to the Web
via HTML



THE WORLD FACTBOOK

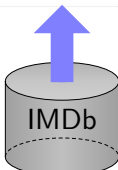
Please select a country to view

UNIQUE UNITED KINGDOM

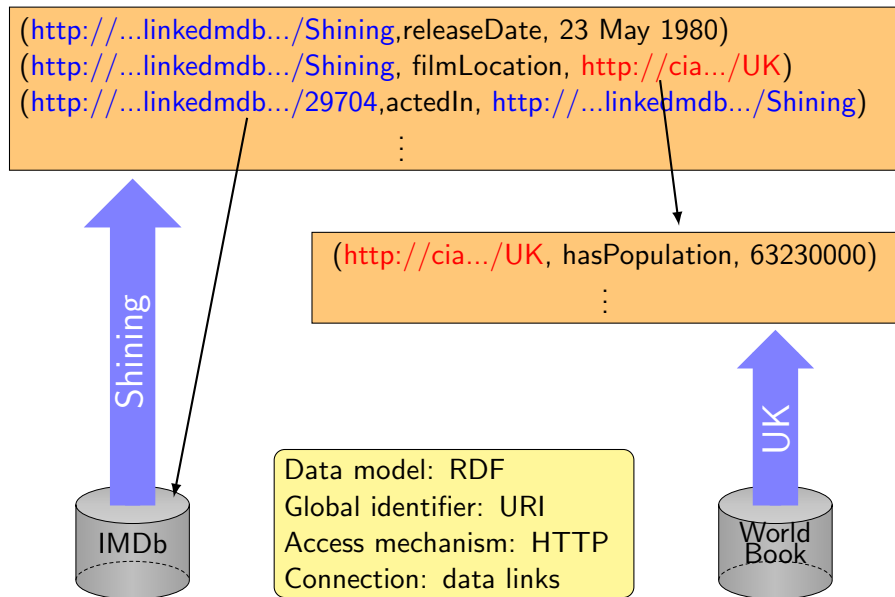
First published on June 10, 2010

Background:
The United Kingdom has historically played a leading role in developing parliamentary democracy and in advancing literature and science. At its zenith in the 19th century, the British Empire stretched over one-fourth of the earth's surface. The first half of the 20th century saw the UK's strength seriously depleted by two world wars and the Irish Republic's withdrawal from the union. The second half witnessed the dismantling of the Empire and the UK rebuilding itself into a modern and prosperous European nation. As one of the permanent members of the UN Security Council and a founding member of NATO and the Commonwealth, the UK promotes a global approach to foreign policy. The UK is also an active member of the EU, although it chose to remain outside the Economic and Monetary Union. The Scottish Parliament, the National Assembly for Wales, and the Northern Ireland Assembly were established in 1999. The latter was suspended until May 2007 due to ongoing power process, but devolution was fully completed in March 2010.

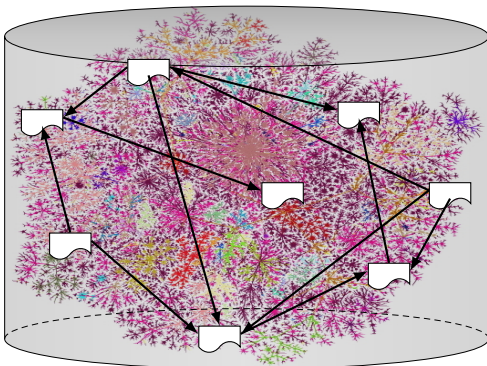
Geography:
Location: Western Europe, islands - including the northern one-sixth of the island of Ireland - between the North Atlantic Ocean and the North Sea, northwest of France
Geographic coordinates: 54 00 N, 0 00 W
Map references: Europe
Area:
Total: 243,610 sq km
country comparison to the world: 66



Linked Data Publishing Principles



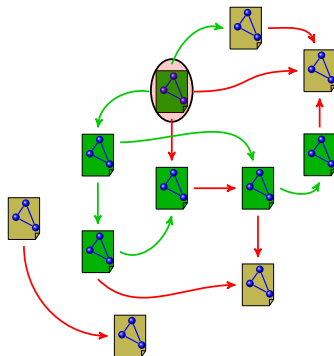
Live Query Processing



- Not all data resides at SPARQL endpoints
- Freshness of access to data important
- Potentially countably infinite data sources
- Live querying
 - On-line execution
 - Only rely on linked data principles
- Alternatives
 - Traversal-based approaches
 - Index-based approaches
 - Hybrid approaches

