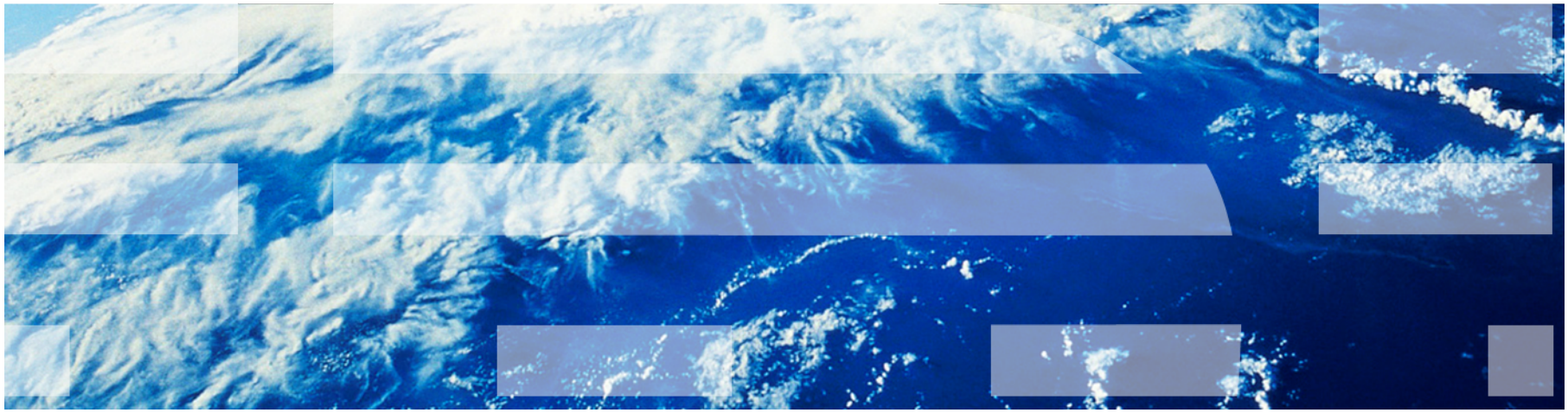
Columbia University

E6895 Advanced Big Data and AI Lecture 2:

Big Data Foundations

Ching-Yung Lin, Ph.D.

Adjunct Professor, Dept. of Electrical Engineering and Computer Science



January 27th, 2023

Milestone 1 Instruction

Milestone 1

- Each team will present on either February 3rd or February 10th
- Prepare about 10-12 mins of presentation (and expect 3-5 mins of Q&A)
- All teams will submit a written Milestone 1 report on February 10th.

- Key elements in Milestone 1 presentation:
  - Task Goal:
    - What do you want to achieve?
    - Why is the research and development important?
    - Is this a new topic that considers challenges of
      - Volume
      - Velocity
      - Variety
    - Does this topic try to incorporate multi-discipline knowledge?

Milestone 1

- Literature Survey:
  - What are the prior arts? What related works were done before?
  - Which research publications, tools and products may be utilized to build upon them to achieve the goal?

- Methodology:
  - What types of novel algorithms I shall try to invent and implement?
  - Where will you try to gather the data — Existing dataset? Self-collected dataset? Live dataset?

Milestone 1

- System:
  - What will your final system look like — potential backend components and interaction with front end?
  - How will you create visualization and user interface to help users consume your analysis outcome?

- Timeline:
  - What may you achieve in Milestones 2 and 3, and in the Final Project?

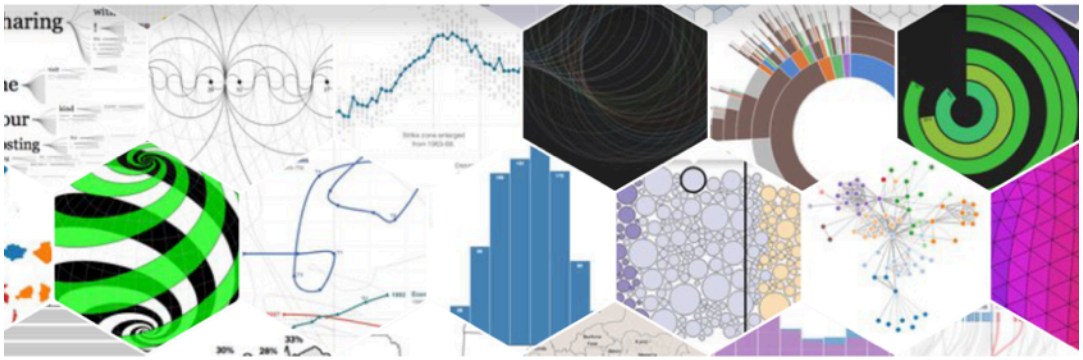
Note — it's good to think about what you want to achieve as complete as possible and study what other people have done. But, like all projects, everything can change when you make progress. Please do not afraid of making bold assumptions and attempt!!

Big Data Foundations

Key Open Source Big Data Foundations

Other Important Foundations (visualization and web servers)

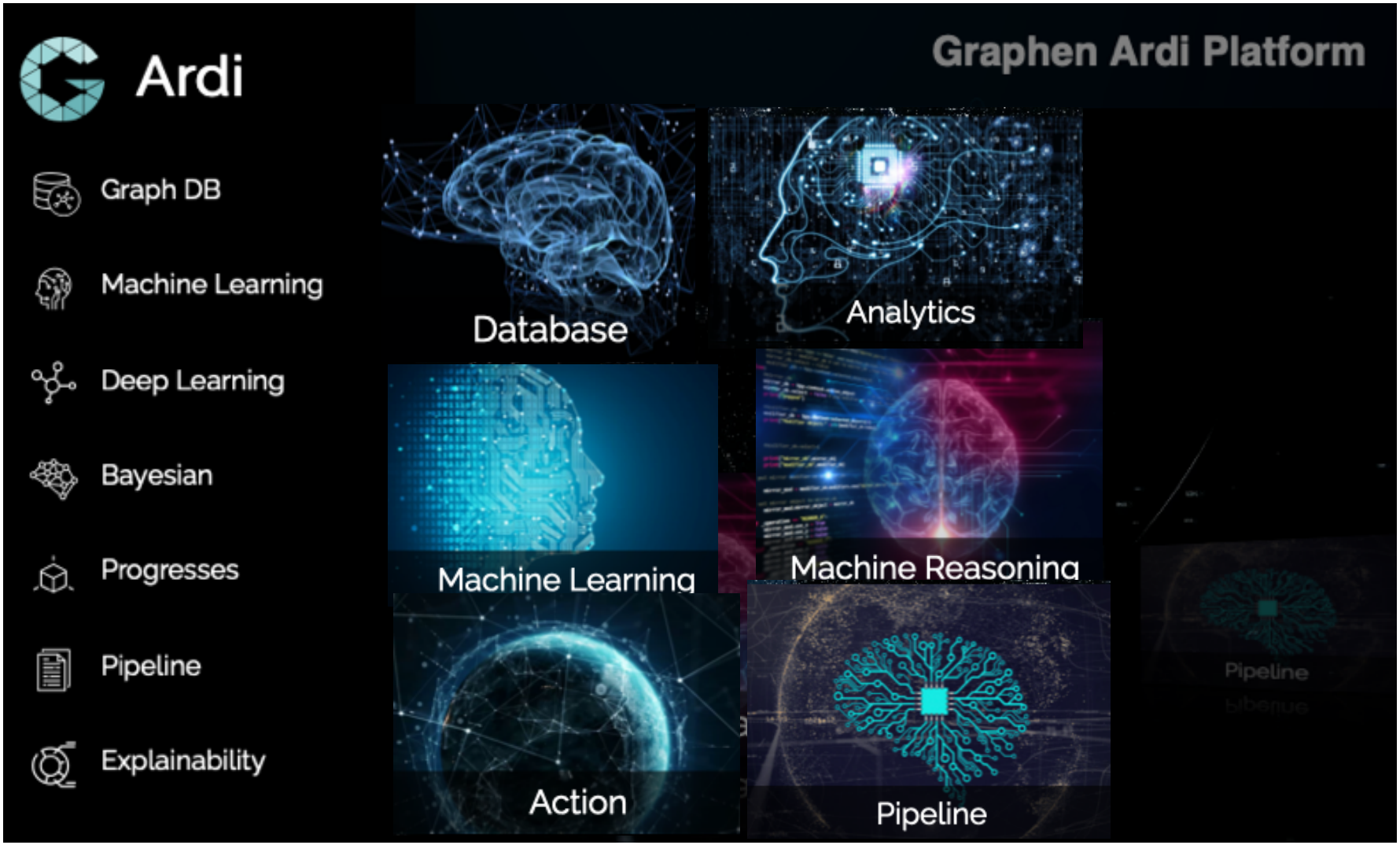
Data-Driven Documents



Flask
web development,
one drop at a time

APACHE
HTTP SERVER PROJECT

Reference Foundation (AI Platform)



123rf.com

Where to store data?

How to get data in and out?

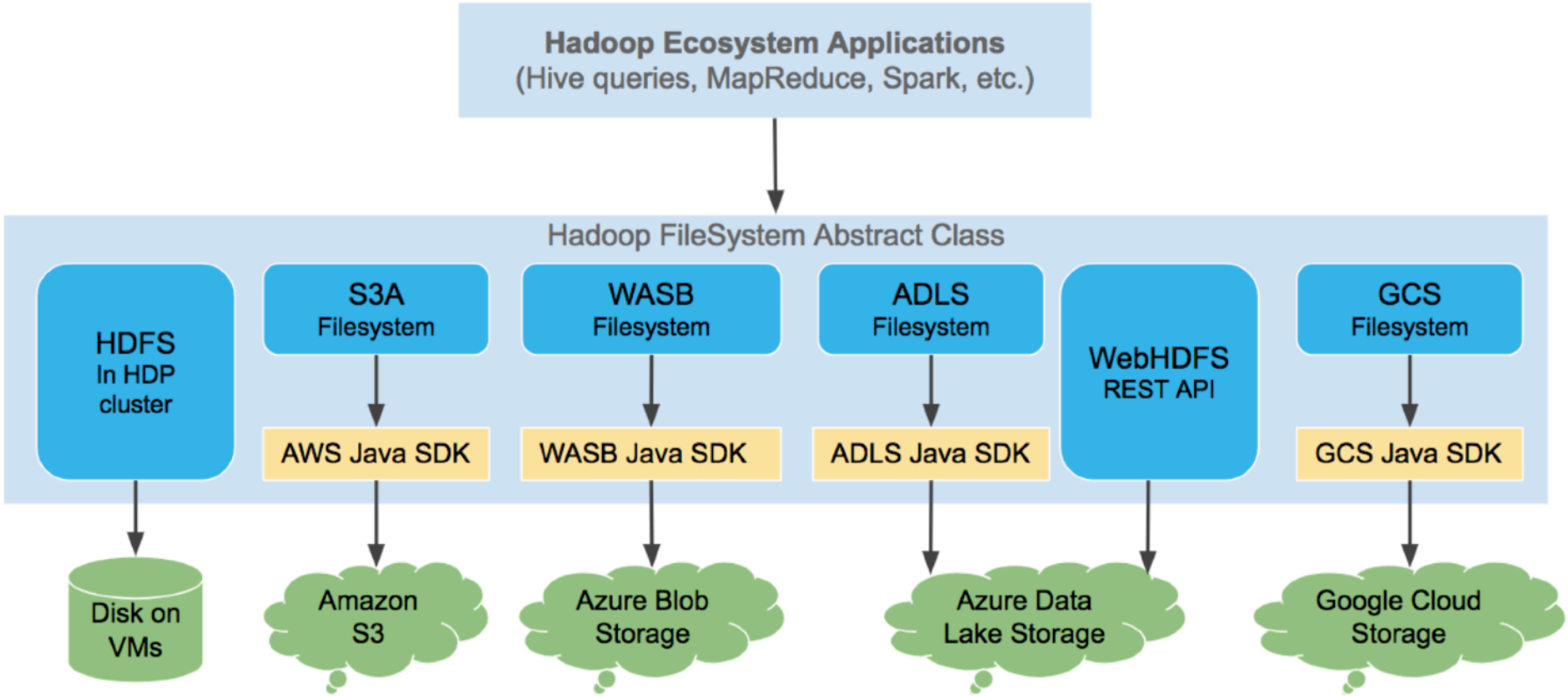
How to manage access of data?

Hadoop

Key Layers:

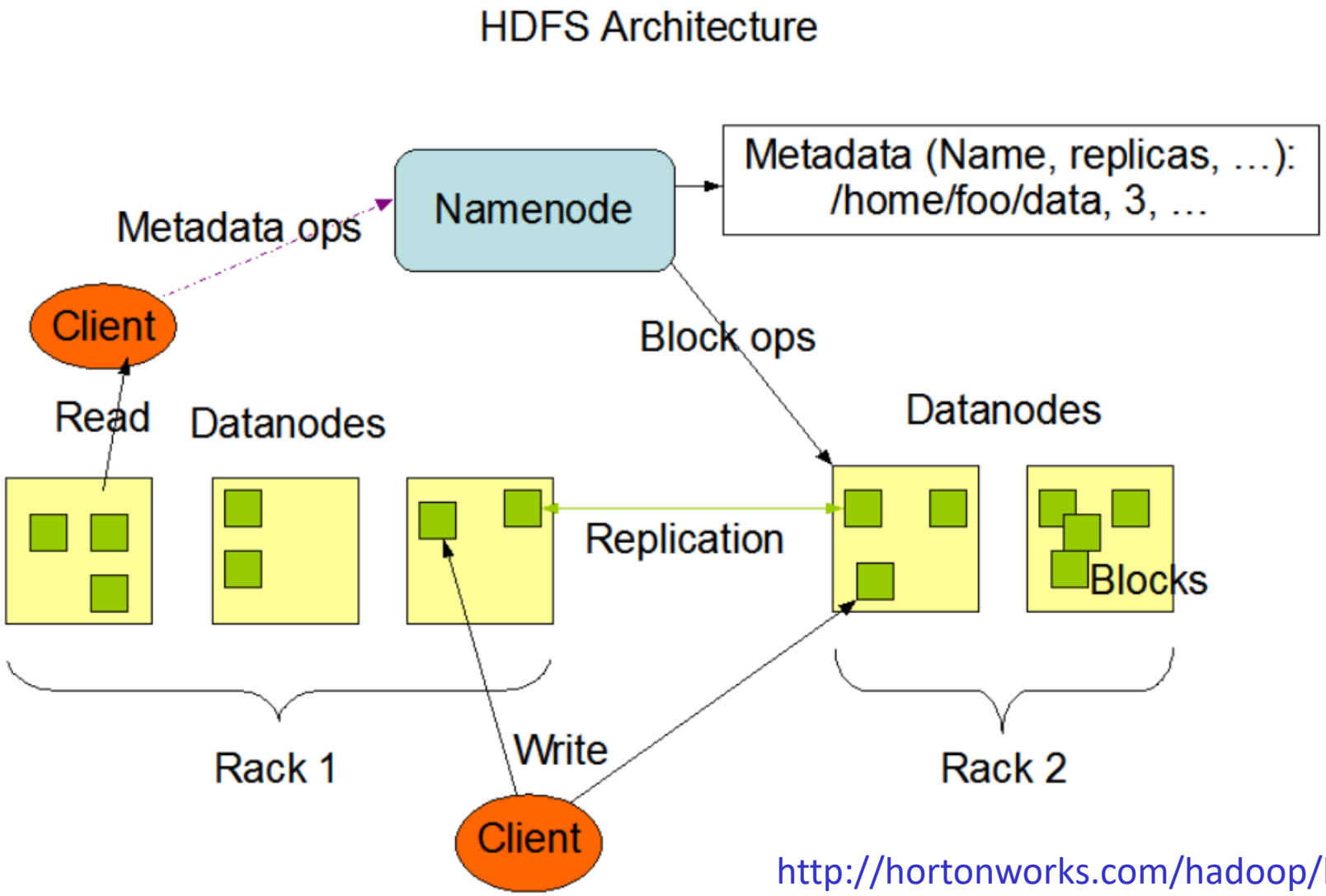
- Storage
- Compute Platform
- Shared Services
- Compute Frameworks

