Data Management
10 NoSQL Systems

Matthias Boehm

Graz University of Technology, Austria
Computer Science and Biomedical Engineering
Institute of Interactive Systems and Data Science
BMK endowed chair for Data Management

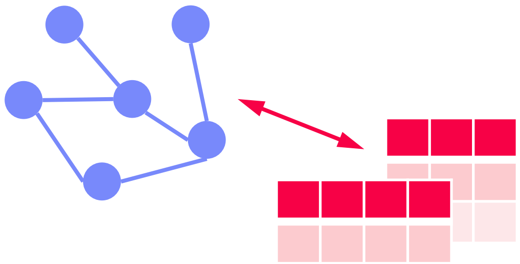
Last update: May 29, 2021

Announcements/Org

- **#1 Video Recording**
  - Link in TeachCenter & TUbe (lectures will be public)
  - https://tugraz.webex.com/meet/m.boehm
  - Corona traffic light **RED** → May 17: **ORANGE** (but tests required)
- **#2 Reminder Communication**
  - **Newsgroup:** news://news.tugraz.at/tu-graz.lv.dbase
  - **Office hours:** Mo 12.30-1.30pm (https://tugraz.webex.com/meet/m.boehm)
- **#3 Exercises/Exams**
  - **Grading:** Exercise 1 - done, Exercise 2 – this/next week
  - **Submission:** **Exercise 3:** last chance Jun 1, **Exercise 4:** published May 29
  - **Exams:** Jun 30 5.30pm (i11, i12, i13), Jul 5 3.30pm (i13), 6.30 (i13)
- **#4 Course Evaluation**
  - Please participate; open period: **June 1 – July 15**

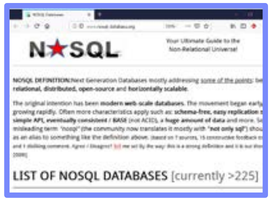
SQL vs NoSQL Motivation

- **#1 Data Models/Schema**
  - **Non-relational:** key-value, graph, doc, time series (logs, social media, documents/media, sensors)
  - Impedance mismatch / complexity
  - **Pay-as-you-go/schema-free** (flexible/implicit)



- **#2 Scalability**
  - Scale-up vs simple scale-out
  - Horizontal partitioning (sharding) and scaling
  - **Commodity hardware, network, disks** ($)

- **NoSQL Evolution**
  - Late 2000s: Non-relational, distributed, open source DBMSs
  - Early 2010s: NewSQL: modern, distributed, relational DBMSs
  - Not Only SQL: combination with relational techniques
  - ➔ **RDBMS and specialized systems** (consistency/data models)



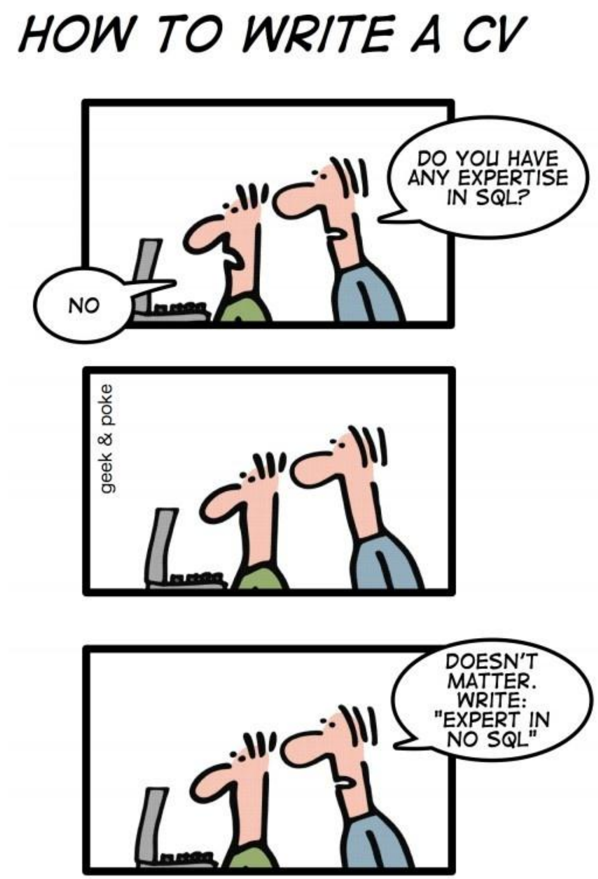
[**Credit:** http://nosql-database.org/]

Agenda

- Consistency and Data Models
- Key-Value Stores
- Document Stores
- Graph Processing
- Time Series Databases
- Exercise 4: Large-Scale Data Analysis

Lack of standards and **imprecise classification**

[Wolfram Wingerath, Felix Gessert, Norbert Ritter: NoSQL & Real-Time Data Management in Research & Practice. **BTW 2019**]



Leverage the NoSQL boom

[http://geek-and-poke.com/]

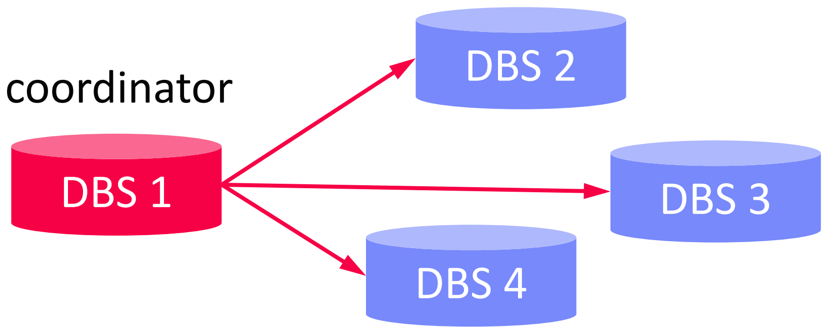
Consistency and Data Models

Recap: ACID Properties

- **Atomicity**
  - A transaction is executed atomically (**completely or not at all**)
  - If the transaction fails/aborts no changes are made to the database (**UNDO**)
- **Consistency**
  - A successful transaction ensures that all **consistency constraints are met** (referential integrity, semantic/domain constraints)
- **Isolation**
  - Concurrent transactions are executed in isolation of each other
  - **Appearance of serial transaction execution**
- **Durability**
  - **Guaranteed persistence** of all changes made by a successful transaction
  - In case of system failures, the database is recoverable (**REDO**)