Traversal Approaches

- Discover relevant URLs recursively by traversing (specific) data links at query execution runtime [Hartig, 2013; Ladwig and Tran, 2011]
- Implements reachability-based query semantics
 - Start from a set of seed URLs
 - Recursively follow and discover new URLs
- Important issue is selection of seed URLs
- Retrieved data serves to discover new URLs and to construct result



Traversal Approaches

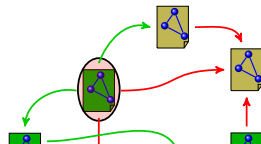
- Discover relevant URIs recursively by traversing (specific) data links at query execution runtime [Hartig, 2013; Ladwig and Tran, 2011]
- Implements reachability-based query

Advantages

Easy to implement.

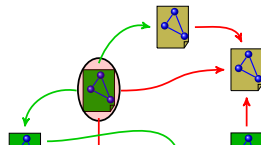
No data structure to maintain.

- Important issue is selection of seed URIs
- Retrieved data serves to discover new URIs and to construct result



Traversal Approaches

- Discover relevant URLs recursively by traversing (specific) data links at query execution runtime [Hartig, 2013; Ladwig and Tran, 2011]
- Implements reachability-based query



Advantages

Easy to implement.

No data structure to maintain.

Disadvantages

Possibilities for parallelized data retrieval are limited

Repeated data retrieval introduces significant query latency.



References I

- Abadi, D. J., Marcus, A., Madden, S., and Hollenbach, K. (2009). SW-Store: a vertically partitioned DBMS for semantic web data management. *VLDB J.*, 18(2):385–406.
- Abadi, D. J., Marcus, A., Madden, S. R., and Hollenbach, K. (2007). Scalable semantic web data management using vertical partitioning. In *Proc. 33rd Int. Conf. on Very Large Data Bases*, pages 411–422.
- Acosta, M., Vidal, M.-E., Lampo, T., Castillo, J., and Ruckhaus, E. (2011). ANAPSID: an adaptive query processing engine for SPARQL endpoints. In *Proc. 10th Int. Semantic Web Conf.*, pages 18–34.
- Aluç, G., Özsu, M. T., Daudjee, K., and Hartig, O. (2013). chameleon-db: a workload-aware robust RDF data management system. Technical Report CS-2013-10, University of Waterloo. Available at <https://cs.uwaterloo.ca/sites/ca.computer-science/files/uploads/files/CS-2013-10.pdf>.
- Bornea, M. A., Dolby, J., Kementsietsidis, A., Srinivas, K., Dantressangle, P., Udrea, O., and Bhattacharjee, B. (2013). Building an efficient RDF store over a relational database. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 121–132. Available from: <http://doi.acm.org/10.1145/2463676.2463718>.

References II

- Bu, Y., Howe, B., Balazinska, M., and Ernst, M. D. (2010). HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endowment*, 3(1):285–296. Available from: <http://dl.acm.org/citation.cfm?id=1920841.1920881>.
- Bu, Y., Howe, B., Balazinska, M., and Ernst, M. D. (2012). The HaLoop approach to large-scale iterative data analysis. *VLDB J.*, 21(2):169–190.
- Chen, C., Yan, X., Zhu, F., Han, J., and Yu, P. S. (2008). Graph OLAP: Towards online analytical processing on graphs. In *Proc. 8th IEEE Int. Conf. on Data Mining*, pages 103–112.
- Chen, R., Shi, J., Chen, Y., and Chen, H. (2015). Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proc. 10th ACM SIGOPS/EuroSys European Conf. on Comp. Syst.*, pages 1:1–1:15. Available from: <http://doi.acm.org/10.1145/2741948.2741970>.
- Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113. Available from: <http://doi.acm.org/10.1145/1327452.1327492>.

References III

- Galarraga, L., Hose, K., and Schenkel, R. (2014). Partout: a distributed engine for efficient RDF processing. In *Proc. 26th Int. World Wide Web Conf. (Companion Volume)*, pages 267–268.
- Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. (2012). PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proc. 10th USENIX Symp. on Operating System Design and Implementation*, pages 17–30. Available from: <http://dl.acm.org/citation.cfm?id=2387880.2387883>.
- Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. (2014). GraphX: graph processing in a distributed dataflow framework graph processing in a distributed dataflow framework. In *Proc. 11th USENIX Symp. on Operating System Design and Implementation*, pages 599–613. Available from: <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez>.
- Görlitz, O. and Staab, S. (2011). SPLENDID: SPARQL endpoint federation exploiting VOID descriptions. In *Proc. 2nd Int. Workshop on Consuming Linked Data*.
- Gurajada, S., Seufert, S., Miliaraki, I., and Theobald, M. (2014). TriAD: A distributed shared-nothing RDF engine based on asynchronous message passing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 289–300.