Hadoop Storage is widely used



By Tom McCuch, Cloudera, 8/20/2019

Hadoop Distributed File System (HDFS)



http://hortonworks.com/hadoop/hdfs/

Basic Data Storage Operations on HDFS

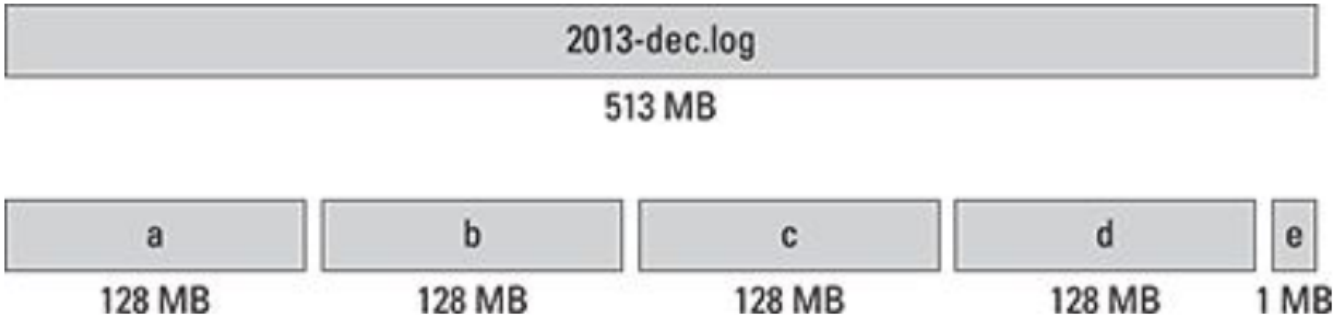
- Hadoop is designed to work best with a modest number of extremely large files.
- Average file sizes ➔ larger than 500MB.

- Write Once, Read Often model.
- Content of individual files cannot be modified, other than appending new data at the end of the file.

- What we can do:
  - Create a new file
  - Append content to the end of a file
  - Delete a file
  - Rename a file
  - Modify file attributes like owner

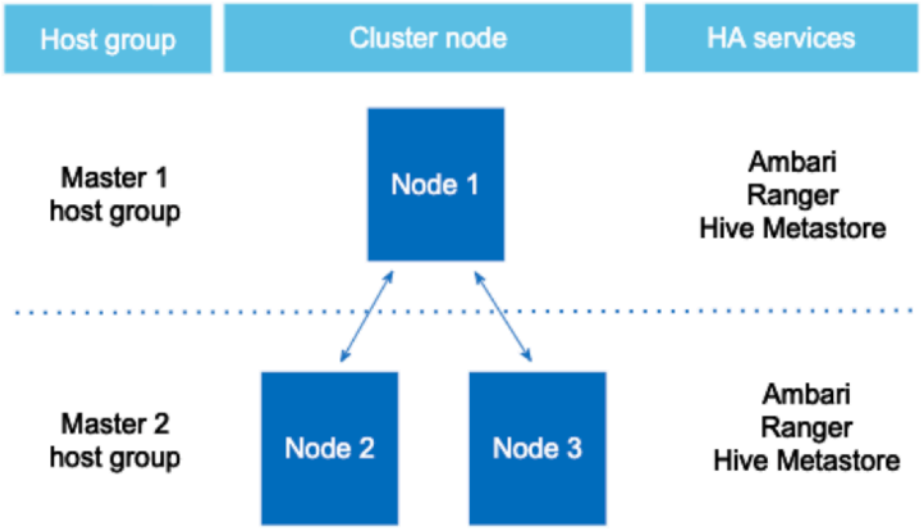
HDFS blocks

- File is divided into blocks (default: 64MB) and duplicated in multiple places (default: 3)



- Dividing into blocks is normal for a file system. E.g., the default block size in Linux is 4KB. The difference of HDFS is the scale.
- Hadoop was designed to operate at the petabyte scale.
- Every data block stored in HDFS has its own metadata and needs to be tracked by a central server.

Reliable System — High Availability



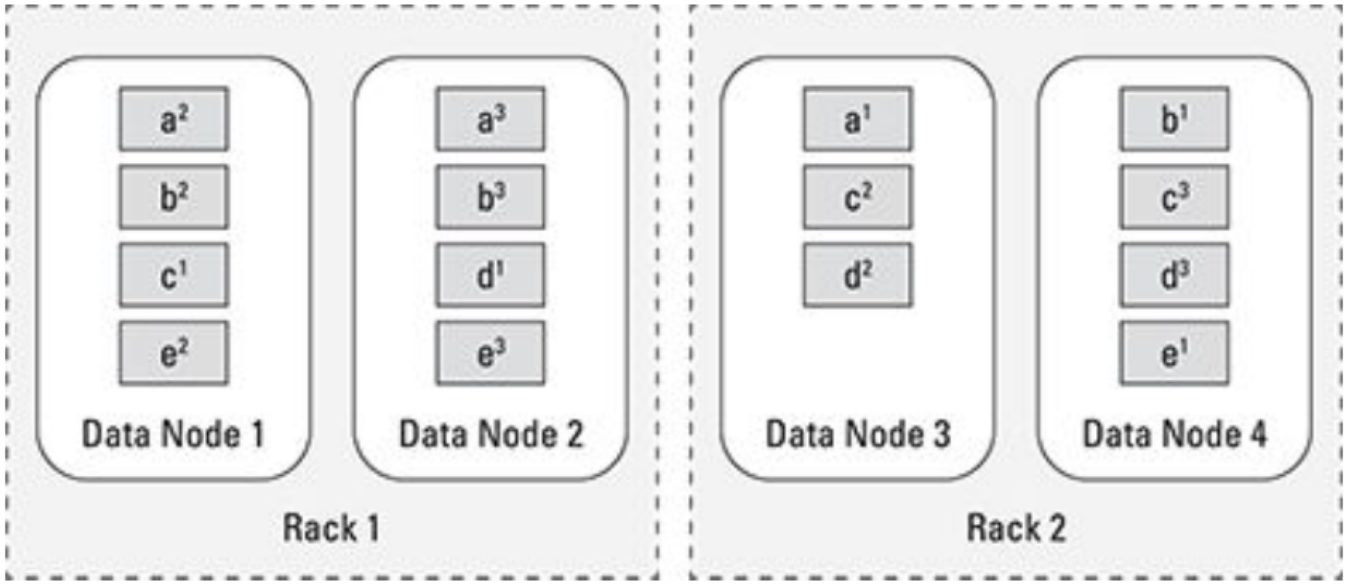
Availability %	Downtime per year[note 1]	Downtime per month	Downtime per week	Downtime per day
90% ("one nine")	36.53 days	73.05 hours	16.80 hours	2.40 hours
99% ("two nines")	3.65 days	7.31 hours	1.68 hours	14.40 minutes
99.9% ("three nines")	8.77 hours	43.83 minutes	10.08 minutes	1.44 minutes
99.99% ("four nines")	52.60 minutes	4.38 minutes	1.01 minutes	8.64 seconds

There are three principles of systems design in reliability engineering which can help achieve high availability.

1. Elimination of single points of failure. This means adding redundancy to the system so that failure of a component does not mean failure of the entire system.
2. Reliable crossover. In redundant systems, the crossover point itself tends to become a single point of failure. Reliable systems must provide for reliable crossover.
3. Detection of failures as they occur. If the two principles above are observed, then a user may never see a failure – but the maintenance activity must.

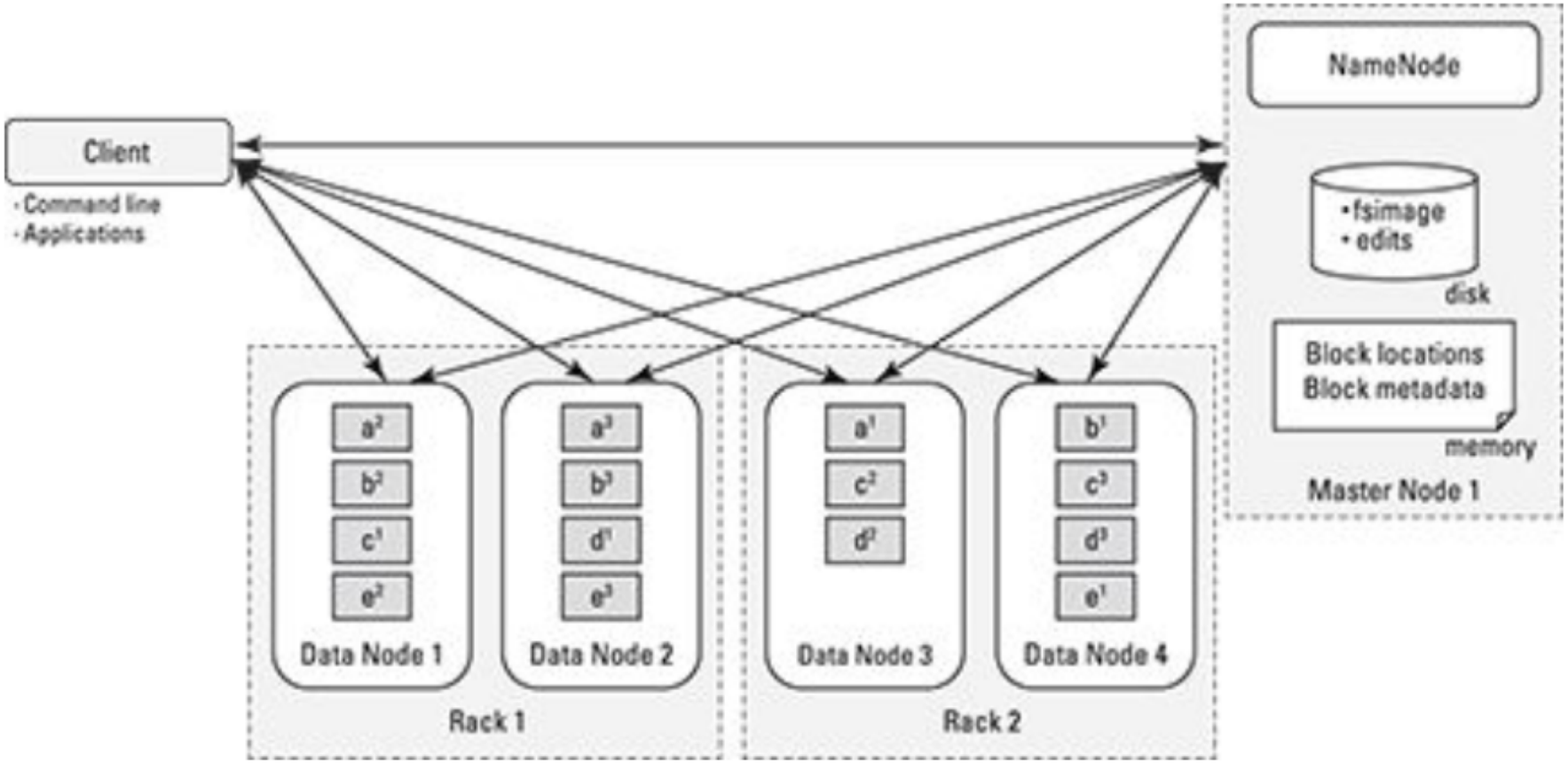
HDFS blocks

- Replication patterns of data blocks in HDFS.



- When HDFS stores the replicas of the original blocks across the Hadoop cluster, it tries to ensure that the block replicas are stored in different failure points.

Interaction between HDFS components



Several useful commands for HDFS

- All hadoop commands are invoked by the bin/hadoop script.

  hadoop [--config confdir] [COMMAND]
  [GENERIC_OPTIONS] [COMMAND_OPTIONS]

- % hadoop fsck / -files –blocks:
  ➔ list the blocks that make up each file in HDFS.