Two-Phase Commit (2PC) Protocol

- **Distributed TX Processing**
  - N nodes with logically related but physically distributed data (e.g., vertical data partitioning)
  - **Distributed TX processing to ensure consistent view** (atomicity/durability)
- **Two-Phase Commit** (via 2N msgs)
  - **Phase 1 PREPARE:** check for successful completion, logging
  - **Phase 2 COMMIT:** release locks, and other cleanups
  - **Problem: Blocking protocol**



- **Excursus: Wedding Analogy**
  - Coordinator: marriage registrar
  - **Phase 1:** Ask for willingness
  - **Phase 2:** If all willing, declare marriage

CAP Theorem

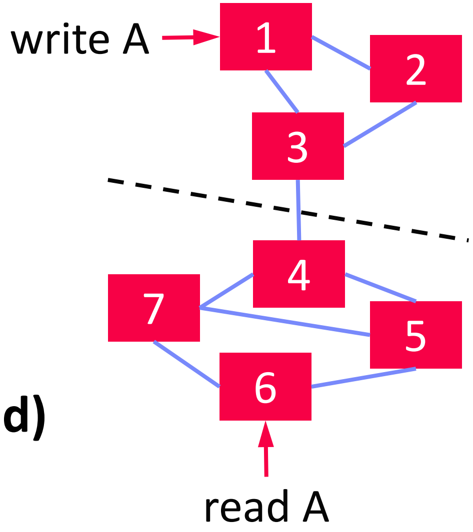
- **Consistency**
  - Visibility of updates to distributed data (atomic or linearizable consistency)
  - Different from ACIDs consistency in terms of integrity constraints
- **Availability**
  - Responsiveness of a services (clients reach available service, **read/write**)
- **Partition Tolerance**
  - Tolerance of temporarily unreachable network partitions
  - System characteristics (e.g., latency) maintained
- **CAP Theorem** ***"You can have AT MOST TWO of these properties for a networked shared-data systems."***

[Eric A. Brewer: Towards robust distributed systems (abstract). **PODC 2000**]

- **Proof**

[Seth Gilbert, Nancy A. Lynch: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. **SIGACT News 2002**]

CAP Theorem, cont.



- **CA: Consistency & Availability (ACID single node)**
  - Network partitions cannot be tolerated
  - Visibility of updates (consistency) in conflict with availability ➔ **no distributed systems**

- **CP: Consistency & Partition Tolerance (ACID distributed)**
  - Availability cannot be guaranteed
  - **On connection failure, unavailable** (wait for overall system to become consistent)

- **AP: Availability & Partition Tolerance (BASE)**
  - Consistency cannot be guaranteed, use of optimistic strategies
  - Simple to implement, main concern: availability to ensure revenue ($$$)

➔ **BASE consistency model**

BASE Properties

- **Basically Available**
  - **Major focus on availability**, potentially with outdated data
  - No guarantee on global data consistency across entire system
- **Soft State**
  - Even without explicit state updates, the **data might change** due to asynchronous propagation of updates and nodes that become available
- **Eventual Consistency**
  - Updates eventually propagated, system would reach consistent state if no further updates, and network partitions fixed
  - No temporal guarantees on changes are propagated

Eventual Consistency

[Peter Bailis, Ali Ghodsi: Eventual consistency today: limitations, extensions, and beyond. **Commun. ACM 2013**]

- **Basic Concept**
  - Changes made to a copy eventually migrate to all
  - If update activity stops, replicas will **converge to a logically equivalent state**
  - **Metric:** time to reach consistency (probabilistic bounded staleness)

Amazon SimpleDB	500ms
Cassandra	200ms
Amazon S3	12s

- **#1 Monotonic Read Consistency**
  - After reading data object A, the client never reads an older version
- **#2 Monotonic Write Consistency**
  - After writing data object A, it will never be replaced with an older version
- **#3 Read Your Own Writes / Session Consistency**
  - After writing data object A, a client never reads an older version
- **#4 Causal Consistency**
  - If client 1 communicated to client 2 that data object A has been updated, subsequent reads on client 2 return the new value

Key-Value Stores

Motivation and Terminology

- **Motivation**
  - **Basic key-value mapping via simple API** (more complex data models can be mapped to key-value representations)
  - **Reliability at massive scale on commodity HW** (cloud computing)
- **System Architecture**
  - **Key**-value maps, where values can be of a variety of data types
  - APIs for CRUD operations (create, read, update, delete)
  - Scalability via sharding (horizontal partitioning)

users:1:a	"Inffeldgasse 13, Graz"
users:1:b	"[12, 34, 45, 67, 89]"
users:2:a	"Mandellstraße 12, Graz"
users:2:b	"[12, 212, 3212, 43212]"

- **Example Systems**
  - **Dynamo** (2007, AP) → **Amazon DynamoDB** (2012)
  - **Redis** (2009, CP/AP)

[Giuseppe DeCandia et al: Dynamo: amazon's highly available **key-value store**. **SOSP 2007**]

Example Systems

- **Redis Data Types**
  - Redis is not a plain KV-store, but "data structure server" with persistent log (**appendfsync no/everysec/always**)
  - **Key:** ASCII string (max 512MB, common key schemes: comment:1234:reply.to)
  - **Values:** strings, lists, sets, sorted sets, hashes (map of string-string), etc