References IV

- Han, M. (2015). On improving distributed Pregel-like graph processing systems. Master's thesis, University of Waterloo, David R. Cheriton School of Computer Science.
- Han, M. and Daudjee, K. (2015). Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. *Proc. VLDB Endowment*, 8(9):950–961. Available from: <http://www.vldb.org/pvldb/vol8/p950-han.pdf>.
- Hartig, O. (2013). SQUIN: a traversal based query execution system for the web of linked data. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 1081–1084. Available from: <http://doi.acm.org/10.1145/2463676.2465231>.
- Hose, K. and Schenkel, R. (2013). WARP: Workload-aware replication and partitioning for RDF. In *Proc. Workshops of 29th Int. Conf. on Data Engineering*, pages 1–6.
- Huang, J., Abadi, D. J., and Ren, K. (2011). Scalable SPARQL querying of large RDF graphs. *Proc. VLDB Endowment*, 4(11):1123–1134.
- Husain, M. F., McGlothlin, J., Masud, M. M., Khan, L. R., and Thuraisingham, B. (2011). Heuristics-based query processing for large RDF graphs using cloud computing. *IEEE Trans. Knowl. and Data Eng.*, 23(9):1312–1327.
- Kaoudi, Z. and Manolescu, I. (2015). RDF in the clouds: A survey. *VLDB J.*, 24:67–91.

- Karypis, G. and Kumar, V. (1995). Multilevel graph partitioning schemes. In *Proc. 1995 Int. Conf. on Parallel Processing*, pages 113–122.
- Khadilkar, V., Kantarcioglu, M., Thuraisingham, B. M., and Castagna, P. (2012). Jena-HBase: A distributed, scalable and efficient RDF triple store. In *Proc. International Semantic Web Conference Posters & Demos Track*.
- Kiveris, R., Lattanzi, S., Mirrokni, V., Rastogi, V., and Vassilvitskii, S. (2014). Connected components in MapReduce and beyond. In *Proc. 5th ACM Symp. on Cloud Computing*, pages 18:1–18:13.
- Ladwig, G. and Tran, T. (2011). SIHJoin: Querying remote and local linked data. In *Proc. 8th Extended Semantic Web Conf.*, pages 139–153.
- Lee, K. and Liu, L. (2013). Scaling queries over big RDF graphs with semantic hash partitioning. *Proc. VLDB Endowment*, 6(14):1894–1905. Available from: <http://www.vldb.org/pvldb/vol6/p1894-lee.pdf>.
- Li, F., Ooi, B. C., Özsu, M. T., and Wu, S. (2014). Distributed data management using MapReduce. *ACM Comput. Surv.*, 46(3):Article No. 31.
- Lin, J. (2018). Scale up or scale out for graph processing? *IEEE Internet Comput.*, 22(3):72–78.

References VI

- Maali, F., Campinas, S., and Decker, S. (2015). Gagg: A graph aggregation operator. In *Proc. 14th Int. Semantic Web Conf.*, pages 491–504.
- Michiardi, P. (2015). Introduction to Spark internals. Slideshare. Available from: <http://bit.ly/2pdGLor> [Last retrieved: 9 July 2015].
- Neumann, T. and Weikum, G. (2008). RDF-3X: a RISC-style engine for RDF. *Proc. VLDB Endowment*, 1(1):647–659.
- Neumann, T. and Weikum, G. (2009). The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113.
- Peng, P., Zou, L., Özsu, M. T., Chen, L., and Zhao, D. (2016). Processing SPARQL queries over distributed RDF graphs. *VLDB J.*, 25(2):243–268.
- Qin, L., Yu, J. X., Chang, L., Cheng, H., Zhang, C., and Lin, X. (2014). Scalable big graph processing in mapreduce. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 827–838. Available from: <http://doi.acm.org/10.1145/2588555.2593661>.
- Quilitz, B. and Leser, U. (2008). Querying distributed RDF data sources with SPARQL. In *Proc. 5th European Semantic Web Conf.*, pages 524–538.