- For HDFS, the schema name is hdfs, and for the local file system, the schema name is file.
- A file or director in HDFS can be specified in a fully qualified way, such as:
  hdfs://namenodehost/parent/child or hdfs://namenodehost

- The HDFS file system shell command is similar to Linux file commands, with the following general syntax: ***hadoop hdfs –file_cmd***

- For instance mkdir runs as:
  $hadoop hdfs dfs –mkdir /user/directory_name

Several useful commands for HDFS -- II

For example, to create a directory named "joanna", run this mkdir command:

```
$ hadoop hdfs dfs -mkdir /user/joanna
```

Use the Hadoop put command to copy a file from your local file system to HDFS:

```
$ hadoop hdfs dfs -put file_name /user/login_user_name
```

For example, to copy a file named data.txt to this new directory, run the following put command:

```
$ hadoop hdfs dfs -put data.txt /user/joanna
```

Run the ls command to get an HDFS file listing:

```
$ hadoop hdfs dfs -ls .
```

Hive

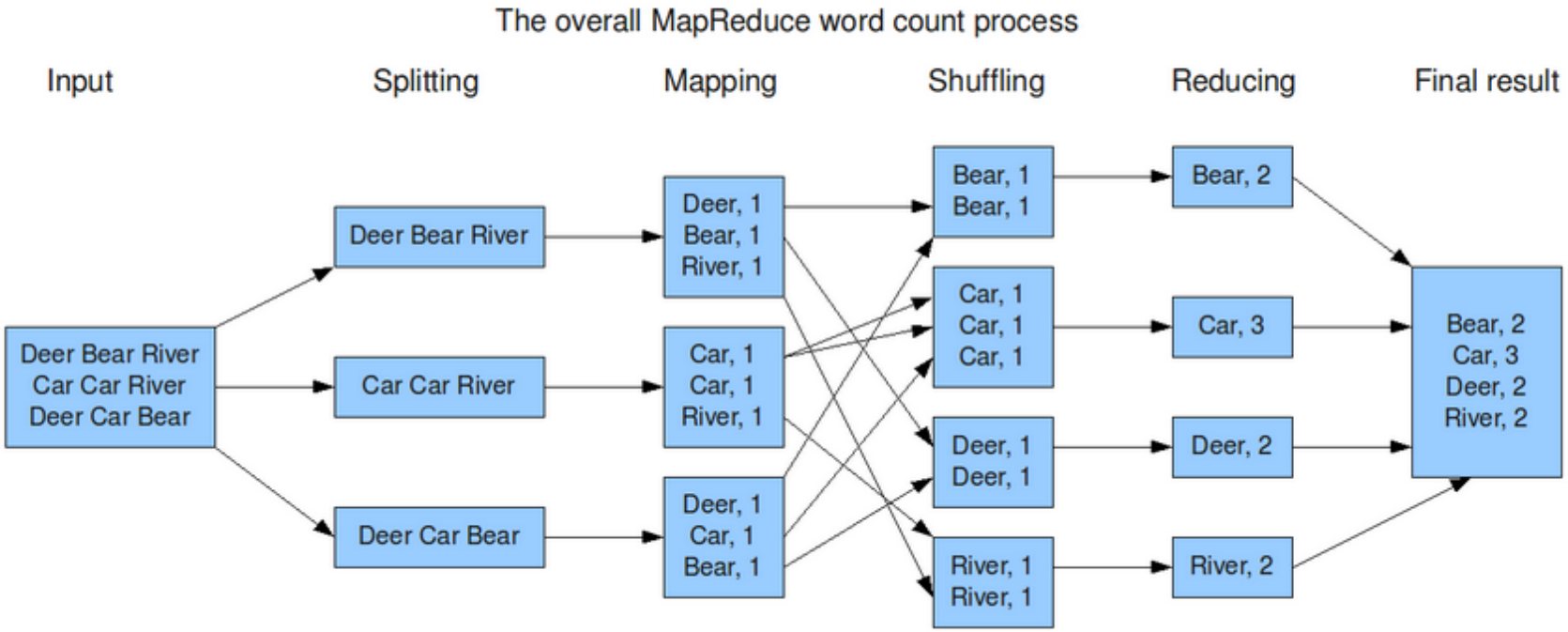
Apache Hive is a data warehouse software project built on top of Apache Hadoop for providing data query and analysis. Hive gives a SQL-like interface to query data stored in various databases and file systems that integrate with Hadoop.

HiveQL Example — Word Count:

```
1 DROP TABLE IF EXISTS docs;
2 CREATE TABLE docs (line STRING);
3 LOAD DATA INPATH 'input_file' OVERWRITE INTO TABLE docs;
4 CREATE TABLE word_counts AS
5 SELECT word, count(1) AS count FROM
6  (SELECT explode(split(line, '\s')) AS word FROM docs) temp
7 GROUP BY word
8 ORDER BY word;
```

Key Characteristics: (1) Schema on Read (vs Schema on Write of Traditional DB); (2) ACID — Atomicity, Consistency, Isolation, and Durability.

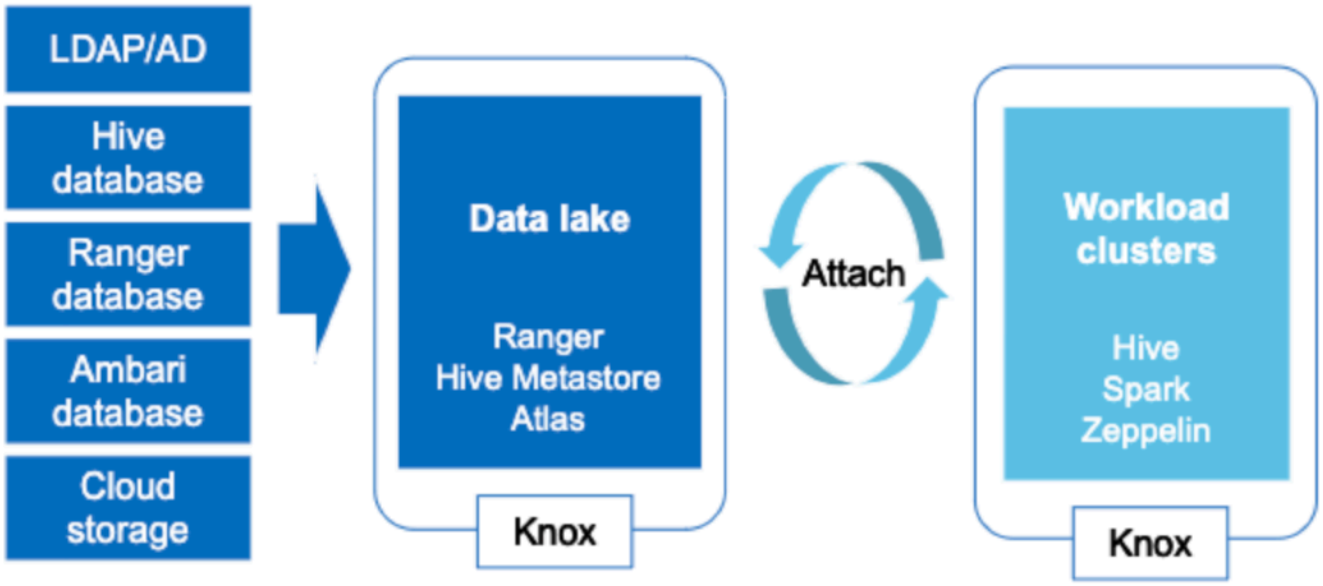
MapReduce example



http://www.alex-hanna.com

Data Lake

A data lake provides a way to centrally apply and enforce authentication, authorization, and audit policies across multiple ephemeral workload clusters.



"Working with Data Lakes", Cloudera

Term	Description
Data lake	Runs Ranger, which is used for configuring authorization policies and is used for audit capture. Runs Hive Metastore, which is used for data schema.
Workload clusters	The clusters that get attached to the data lake to run workloads. This is where you run workloads such as Hive via JDBC.

Data Lake — Components

Component	Technology	Description
Schema	Apache Hive	Provides Hive schema (tables, views, and so on). If you have two or more workloads accessing the same Hive data, you need to share schema across these workloads.
Policy	Apache Ranger	Defines security policies around Hive schema. If you have two or more users accessing the same data, you need security policies to be consistently available and enforced.
Audit	Apache Ranger	Audits user access and captures data access activity for the workloads.
Governance	Apache Atlas	Provides metadata management and governance capabilities.
Directory	LDAP/AD	Provides an authentication source for users and a definition of groups for authorization.
Gateway	Apache Knox	Supports a single workload endpoint that can be protected with SSL and enabled for authentication to access to resources.

Apache Ambari:

- Provision a Hadoop Cluster
- Manage a Hadoop Cluster
- Monitor a Hadoop Cluster

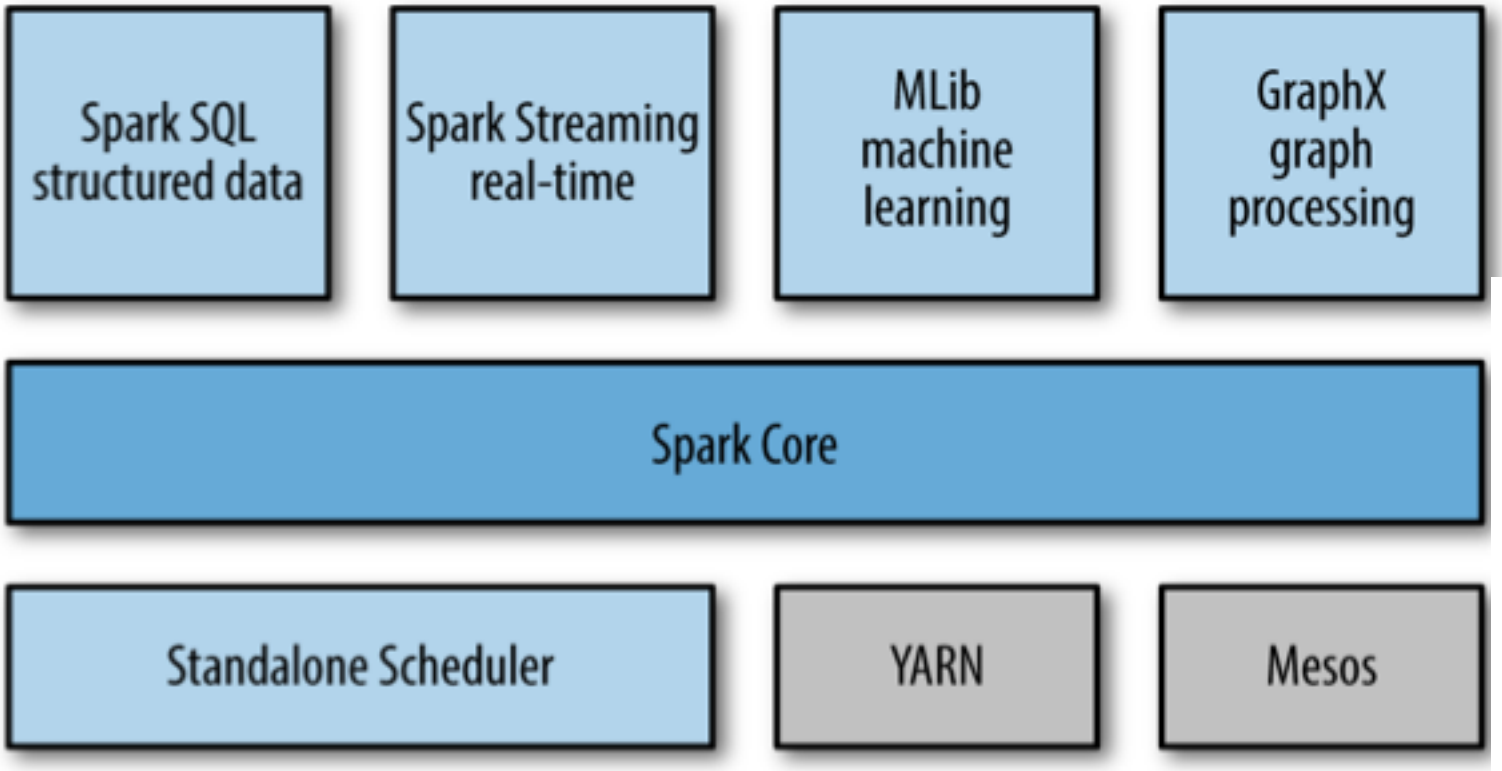
Apache Zeppelin:

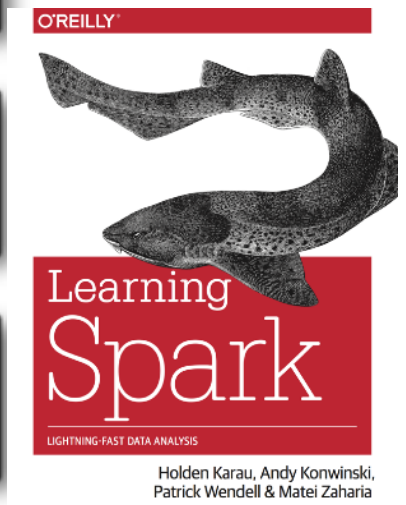
- Web-based notebook for data analytics and collaborative documents

How do I process the data?
How do I execute machine learning from the data?
How do I tell people my analytics results?

Spark

Lightning-fast unified analytics engine





Spark MLlib

Includes:

- ML Algorithms: common learning algorithms such as classification, regression, clustering, and collaborative filtering
- Featurization: feature extraction, transformation, dimensionality reduction, and selection
- Pipelines: tools for constructing, evaluating, and tuning ML Pipelines
- Persistence: saving and load algorithms, models, and Pipelines
- Utilities: linear algebra, statistics, data handling, etc.

MLlib: Main Guide

- Basic statistics
- Data sources
- Pipelines
- Extracting, transforming and selecting features
- Classification and Regression
- Clustering
- Collaborative filtering
- Frequent Pattern Mining
- Model selection and tuning
- Advanced topics

Spark MLlib Basic Statistics

Includes:

- Correlation
- Hypothesis testing
- Summarizer

Example of Calculating Correlation of Time Sequences:

```
from pyspark.ml.linalg import Vectors
from pyspark.ml.stat import Correlation

data = [(Vectors.sparse(4, [(0, 1.0), (3, -2.0)]),),
        (Vectors.dense([4.0, 5.0, 0.0, 3.0]),),
        (Vectors.dense([6.0, 7.0, 0.0, 8.0]),),
        (Vectors.sparse(4, [(0, 9.0), (3, 1.0)]),)]
df = spark.createDataFrame(data, ["features"])

r1 = Correlation.corr(df, "features").head()
print("Pearson correlation matrix:\n" + str(r1[0]))

r2 = Correlation.corr(df, "features", "spearman").head()
print("Spearman correlation matrix:\n" + str(r2[0]))
```

MLlib: Main Guide

- Basic statistics
- Data sources
- Pipelines
- Extracting, transforming and selecting features
- Classification and Regression
- Clustering
- Collaborative filtering
- Frequent Pattern Mining
- Model selection and tuning
- Advanced topics

Spark MLlib Features

Includes:

- Extraction: Extracting features from "raw" data
- Transformation: Scaling, converting, or modifying features
- Selection: Selecting a subset from a larger set of features
- Locality Sensitive Hashing (LSH): This class of algorithms combines aspects of feature transformation with other algorithms.