- **Redis APIs**
  - **SET/GET/DEL:** insert a key-value pair, lookup value by key, or delete by key
  - **MSET/MGET:** insert or lookup multiple keys at once
  - **INCRBY/DECBY:** increment/decrement counters
  - Others: EXISTS, LPUSH, LPOP, LRANGE, LTRIM, LLEN, etc

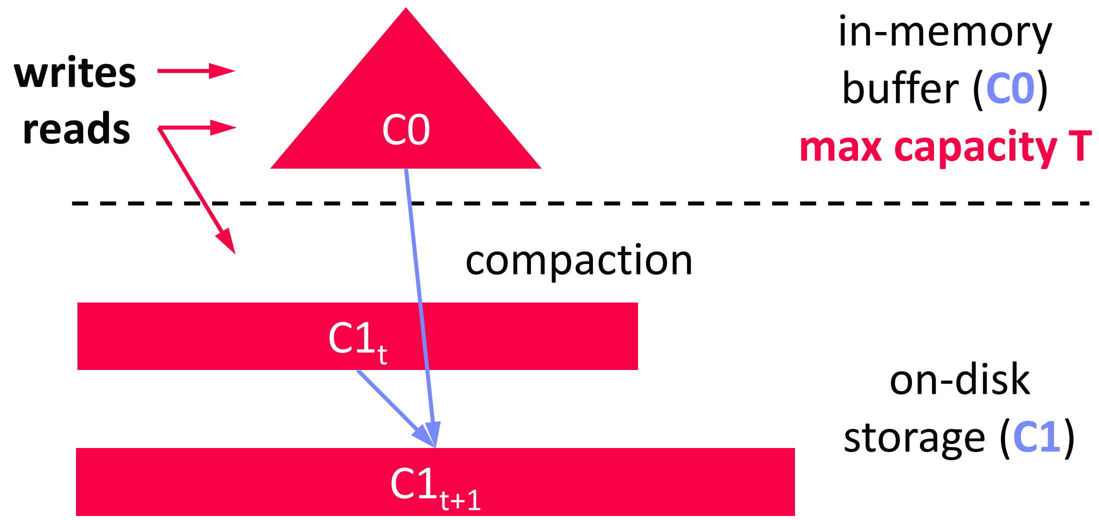
- **Other systems**
  - Classic KV stores (AP): **Riak**, **Aerospike**, **Voldemort**, **LevelDB**, **RocksDB**, **FoundationDB**, **Memcached**
  - Wide-column stores: **Google BigTable** (CP), **Apache HBase** (CP), **Apache Cassandra** (AP)

Log-structured Merge Trees

[Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, Elizabeth J. O'Neil: The Log-Structured Merge-Tree (LSM-Tree). **Acta Inf. 1996**]

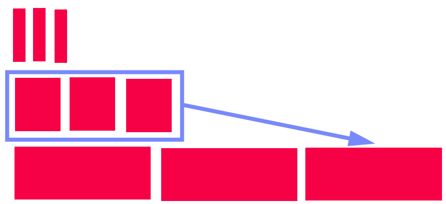
- **LSM Overview**
  - Many KV-stores rely on LSM-trees as their storage engine (e.g., **BigTable**, **DynamoDB**, **LevelDB**, **Riak**, **RocksDB**, **Cassandra**, **HBase**)
  - **Approach:** Buffers writes in memory, flushes data as sorted runs to storage, merges runs into larger runs of next level (compaction)

- **System Architecture**
  - Writes in C0
  - Reads against C0 and C1 (w/ buffer for C1)
  - Compaction (rolling merge): sort, merge, including **deduplication**

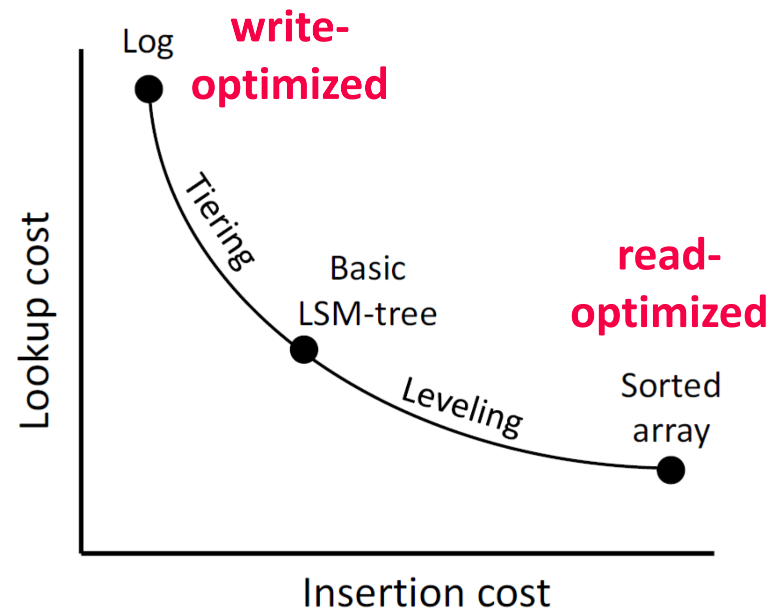


Log-structured Merge Trees, cont.