References VII

- Rastogi, V., Machanavajjhala, A., Chitnis, L., and Sarma, A. D. (2013). Finding connected components in map-reduce in logarithmic rounds. In *Proc. 29th Int. Conf. on Data Engineering*, pages 50–61.
- Rohloff, K. and Schantz, R. E. (2010). High-performance, massively scalable distributed systems using the mapreduce software framework: the shard triple-store. In *Proc. Int. Workshop on Programming Support Innovations for Emerging Distributed Applications*. Article No. 4.
- Roy, A., Mihailovic, I., and Zwaenepoel, W. (2013). X-stream: edge-centric graph processing using streaming partitions. In *Proc. 24th ACM Symp. on Operating System Principles*, pages 472–488. Available from: <http://doi.acm.org/10.1145/2517349.2522740>.
- Schwarte, A., Haase, P., Hose, K., Schenkel, R., and Schmidt, M. (2011). Fedx: Optimization techniques for federated query processing on linked data. In *Proc. 10th Int. Semantic Web Conf.*, pages 601–616. Available from: https://doi.org/10.1007/978-3-642-25073-6_38.
- Tian, Y., Balmin, A., Corsten, S. A., Tatikonda, S., and McPherson, J. (2013). From “think like a vertex” to “think like a graph”. *Proc. VLDB Endowment*, 7(3):193–204. Available from: <http://www.vldb.org/pvldb/vol7/p193-tian.pdf>.

References VIII

- Tian, Y., Hankins, R. A., and Patel, J. M. (2008). Efficient aggregation for graph summarization. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 567–580.
- Ugander, J. and Backstrom, L. (2013). Balanced label propagation for partitioning massive graphs. In *Proc. 6th ACM Int. Conf. Web Search and Data Mining*, pages 507–516. Available from: <http://doi.acm.org/10.1145/2433396.2433461>.
- Valiant, L. G. (1990). A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111.
- Wang, G., Xie, W., Demers, A. J., and Gehrke, J. (2013). Asynchronous large-scale graph processing made easy. In *Proc. 6th Biennial Conf. on Innovative Data Systems Research*.
- Wang, Z., Fan, Q., Wang, H., Tan, K.-L., Agrawal, D., and El-Abbadi, A. (2014). Pagrol: PArallel GRaph OLap over lege-scale attributed graphs. In *Proc. 30th Int. Conf. on Data Engineering*, pages 496–507.
- Weiss, C., Karras, P., and Bernstein, A. (2008). Hexastore: sextuple indexing for semantic web data management. *Proc. VLDB Endowment*, 1(1):1008–1019.

References IX

- Wilkinson, K. (2006). Jena property table implementation. Technical Report HPL-2006-140, HP Laboratories Palo Alto.
- Yan, D., Cheng, J., Lu, Y., and Ng, W. (2014). Blogel: A block-centric framework for distributed computation on real-world graphs. *Proc. VLDB Endowment*, 7(14):1981–1992. Available from: <http://www.vldb.org/pvldb/vol7/p1981-yan.pdf>.
- Yuan, P., Liu, P., Wu, B., Jin, H., Zhang, W., and Liu, L. (2013). TripleBit: a fast and compact system for large scale RDF data. *Proc. VLDB Endowment*, 6(7):517–528. Available from: <http://www.vldb.org/pvldb/vol6/p517-yuan.pdf>.
- Zaharia, M. (2016). *An Architecture for Fast and General Data Processing on Large Clusters*. ACM Books.
- Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. 9th USENIX Symp. on Networked Systems Design & Implementation*, pages 2–2.

- Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. In *Proc. 2nd USENIX Workshop on Hot Topics in Cloud Computing*, pages 10–10.
- Zhang, X., Chen, L., Tong, Y., and Wang, M. (2013). EAGRE: towards scalable I/O efficient SPARQL query evaluation on the cloud. In *Proc. 29th Int. Conf. on Data Engineering*, pages 565–576.
- Zhao, P., Li, X., Xin, D., and Han, J. (2011). Graph cube: on warehousing and OLAP multidimensional networks. In *Proc. ACM International Conference on Management of Data*, pages 853–864.
- Zou, L., Mo, J., Chen, L., Özsu, M. T., and Zhao, D. (2011). gStore: answering SPARQL queries via subgraph matching. *Proc. VLDB Endowment*, 4(8):482–493.
- Zou, L. and Özsu, M. T. (2017). Graph-based RDF data management. *Data Science and Engineering*, 2(1):56–70. Available from: <https://dx.doi.org/10.1007/s41019-016-0029-6>.
- Zou, L., Özsu, M. T., Chen, L., Shen, X., Huang, R., and Zhao, D. (2014). gStore: A graph-based SPARQL query engine. *VLDB J.*, 23(4):565–590.