- Feature Extractors
  - TF-IDF
  - Word2Vec
  - CountVectorizer
  - FeatureHasher

- Feature Selectors
  - VectorSlicer
  - RFormula
  - ChiSqSelector

- Feature Transformers
  - Tokenizer
  - StopWordsRemover
  - n-gram
  - Binarizer
  - PCA
  - PolynomialExpansion
  - Discrete Cosine Transform (DCT)
  - StringIndexer
  - IndexToString
  - OneHotEncoder (Deprecated since 2.3.0)
  - OneHotEncoderEstimator
  - VectorIndexer
  - Interaction
  - Normalizer
  - StandardScaler
  - MinMaxScaler
  - MaxAbsScaler
  - Bucketizer
  - ElementwiseProduct
  - SQLTransformer
  - VectorAssembler
  - VectorSizeHint
  - QuantileDiscretizer
  - Imputer

Spark MLlib Supervised Machine Learning Algorithms

Classification
- Logistic regression
  - Binomial logistic regression
  - Multinomial logistic regression
- Decision tree classifier
- Random forest classifier
- Gradient-boosted tree classifier
- Multilayer perceptron classifier
- Linear Support Vector Machine
- One-vs-Rest classifier (a.k.a. One-vs-All)
- Naive Bayes

Regression
- Linear regression
- Generalized linear regression
  - Available families
- Decision tree regression
- Random forest regression
- Gradient-boosted tree regression
- Survival regression
- Isotonic regression

Spark MLlib UnSupervised Machine Learning & Recommendation Algorithms

Clustering:

- K-means
  - Input Columns
  - Output Columns
- Latent Dirichlet allocation (LDA)
- Bisecting k-means
- Gaussian Mixture Model (GMM)
  - Input Columns
  - Output Columns

Collaborative Filtering:

- Explicit vs. implicit feedback
- Scaling of the regularization parameter
- Cold-start strategy

Spark MLlib Model Selection and Tuning

- Model selection (a.k.a. hyperparameter tuning)
- Cross-Validation
- Train-Validation Split

Kafka

Kafka is a distributed streaming platform that is used publish and subscribe to streams of records.

Kafka is used for fault tolerant storage.

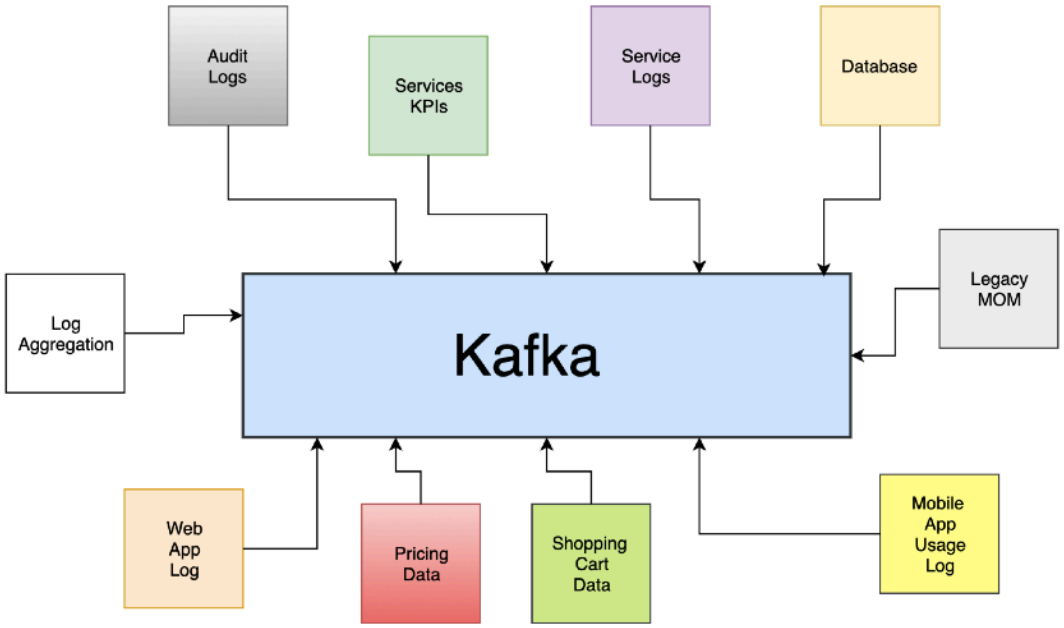
Kafka replicates topic log partitions to multiple servers.

Kafka is designed to allow your apps to process records as they occur.

Kafka is fast and uses IO efficiently by batching and compressing records.

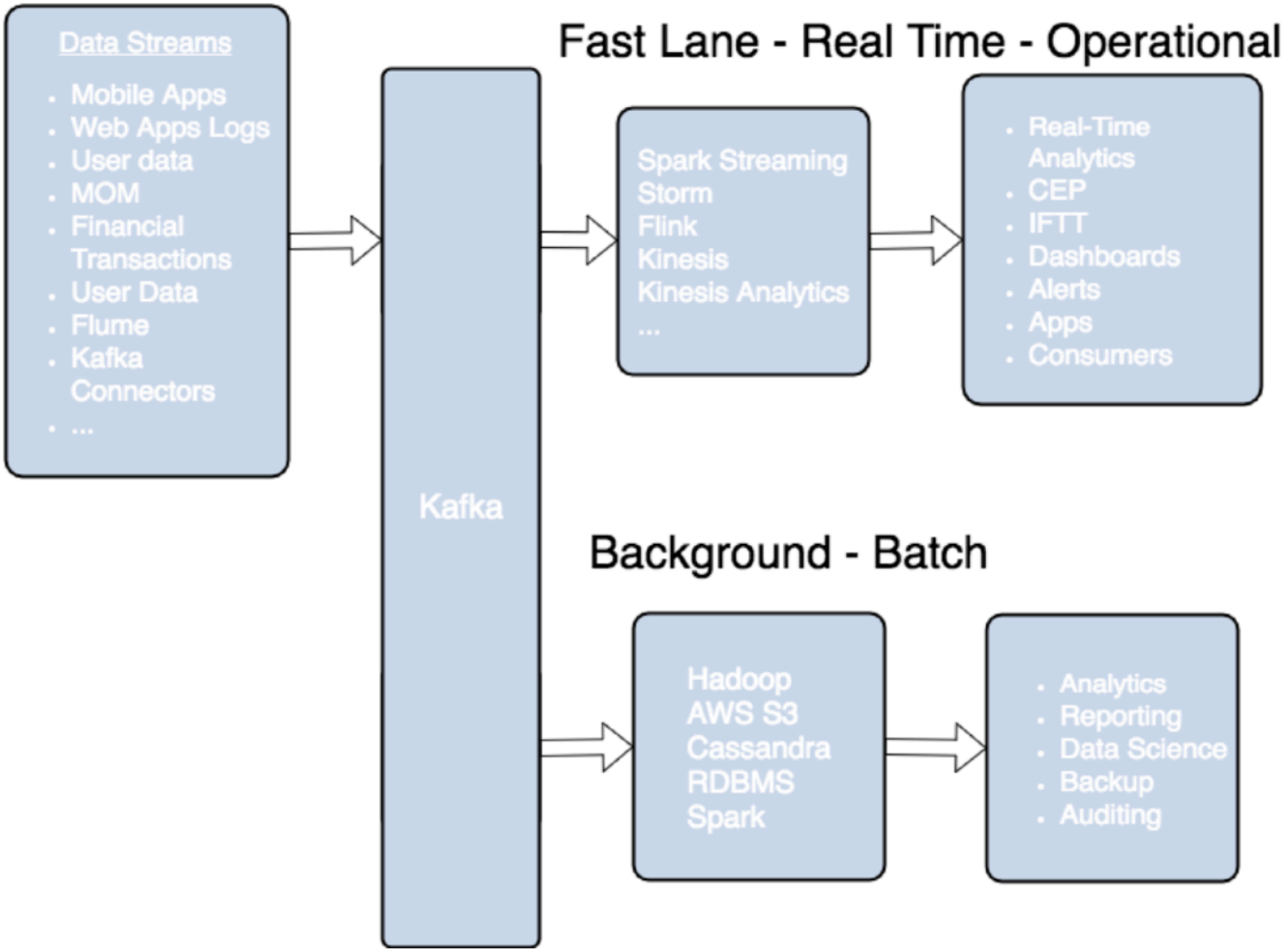
Kafka is used for decoupling data streams.

Kafka is used to stream data into data lakes, applications, and real-time stream analytics systems.



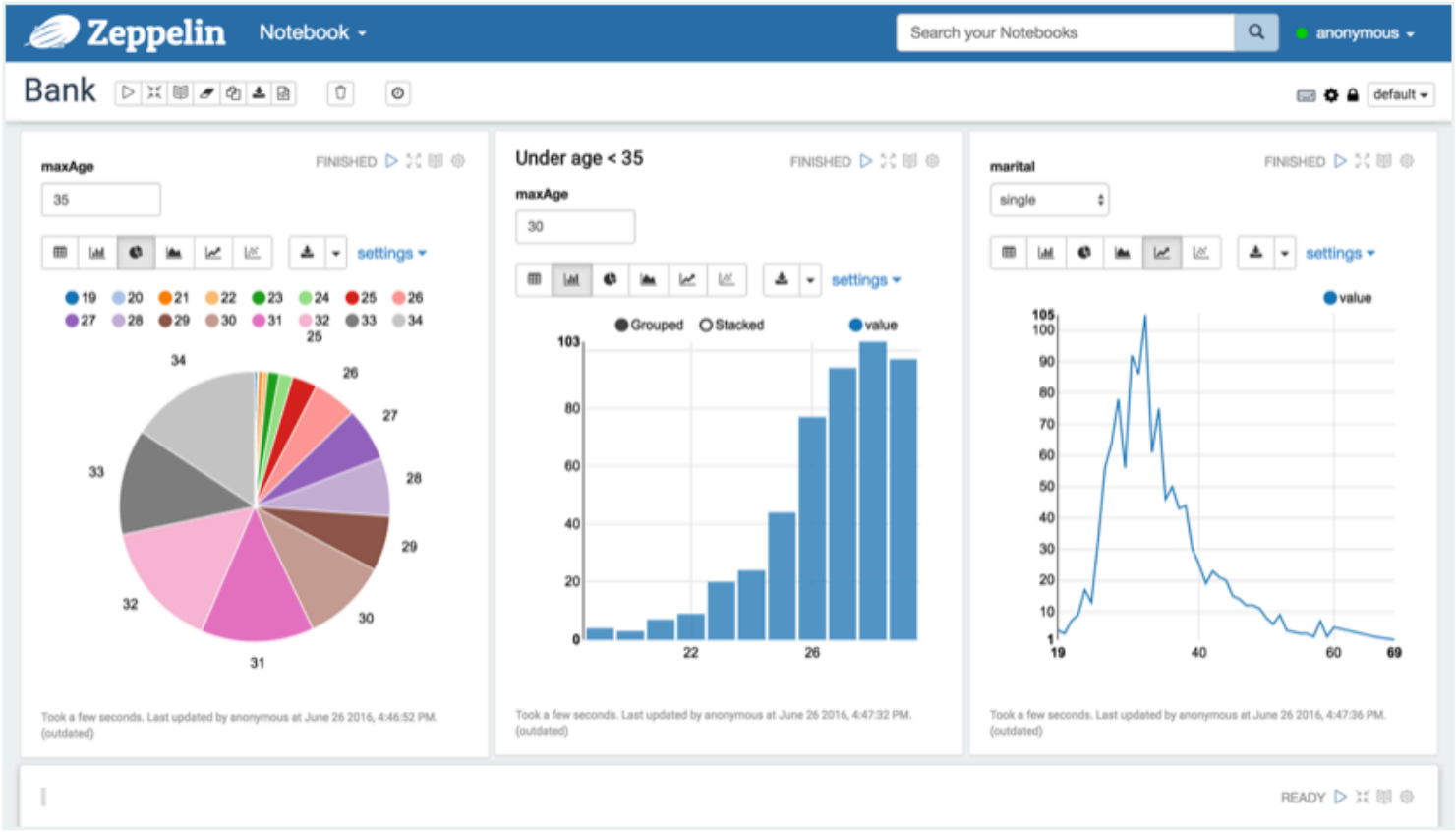
Kafka decoupling data streams

Where to use Kafka



Kafka streaming architecture diagram

Zeppelin (early stage)



Data Ingestion; Data Discovery; Data Analytics; Data Visualization & Collaboration

Webservers

Apache (http server) — the oldest and most popular web server exists in every linux machine, including MacOS machines.

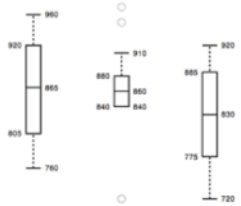
— display webpages of those files reside in its http root directory


```
from flask import Flask, escape, request

app = Flask(__name__)

@app.route('/')
def hello():
    name = request.args.get("name", "World")
    return f'Hello, {escape(name)}!'
```