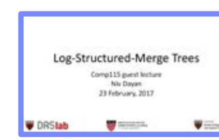
- **LSM Tiering**
  - Keep up to T-1 runs per level L
  - Merge all runs of L_i into 1 run of L_{i+1}
    - L1
    - L2
    - L3



- **LSM Leveling**
  - Keep 1 run per level L
  - Merge run of Li with Li+1
    - L1
    - L2
    - L3



[Niv Dayan: Log-Structured-Merge Trees, **Comp115 guest lecture, 2017**]



[Stratos Idreos, Mark Callaghan: Key-Value Storage Engines (Tutorial), **SIGMOD 2020**]

Document Stores

Recap: JSON (JavaScript Object Notation)

- **JSON Data Model**
  - Data exchange format for **semi-structured data**
  - **Not as verbose as XML** (especially for arrays)
  - Popular format (e.g., Twitter)

```
{“students:”[
  {“id”: 1, “courses”:[
    {“id“:“INF.01017UF”, “name“:“DM”},
    {“id“:“706.550”, “name“:“AMLS”}]},
 {“id”: 5, “courses”:[
    {“id“:“706.520”, “name“:“DIA”}]},
]}
```

- **Query Languages**
  - **Most common: libraries** for tree traversal and data extraction
  - **JSONig:** XQuery-like query language
  - **JSONPath:** XPath-like query language

JSONiq Example:

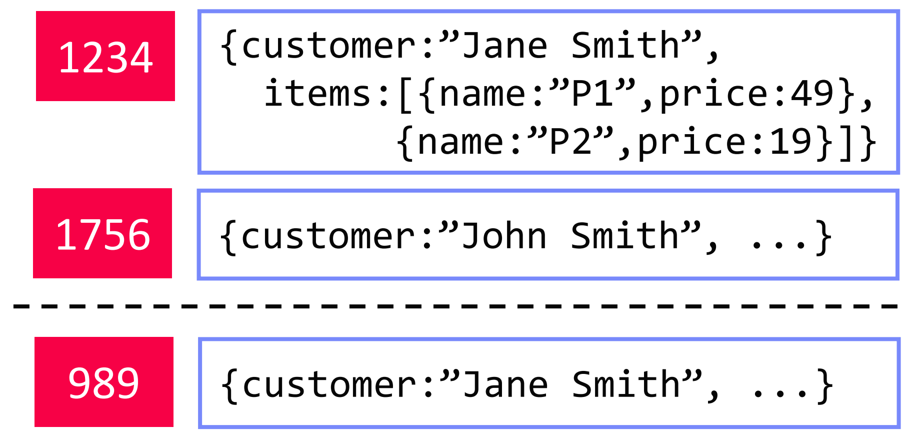
```
declare option jsoniq-version “…”;
for $x in collection(“students”)
  where $x.id lt 10
  let $c := count($x.courses)
  return {“sid”:$x.id, “count”:$c}
```

[http://www.jsoniq.org/docs/JSONiq/html-single/index.html]

Motivation and Terminology

- **Motivation**
  - Application-oriented management of **structured, semi-structured, and unstructured information** (pay-as-you-go, schema evolution)
  - Scalability via parallelization on commodity HW (cloud computing)

- **System Architecture**
  - Collections of (**key**, **document**)
  - Scalability via sharding (horizontal partitioning)
  - Custom SQL-like or functional query languages



- **Example Systems**
  - **MongoDB** (C++, 2007, **CP**) → **RethinkDB**, **Espresso**, **Amazon DocumentDB** (Jan 2019)
  - **CouchDB** (Erlang, 2005, **AP**) → **CouchBase**

Example MongoDB

[**Credit:** https://api.mongodb.com/python/current]

- **Creating a Collection**

```
import pymongo as m
conn = m.MongoClient("mongodb://localhost:123/")
db = conn["dbs19"]        # database dbs19
cust = db["customers"]    # collection customers
```

- **Inserting into a Collection**

```
mdict = {
  "name": "Jane Smith",
  "address": "Inffeldgasse 13, Graz"
}
id = cust.insert_one(mdict).inserted_id
# ids = cust.insert_many(mlist).inserted_ids
```

- **Querying a Collection**

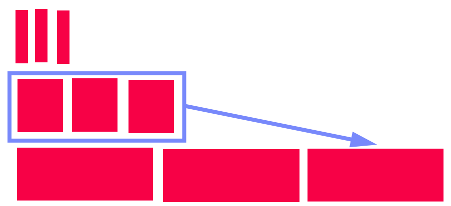
```
print(cust.find_one({"_id": id}))

ret = cust.find({"name": "Jane Smith"})
for x in ret:
  print(x)
```

BREAK (and Test Yourself)

- **NoSQL Systems** (10/100 points)
  - Describe the concept and system architecture of a **key-value store**, including techniques for achieving **high write throughput**, and **scale-out** in distributed environments. […]

- **Solution**
  - Key-value store
    system architecture [4]
  - Write-throughput via **LSM**
    (log-structured merge tree) [3]
  - **Horizontal partitioning** [3]
    (see **07 Physical Design**)



$$R1 = \sigma_{k<=5}(R)$$
$$R2 = \sigma_{k>5 \wedge k<=10}(R)$$
$$R3 = \sigma_{k>10 \wedge k<=15}(R)$$