D3 Visualization (via Javascript) Examples

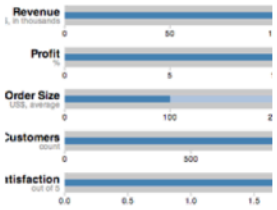
Visual Index

Box Plots

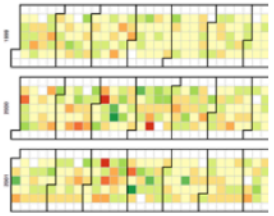


Bubble Chart

Bullet Charts



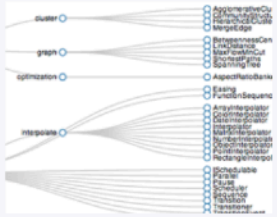
Calendar View



Non-contiguous Cartogram

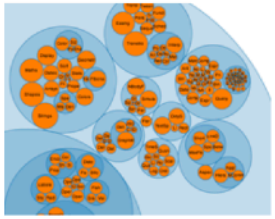
Chord Diagram

Dendrogram

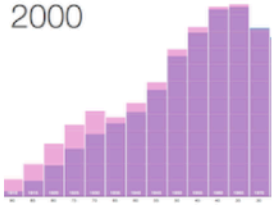


Force-Directed Graph

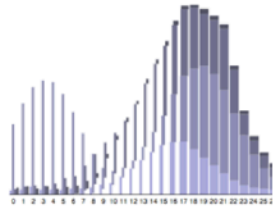
Circle Packing



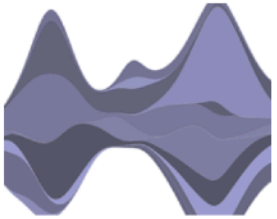
Population Pyramid



Stacked Bars

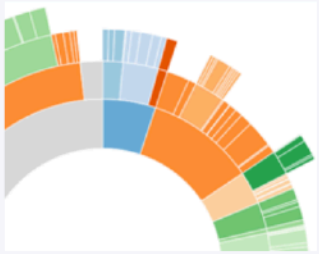


Streamgraph



D3 Visualization (via Javascript) Examples

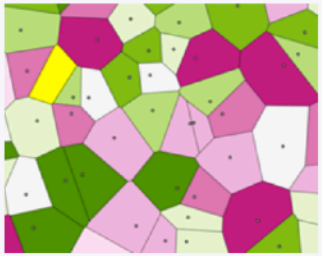
Sunburst



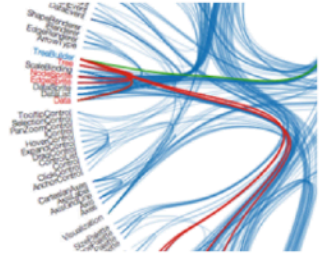
Node-Link Tree

Treemap

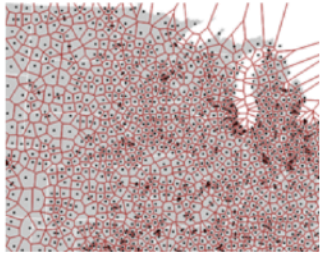
Voronoi Diagram



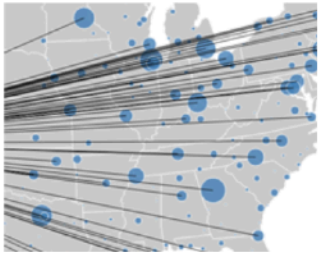
Hierarchical Edge Bundling



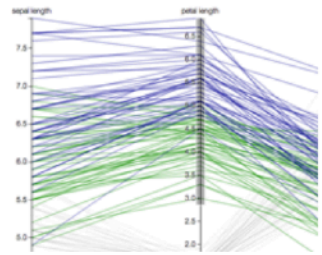
Voronoi Diagram



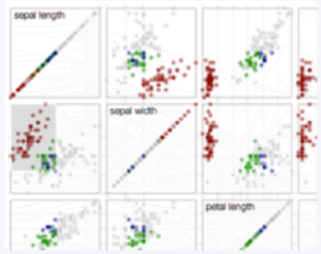
Bubble Map



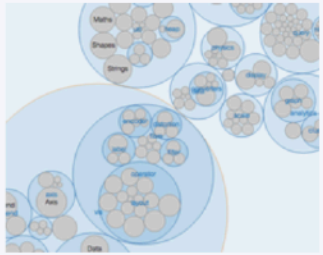
Parallel Coordinates



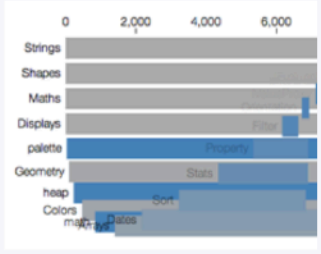
Scatterplot Matrix



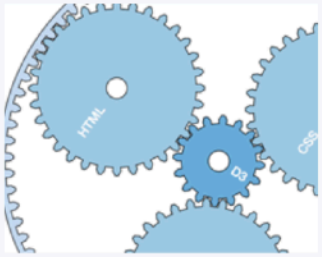
Zoomable Pack Layout



Hierarchical Bars



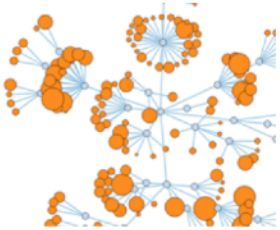
Epicyclical Gears



D3 Visualization (via Javascript) Examples

Collision Detection

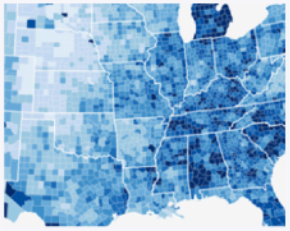
Collapsible Force Layout



Force-Directed States

Versor Dragging

Choropleth

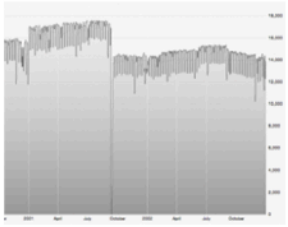


Collapsible Tree Layout

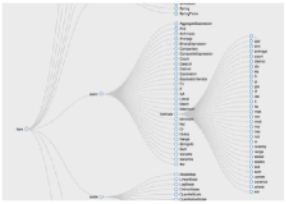
Zoomable Treemap

Zoomable Icicle

Zoomable Area Chart

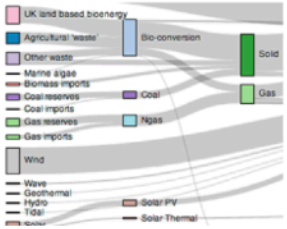


Drag and Drop Collapsible Tree Layout

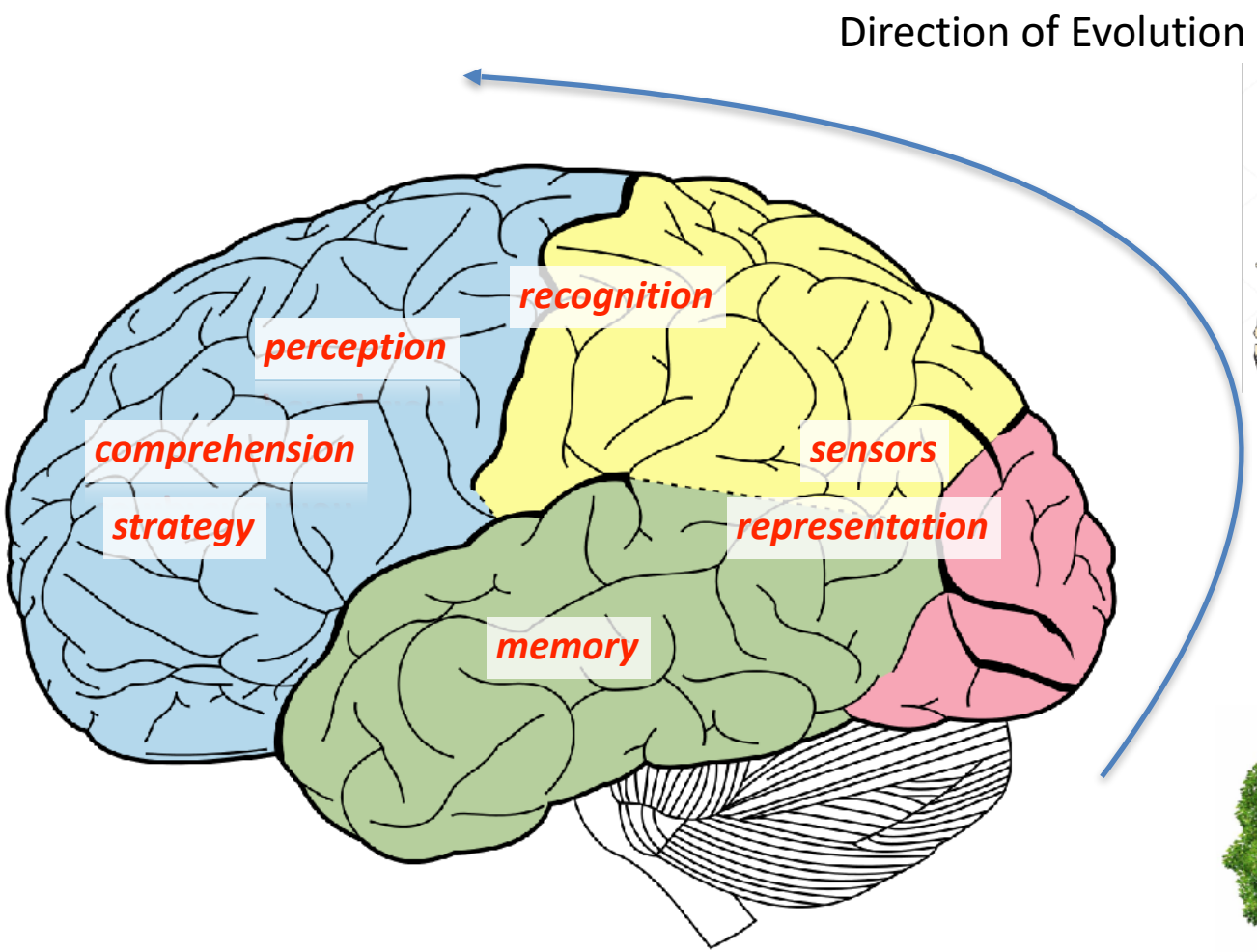


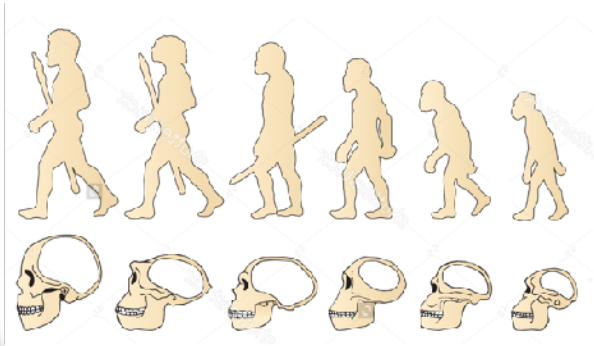
Radial Cluster Layout

Sankey Diagram

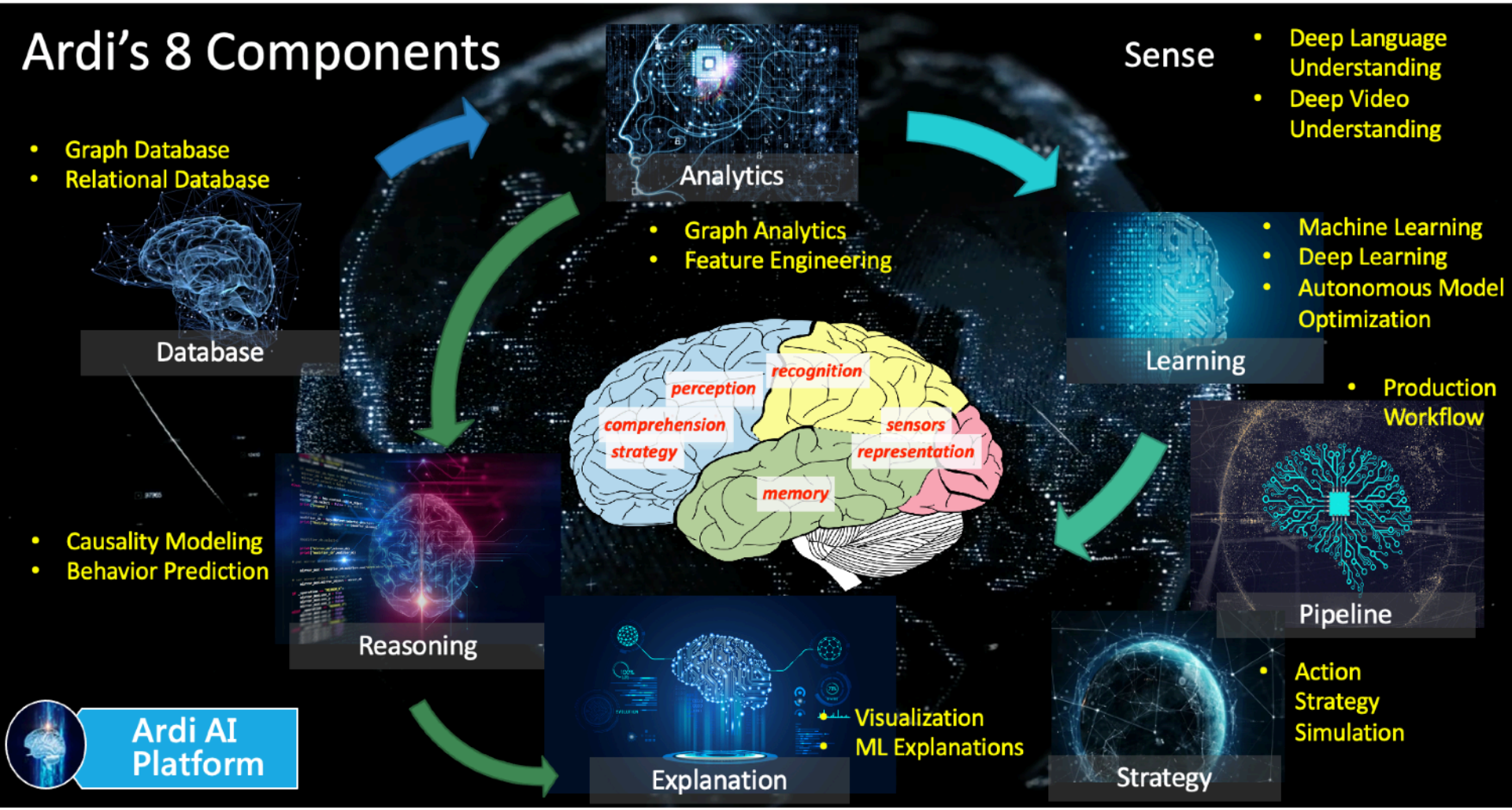


Human brain is a graph of 100B nodes and 700T edges.





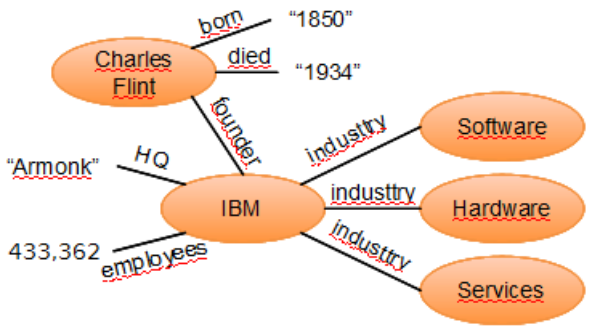
AI Platform



Graph computing emerged as THE foundation to solve complex AI issues

RDF / Property Graph

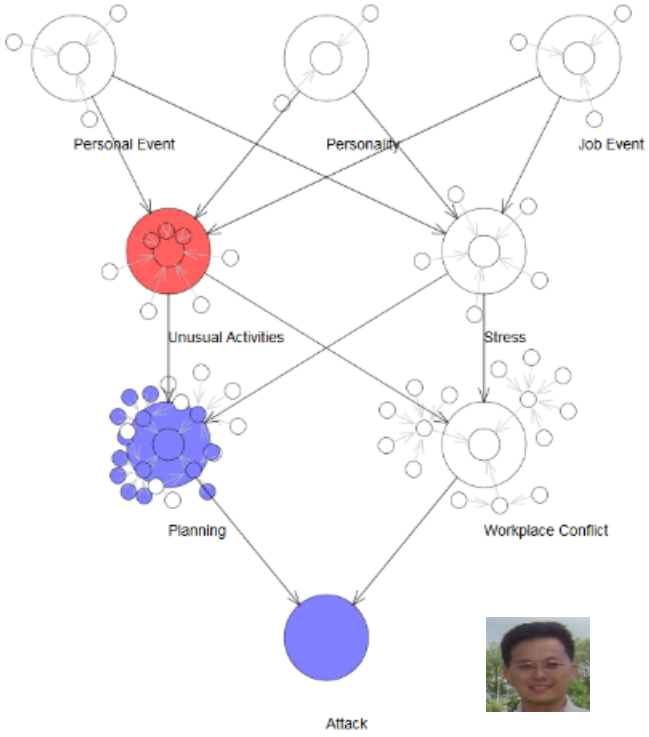
Graph Database



subject	predicate	object
Charles Flint	born	"1850"
Charles Flint	died	"1934"
Charles Flint	founder	IBM
IBM	HQ	"Armonk"
IBM	employees	433,362
IBM	industry	Software
IBM	industry	Hardware
IBM	industry	Services

Activity Graph

Behavior Analysis

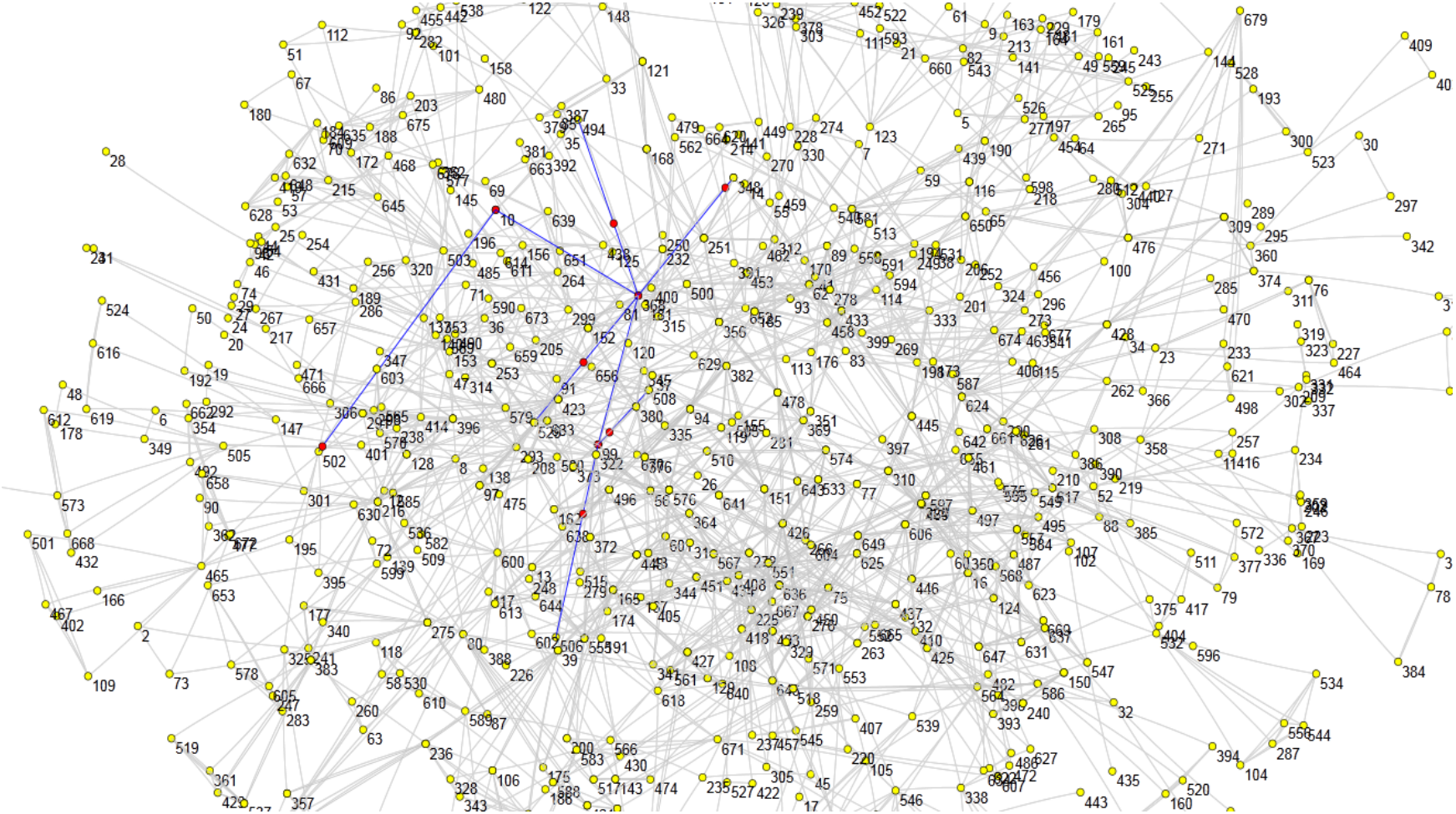


Collective Graph

Relationship Analysis

Improvement on accuracy: 2x-3x by context / relation analysis, 10x+ by reasoning / behavior prediction

Information / Transaction Flows



Intelligence and Knowledge Graph

How do kids grow intelligence?

"Airplane"

"Grandma"

"Grandma is in Taiwan"

"Auntie is also in Taiwan"

"I like grandma"

"I like grandpa"

The boy said:

"I like grandma and grandpa"

Why is Graph Computing the core of AI?

How does a Graph Computing mechanism reason and store data?

"I"

"see"

"Airplane"

"visit"

"Grandma"

"in"

"Taiwan"

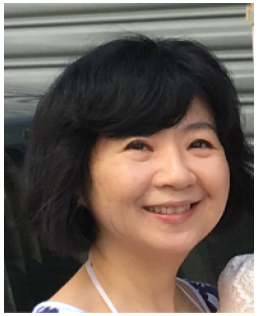
"like"

"and"

"in"

"like"

"Grandpa"

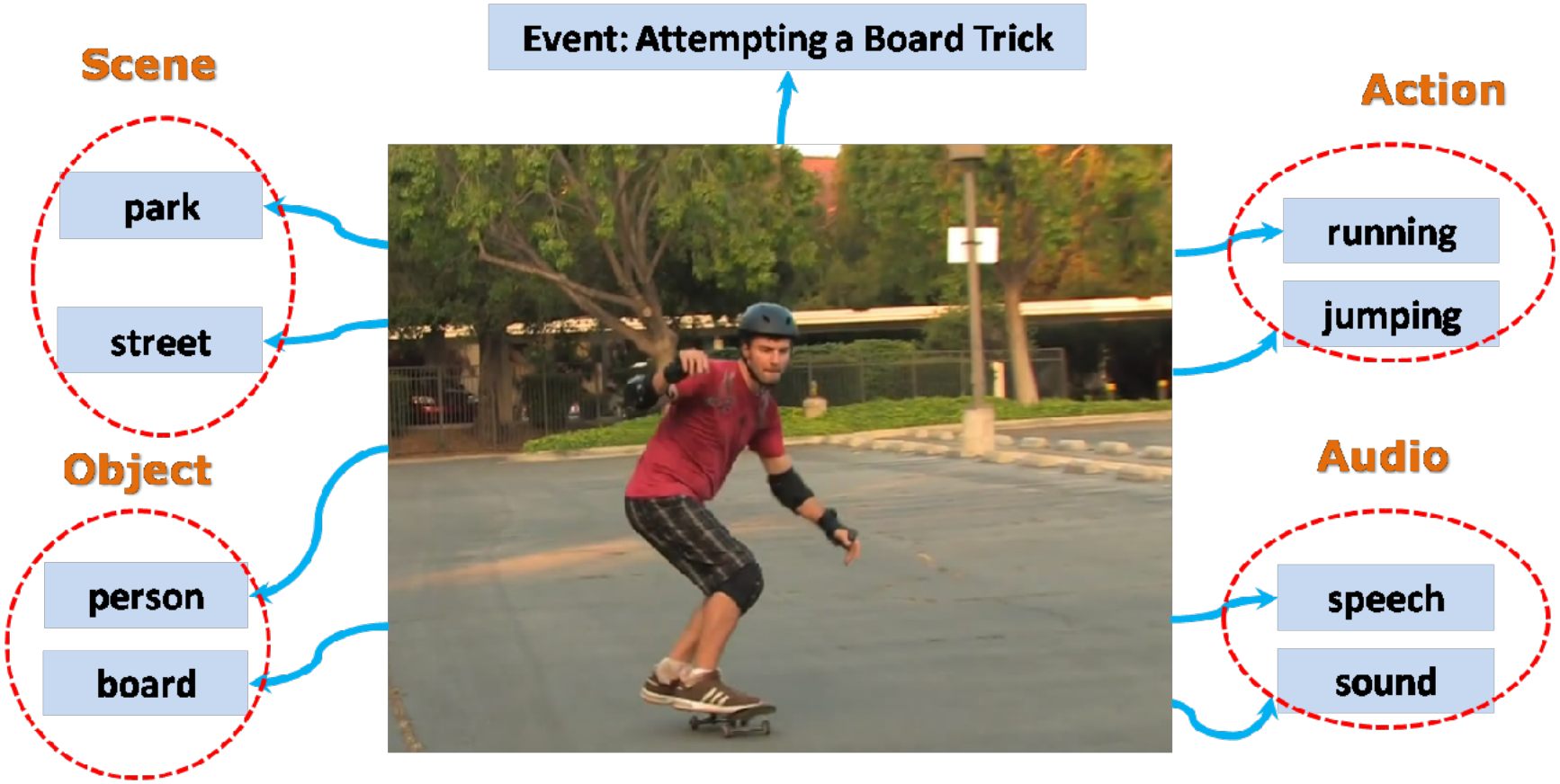


"Auntie"

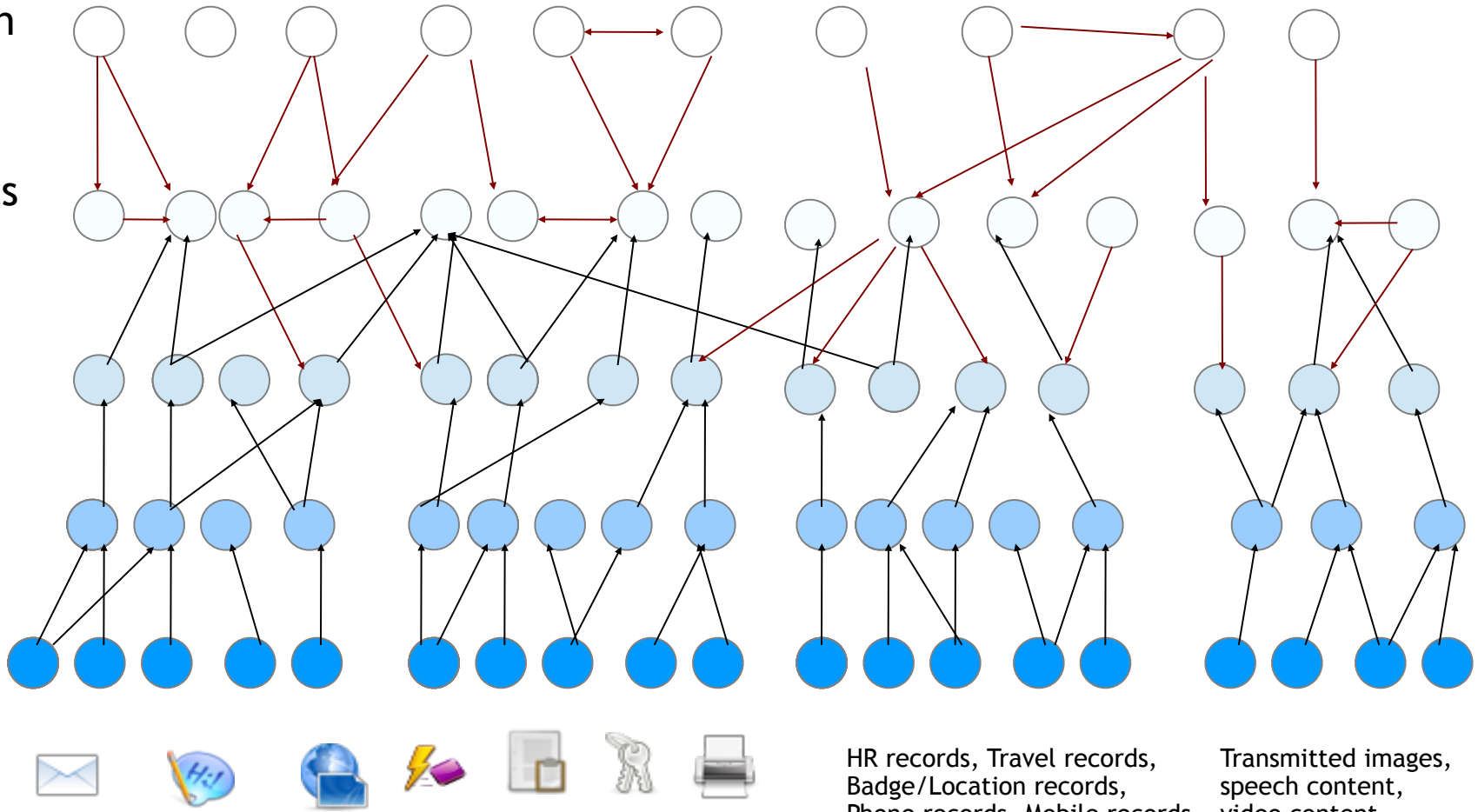
Graph platform works like the human mind, connecting the dots when comprehending.