$$R = (R1 \cup R2) \cup R3)$$

R

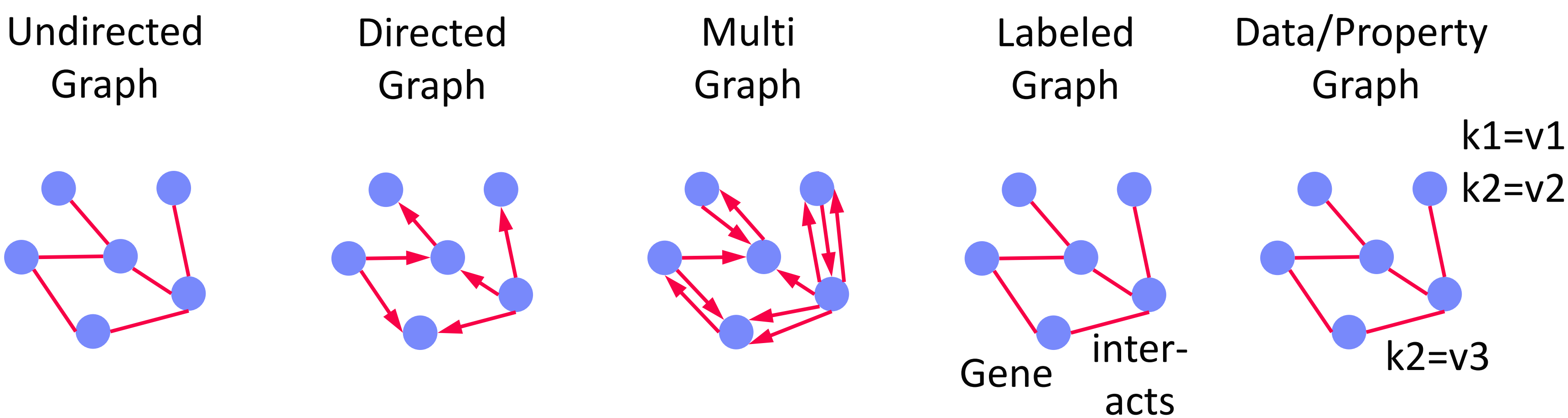
k	v
1	Blob1
2	Blob2
4	Blob4
7	Blob7
15	Blob15
9	Blob9
14	Blob14
8	Blob8

Graph Processing

Motivation and Terminology

- **Ubiquitous Graphs**
  - **Domains:** social networks, open/linked data, knowledge bases, bioinformatics
  - **Applications:** influencer analysis, ranking, topology analysis

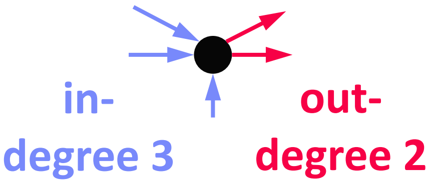
- **Terminology**
  - Graph G = (V, E) of vertices V (set of nodes) and edges E (set of links between nodes)
  - **Different types of graphs**



Terminology and Graph Characteristics

- **Terminology, cont.**
  - **Path:** Sequence of edges and vertices (**walk:** allows repeated edges/vertices)
  - **Cycle:** Closed walk, i.e., a walk that starts and ends at the same vertex
  - **Clique:** Subgraph of vertices where every two distinct vertices are adjacent

- **Metrics**
  - **Degree** (in/out-degree): number of incoming/outgoing edges of that vertex
  - **Diameter:** Maximum distance of pairs of vertices (longest shortest-path)

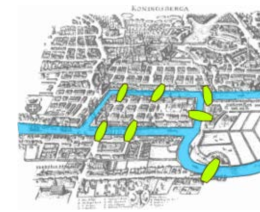


- **Power Law Distribution**
  - Degree of most real graphs follows a power law distribution

Vertex-Centric Processing

[Grzegorz Malewicz et al: Pregel: a system for large-scale graph processing. **SIGMOD 2010**
SIGMOD 2020 Test of Time Award]

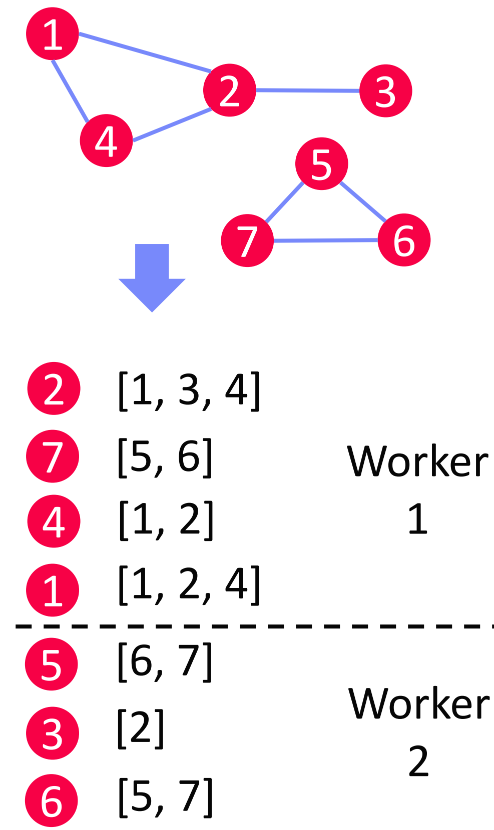
- **Google Pregel**
  - Name: Seven Bridges of Koenigsberg (Euler 1736)
  - **"Think-like-a-vertex"** computation model
  - Iterative processing in super steps, comm.: message passing



- **Programming Model**
  - Represent graph as collection of vertices w/ edge (adjacency) lists
  - Implement algorithms via Vertex API
  - Terminate if all vertices halted / no more msgs

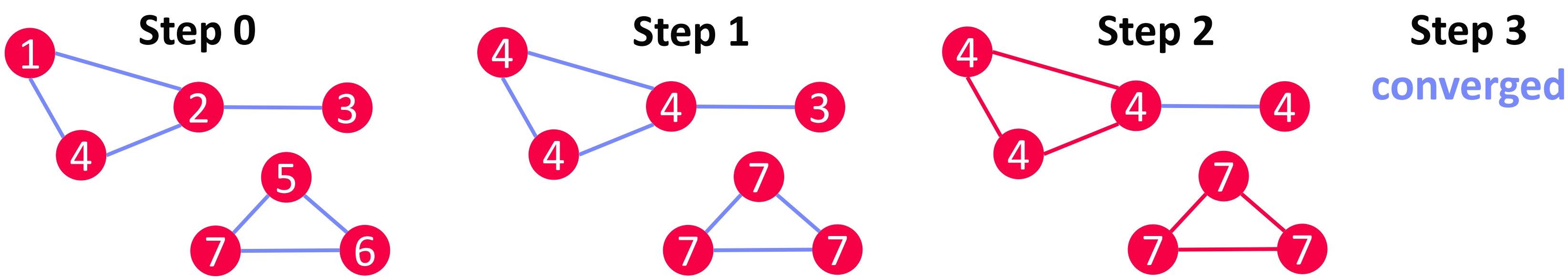
```
public abstract class Vertex {
  public String getID();
  public long superstep();
  public VertexValue getValue();

  public compute(Iterator<Message> msgs);
  public sendMsgTo(String v, Message msg);
  public void voteToHalt();
}
```