Cognitive Recognition

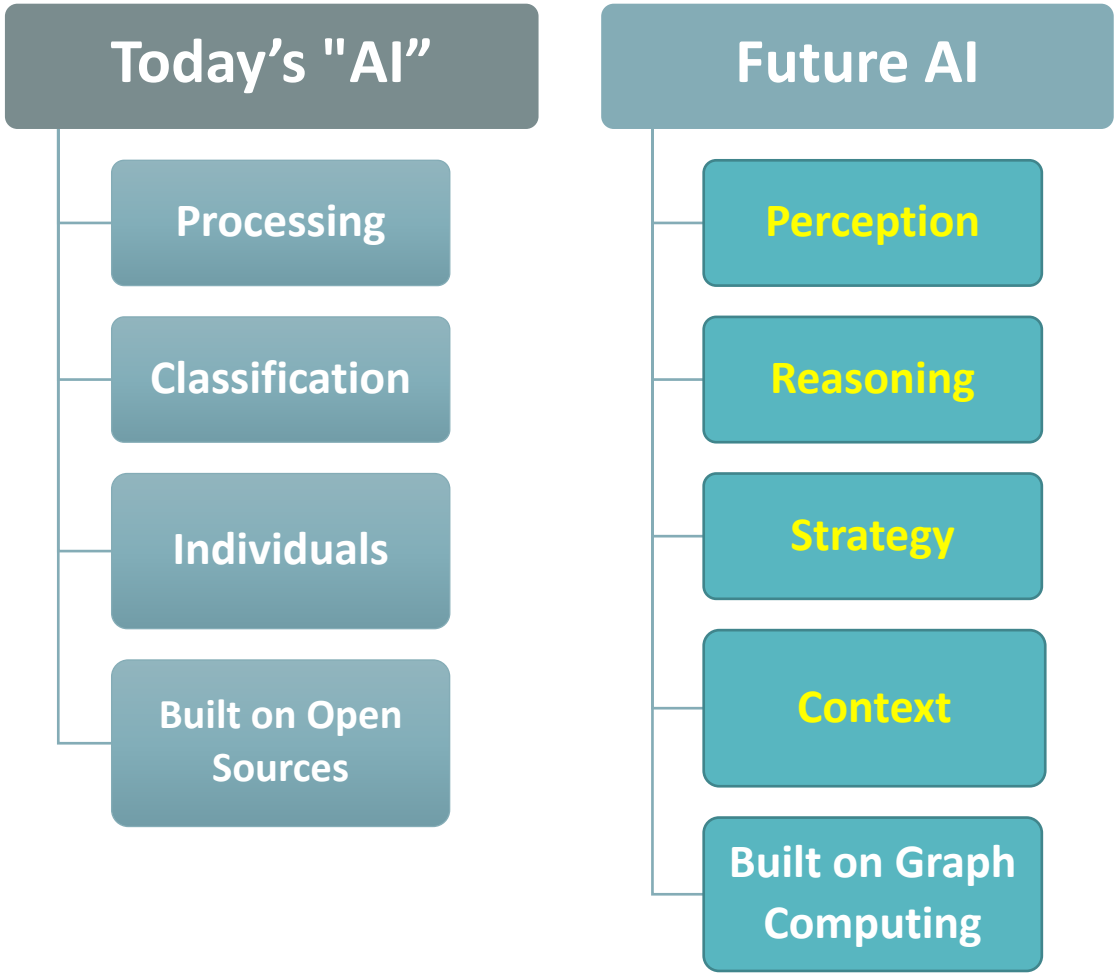
- Recognize an event from sensing -> feature extortion -> object/ scene/action detection -> semantics -> cognition inference



Example of Multi-Modality Multi-Layer Machine Understanding



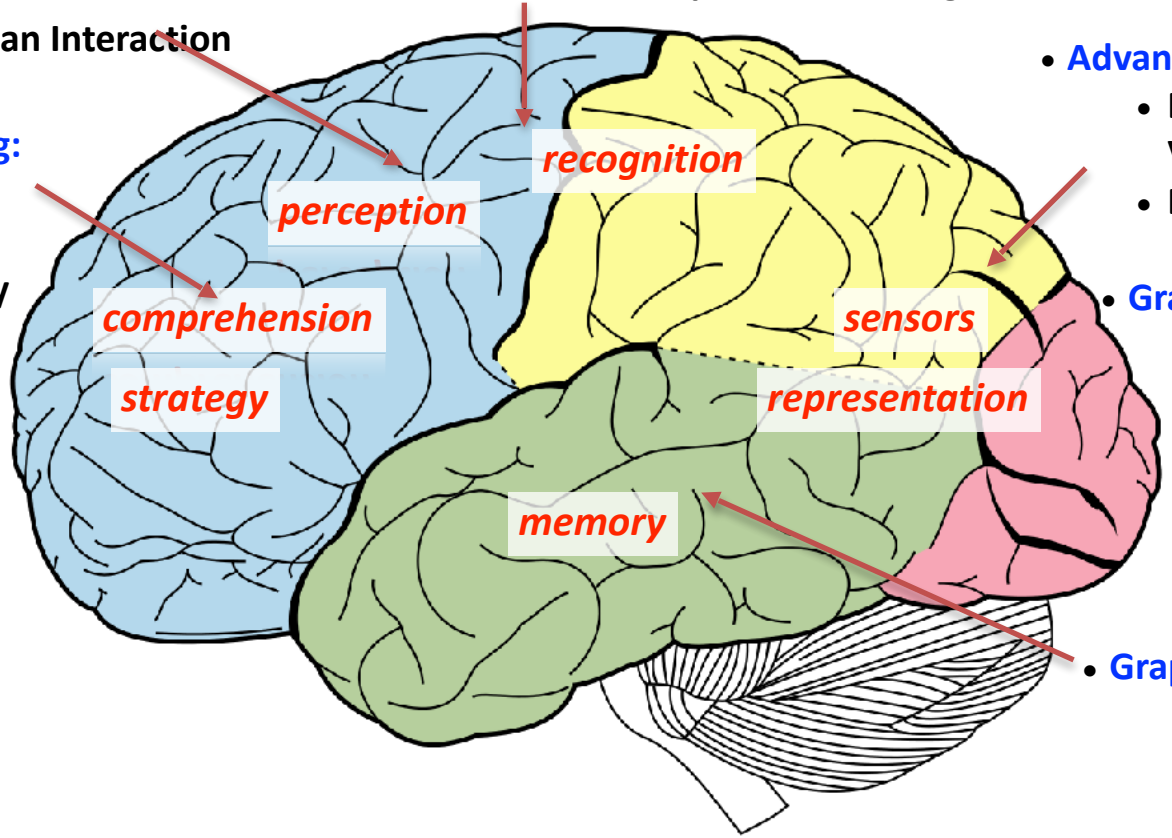
Building towards “Future AI”



Graphen's Ardi Platform makes full-brain AI possible

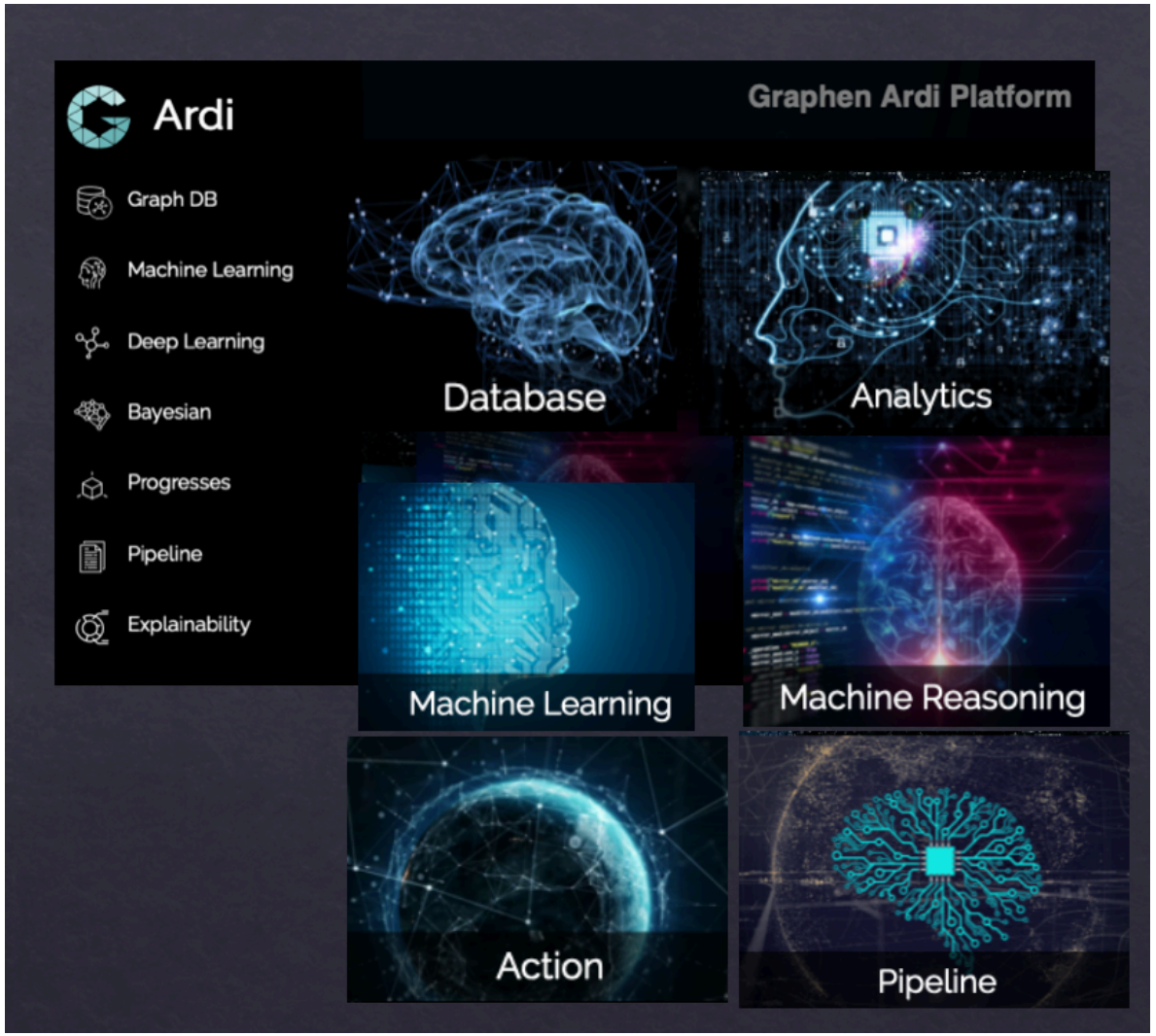
Most of other existing "AI" technology is only one of key fundamental components.

- **Machine Cognition:**
  - Robot Cognition Tools
  - Feeling
  - Robot-Human Interaction
- **Machine Learning:**
  - ML and Deep Learning
  - Autonomous Imperfect Learning
- **Machine Reasoning:**
  - Bayesian Networks
  - Game Theory Tools
- **Advanced Visualization:**
  - Dynamic and Interactiv Viz.
  - Big Data Viz.
- **Graph Analytics:**
  - Network Analysis
  - Flow Prediction
- **Graph Database:**
  - Distributed Native Database