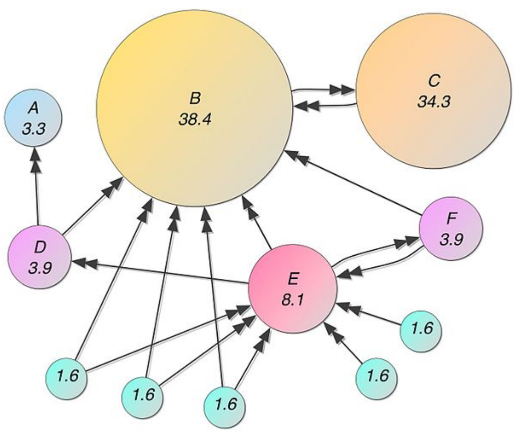


Vertex-Centric Processing, cont.

- **Example1: Connected Components**
  - Determine connected components of a graph (subgraphs of connected nodes)
  - Propagate max(current, msgs) if != current to neighbors, terminate if no msgs



- **Example 2: Page Rank**
  - Ranking of webpages by importance / impact
  - **#1:** Initialize vertices to 1/numVertices()
  - **#2:** In each super step
    - Compute current vertex value:
      `value = 0.15/numVertices()+0.85*sum(msg)`
    - Send to all neighbors:
      `value/numOutgoingEdges()`

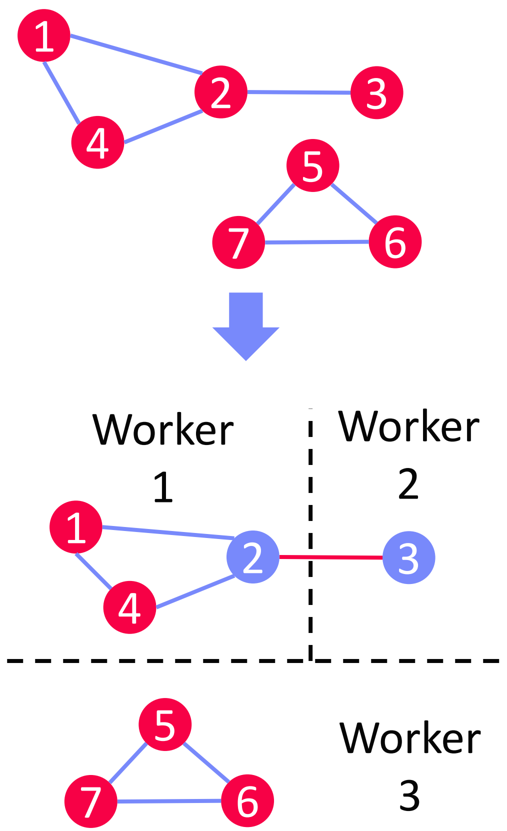


[**Credit:** https://en.wikipedia.org/wiki/PageRank]

Graph-Centric Processing

- **Motivation**
  - Exploit graph structure for algorithm-specific optimizations (number of network messages, scheduling overhead for super steps)
  - **Large diameter / average vertex degree**

- **Programming Model**
  - **Partition graph into subgraphs** (block/graph)
  - Implement algorithm directly against subgraphs (internal and boundary nodes)
  - Exchange messages in super steps only between boundary nodes ➔ **faster convergence**

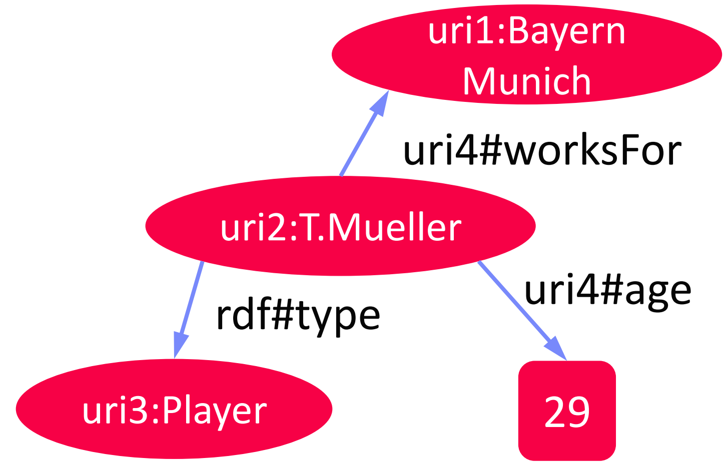


[Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, John McPherson: From "Think Like a Vertex" to "Think Like a Graph". **PVLDB 2013**]

[Da Yan, James Cheng, Yi Lu, Wilfred Ng: Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs. **PVLDB 2014**]

Resource Description Framework (RDF)

- **RDF Data**
  - Data and meta data description via triples
  - **Triple: (subject, predicate, object)**
  - Triple components can be URIs or literals
  - Formats: e.g., RDF/XML, RDF/JSON, Turtle
  - RDF graph is a directed, labeled multigraph



- **Querying RDF Data**
  - SPARQL (SPARQL Protocol And RDF Query Language)
  - Subgraph matching

```
SELECT ?person
WHERE {
  ?person rdf:type uri3:Player ;
          uri4:worksFor uri1:"Bayern Munich" .
}
```

- **Selected Example Systems**

Amazon Neptune, AllegroGraph, Sparksee, Apache Marmotta, Jena

Example Systems

- **Understanding Use in Practice**
  - Types of graphs user have
  - Graph computations run
  - Types of graph systems used

[Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, M. Tamer Özsu: The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. **PVLDB 2017**]

Technology	Software	# Users	
Graph Database System	ArangoDB [3]	40	233
	Caley [8]	14	
	DGraph [14]	33	
	JanusGraph [35]	32	
	Neo4j [48]	69	
	OrientDB [53]	45	
RDF Engine	Apache Jena [38]	87	115
	Sparksee [64]	5	
	Virtuoso [67]	23	
Distributed Graph Processing Engine	Apache Flink (Gelly) [17]	24	39
	Apache Giraph [21]	8	
	Apache Spark (GraphX) [27]	7	
Query Language	Gremlin [28]	82	82
Graph Library	Graph for Scala [22]	4	97
	GraphStream [24]	8	
	Graphtool [25]	28	
	NetworKit [50]	10	
	NetworkX [51]	27	
	SNAP [62]	20	
Graph Visualization	Cytoscape [13]	93	116
	Elasticsearch (X-Pack Graph) [16]	23	
Graph Representation	Conceptual Graphs [11]	6	6

Graph X

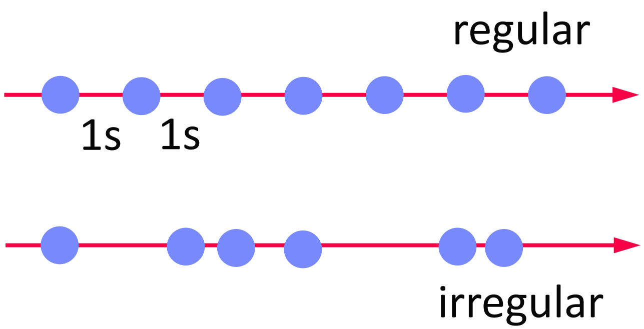
- **Summary of State of the Art Runtime Techniques**

[Da Yan, Yingyi Bu, Yuanyuan Tian, Amol Deshpande, James Cheng: Big Graph Analytics Systems. **SIGMOD 2016**]

Time Series Databases

Motivation and Terminology

- **Ubiquitous Time Series**
  - **Domains:** Internet-of-Things (IoT), sensor networks, smart production/planet, telemetry, stock trading, server/application metrics, event/log streams
  - **Applications:** monitoring, anomaly detection, time series forecasting
  - Dedicated storage and analysis techniques ➔ Specialized systems
- **Terminology**
  - Time series X is a sequence of data points x_i for a specific measurement identity (e.g., sensor) and time granularity
  - **Regular** (equidistant) time series (x_i) vs **irregular** time series (t_i, x_i)








Example InfluxDB

[Paul Dix: InfluxDB Storage Engine Internals, CMU Seminar, 09/2017]

- **Input Data**

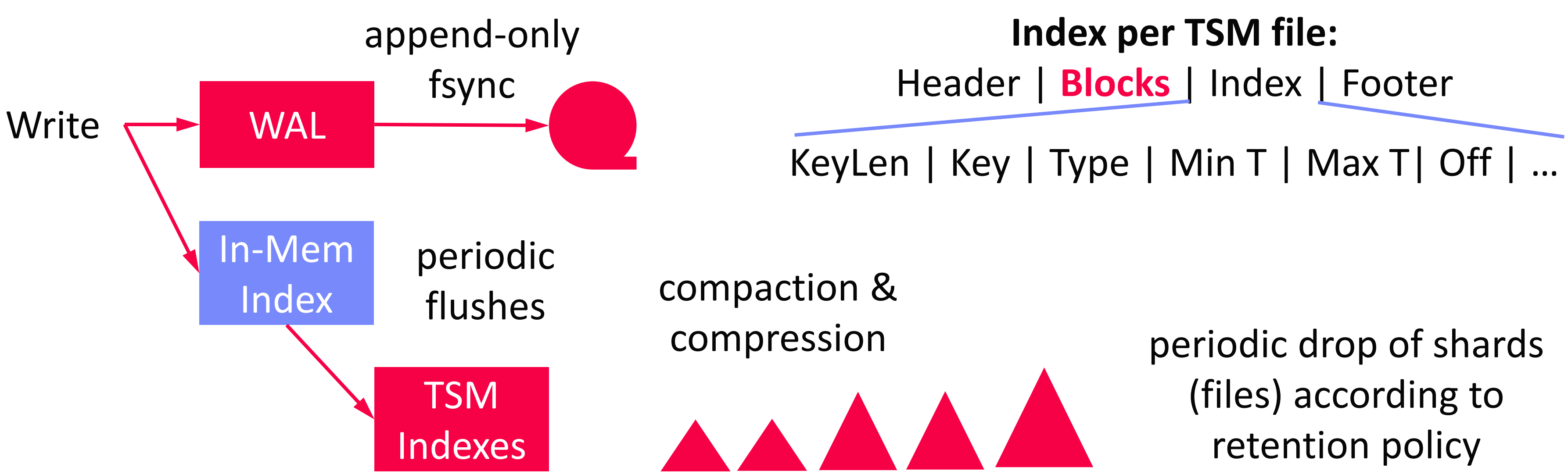
  Measurement

  `cpu,region=west,host=A` — Tags

  `user=85,sys=2,idle=10 1443782126`

  Fields (values) — Time

- **System Architecture**
  - Written in Go, originally **key-value store**, now **dedicated storage engine**
  - **Time Structured Merge Tree (TSM)**, similar to LSM
  - Organized in shards, TSM indexes and inverted index for reads



Example InfluxDB, cont.

- **Compression (of blocks)**
  - Compress up to 1000 values per block (Type | Len | Timestamps | Values)
  - **Timestamps:** Delta + Run-length encoding for regular time series; Simple8B or uncompressed for irregular
  - **Values:** double delta for FP64, bits for Bool, double delta + zig zag for INT64, Snappy for strings

- **Query Processing**
  - SQL-like and functional APIs for filtering (e.g., range) and aggregation
  - Inverted indexes

```
SELECT percentile(90, user)
  FROM cpu WHERE time>now()-12h
  AND “region”=‘west’
  GROUP BY time(10m), host
```

Measurement to fields:
cpu → [user,sys,idle]
host → [A, B]
Region → [west, east]

Posting lists:
cpu → [1,2,3,4,5,6]
host=A → [1,2,3]
host=B → [4,5,6]
region=west → [1,2,3]

Other Systems

- **Prometheus**
  - Metrics, high-dim data model, sharding and federation custom storage and query engine, implemented in Go
- **OpenTSDB**
  - TSDB on top of HBase or Google BigTable, Hadoop
- **TimescaleDB**
  - TSDB on top of PostgreSQL, standard SQL and reliability
- **Druid**
  - Column-oriented storage for time series, OLAP, and search
- **IBM Event Store**
  - HTAP system for high data ingest rates, and data-parallel analytics via Spark
  - Shard-local logs → groomed data

[Ronald Barber et al: Evolving Databases for New-Gen Big Data Applications. **CIDR 2017**]

[Christian Garcia-Arellano et al: **Db2 Event Store:** A Purpose-Built IoT Database Engine. **PVLDB 13(12) 2020**]

Exercise 4:
Large-Scale Data Analysis

Published: May 29

Deadline: June 22

Entire Exercise is Extra Credit

Task 4.1 Apache Spark Setup

3/25 points

- **#1 Pick your Spark Language Binding**
  - Java, Scala, Python
- **#2 Install Dependencies**
  - Java: Maven
    spark-core, spark-sql
  - Python:
    pip install pyspark

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-core_2.11</artifactId>
  <version>2.4.7</version>
 </dependency>
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.11</artifactId>
  <version>2.4.7</version>
</dependency>
```

- **(#3 Win Environment)**
  - Download https://github.com/steveloughran/winutils/tree/master/hadoop-2.7.1/bin/winutils.exe (or https://github.com/cdarlint/winutils)
  - Create environment variable HADOOP_HOME="<some-path>/hadoop"

Task 4.2 Query Processing via Spark RDDs

10/25 points

- **#1 Spark Context Creation**
  - Create a spark context sc w/ local master (`local[*]`)
- **#2 Implement Q07 via RDD Operations**
  - Implement Q07 in self-contained `executeQ07RDD()`
  - All reads should use `sc.textFile(fname)`
  - RDD operations only → stdout

https://spark.apache.org/docs/latest/rdd-programming-guide.html

- **Appendix A.1**

```
-- Determine the top-10 athletes that participated
-- between 1948 and 2016 in the most event occurrences
-- but never won a single medal.
SELECT A.AKey, A.Name, A.DoB, count(*)
  FROM Athletes A, Results R
  WHERE A.AKey = R.AKey
    AND A.AKey NOT IN( -- not in medal winners
      SELECT DISTINCT A2.AKey FROM Athletes A2, Results R2
      WHERE A2.AKey = R2.AKey AND R2.Medal IS NOT NULL)
    AND R.Year BETWEEN 1948 AND 2016
  GROUP BY A.AKey
  ORDER BY count(*) DESC, A.Name ASC
  LIMIT 10
```

Task 4.3 Query Processing via Spark SQL

5/25 points

- **#1 Spark Session Creation**
  - Create a spark session via a spark session builder and w/ local master (`local[*]`)

→ **SQL processing of high importance in modern data management**

- **#2 Implement Q07 via Dataset Operations**
  - Implement Q07 self-contained in `executeQ07Dataset()`
  - All reads should use `sc.read().format("csv")`
  - SQL or Dataset operations only → `out07.json`

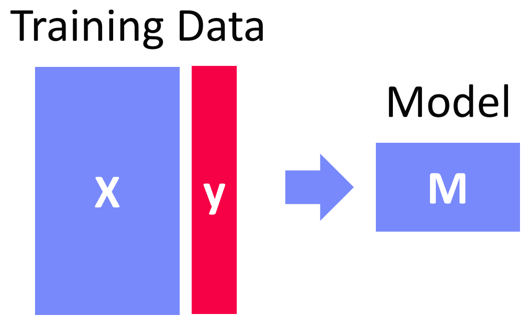
- **WebUI**

```
INFO Utils: Successfully started service 'SparkUI' on port 4040.
INFO SparkUI: Bound SparkUI to […] http://192.168.108.220:4040
```

Task 4.4: Medal Prediction

7/25 points

- **Regression Model** (or Classification Model)
  - Training data Summer Olympics 1896-2012
  - Test data Summer Olympics Rio 2016
  - Predict # meals won by individual athletes at a specific instance of the games
  - Data preparation and training pipeline in Spark



- **Example Data**

Country? One-Hot Encoding? (Year - DoB)? Other Features? y

Train: 146,839
Test: 11,179

akey integer	name character varying (128)	gender character (1)	dob date	height integer	weight integer	year integer	count bigint
88428	Michael Francisco Okantey	M	1940-...	175	66	1964	0
36781	Michael Frankenberg	M	1978-...	187	88	2004	0
94406	Michael Fred Phelps; II	M	1985-...	193	91	2000	0
94406	Michael Fred Phelps; II	M	1985-...	193	91	2004	8
94406	Michael Fred Phelps; II	M	1985-...	193	91	2008	8
94406	Michael Fred Phelps; II	M	1985-...	193	91	2012	6
81183	Michael Friedrich Mllenbeck	M	1970-...	200	120	1996	0

Conclusions and Q&A

- **Summary 10 NoSQL Systems**
  - Consistency and Data Models
  - Key-Value and Document Stores
  - Graph and Time Series Databases

- **Next Lectures** (Part B: Modern Data Management)
  - **11 Distributed Storage and Data Analysis** [Jun 07]
  - **12 Data stream processing systems**, **Q&A** [Jun 14]