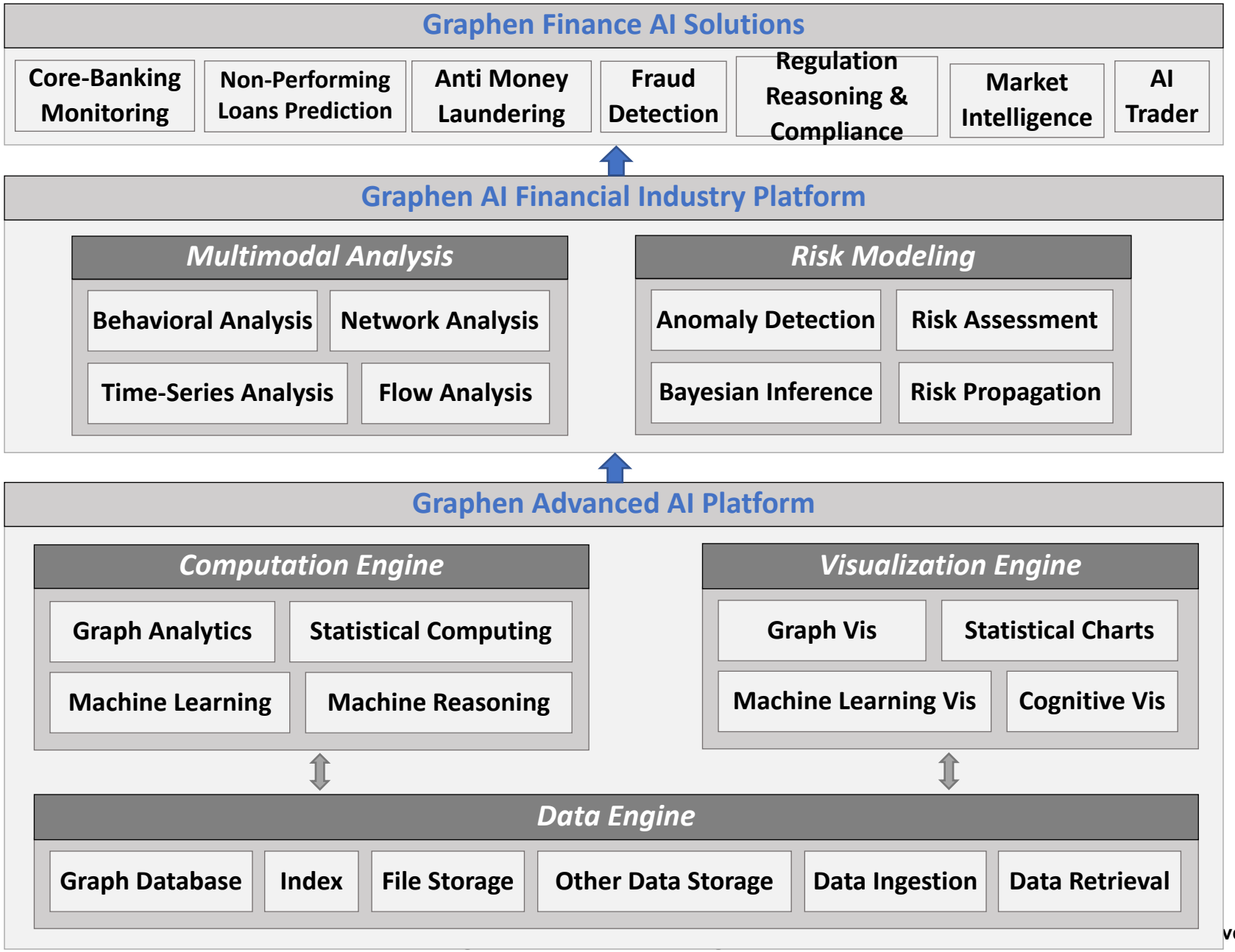


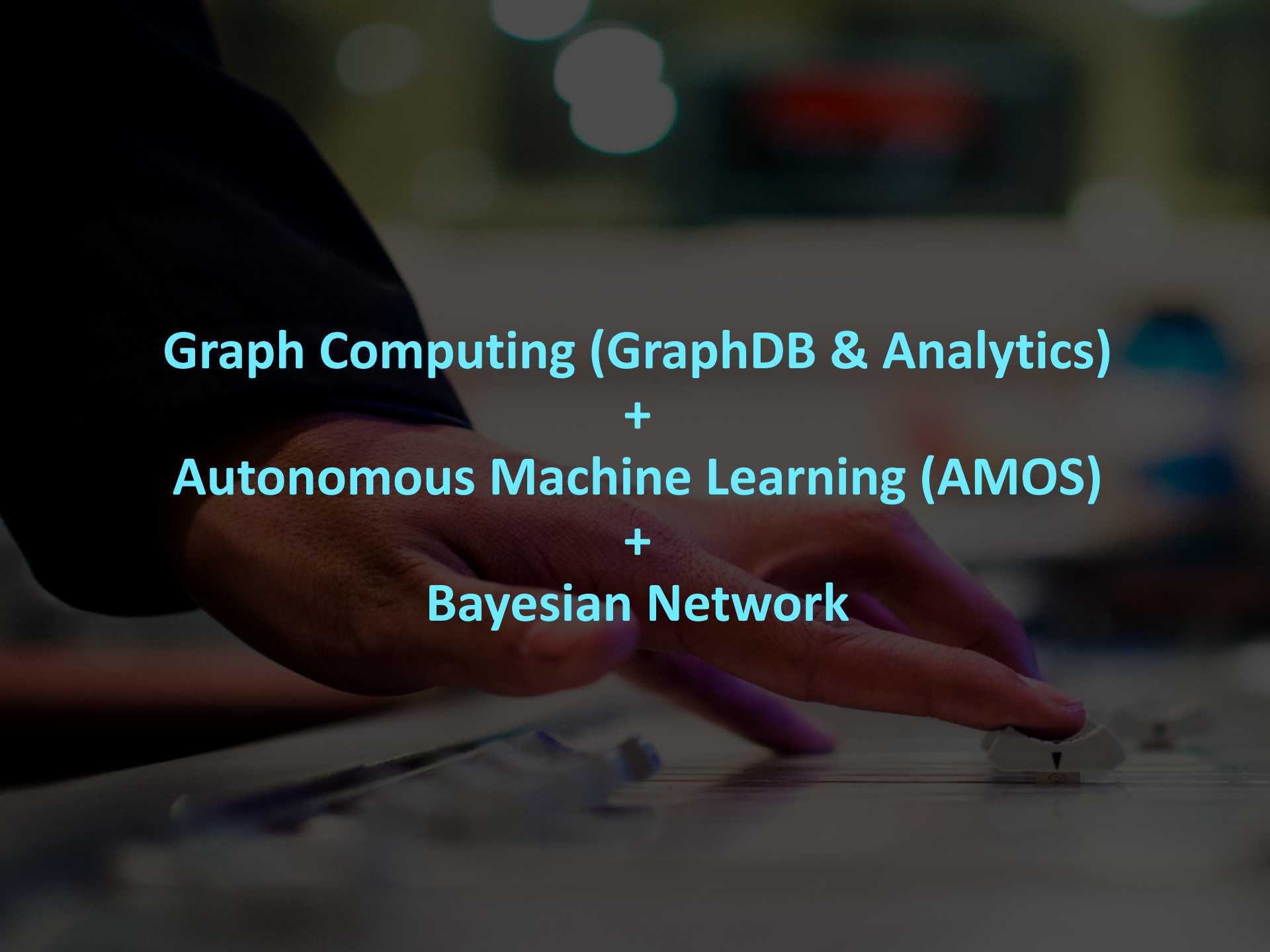
Enterprise-Level AI Platform

- Terabyte-sized native GraphDB, supports trillion of vertices and edges
- ACID-compliant and distributed Graph database and analytics
- Asynchronous job scheduling (both Autonomous ML and GraphDB)
- Scalable, distributed Analytics, modular and expandable through plugins
- Cluster, Replication and High-Availability with disaster recovery
- Error and event Logging, Monitoring, Backup and Recovery

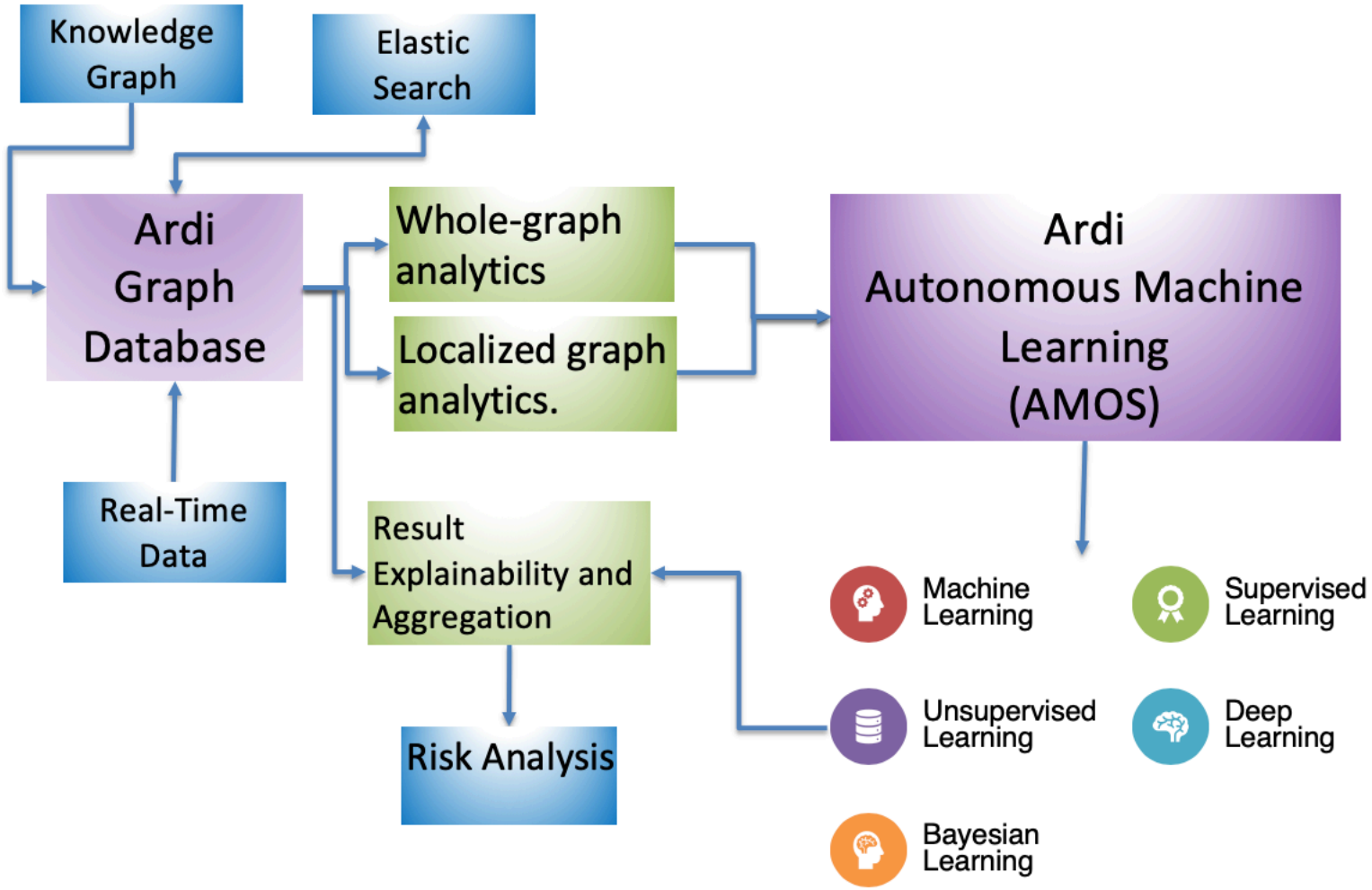


Example — Using AI Platform to Build Finance Products




Graph Computing (GraphDB & Analytics)
+
Autonomous Machine Learning (AMOS)
+
Bayesian Network

An Example Graphen Ardi Data Pipeline



*AMOS: Autonomous Model Optimizer System

Can you see fraud easily here ?

Company	Balance	Loans	Number of payment cycles	Majority Stakeholder	Phones sold	Buyers	VAT
A	100	11	1	Paul	£10M	B	
B	123	43	0	Sam	£10M	C	£1M
E	100	2000000	1	George	£10M	…	
C	345	45	10	Nicole	£10M	D	
D	65	1200	1	Paul	£10M	E	£1M
…							

Q: In RDBMS, given this data can we say if other companies are as good as E ?

How the carousel fraud works in 4 steps

Step 1

Company A (US) sells to Company B (Europe) €10M worth of phones.

Step 2

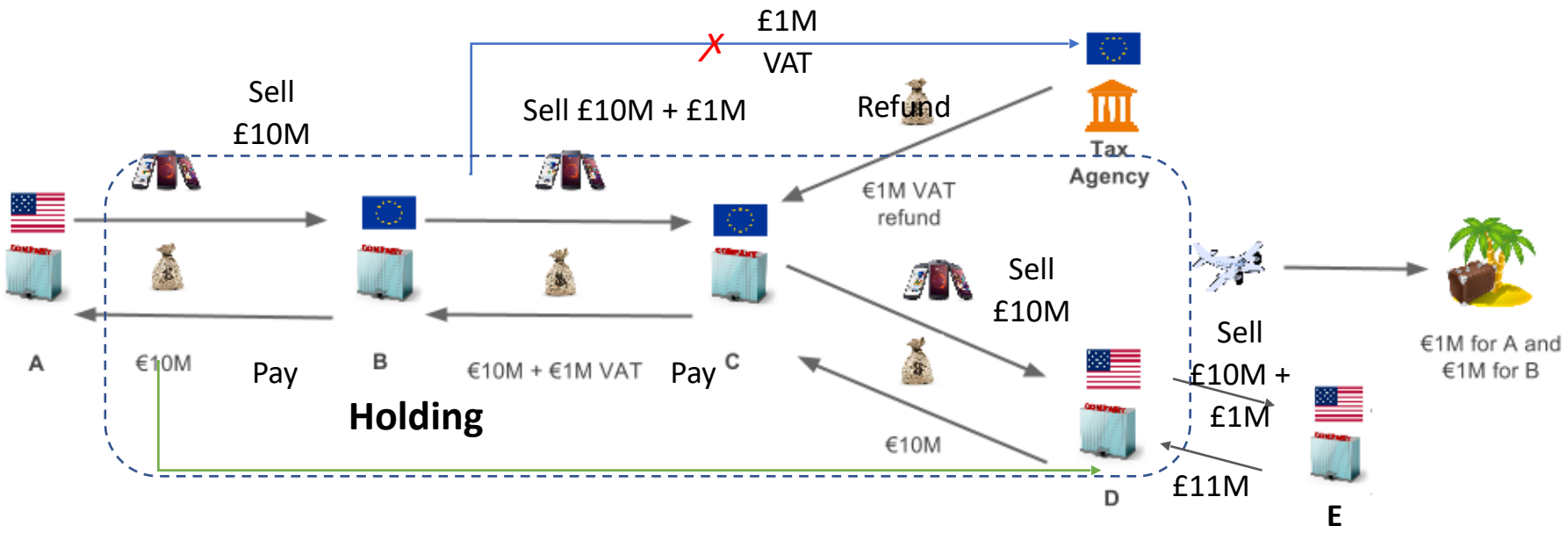
Company B sells the phones to company C. It charges €10M + €1M for the VAT.

Step 3

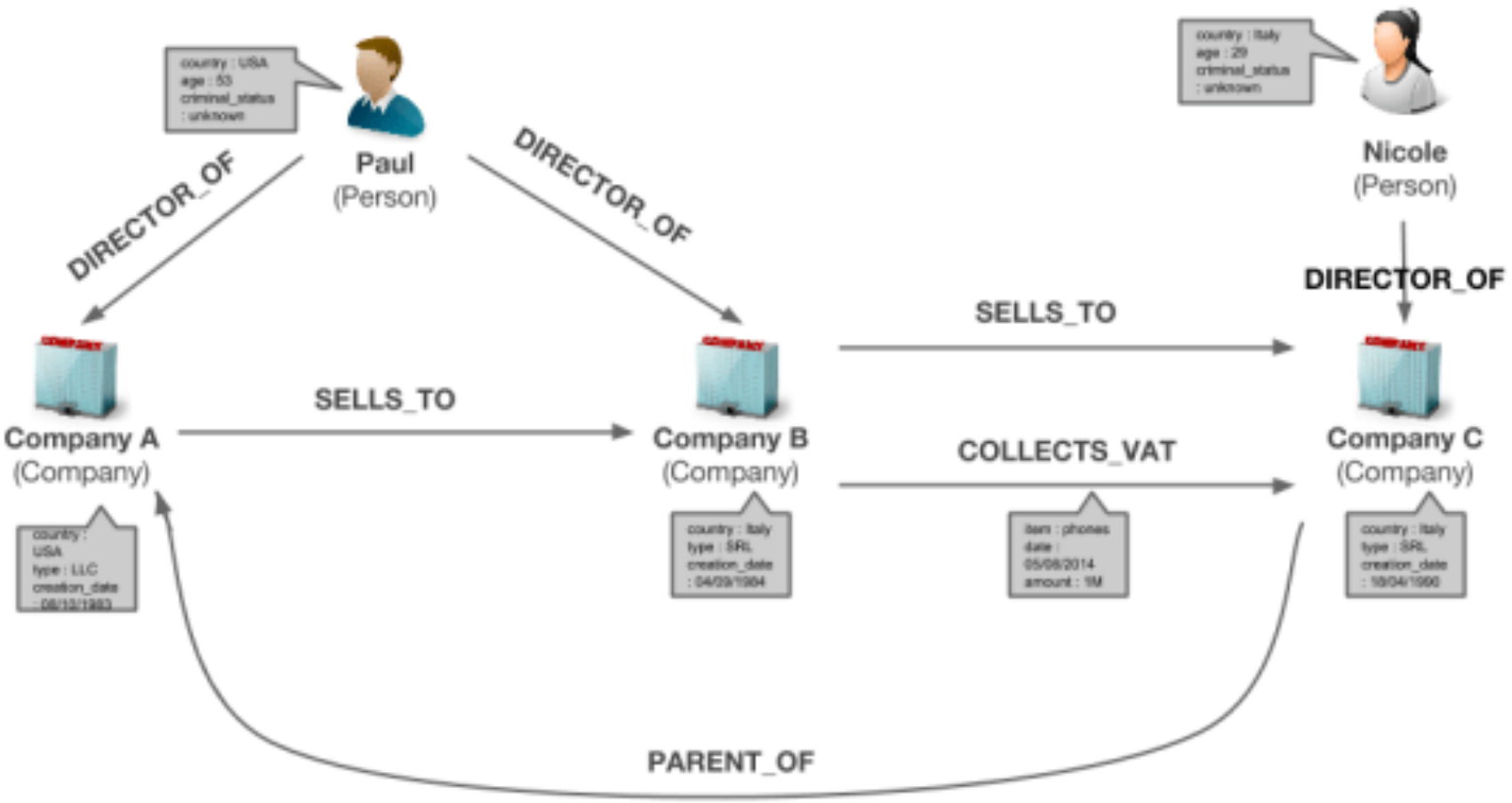
Company B sells the phones to company D (US) and claims a VAT refund.

Step 4

The directors of A and D disappear with €2M in stolen taxes.



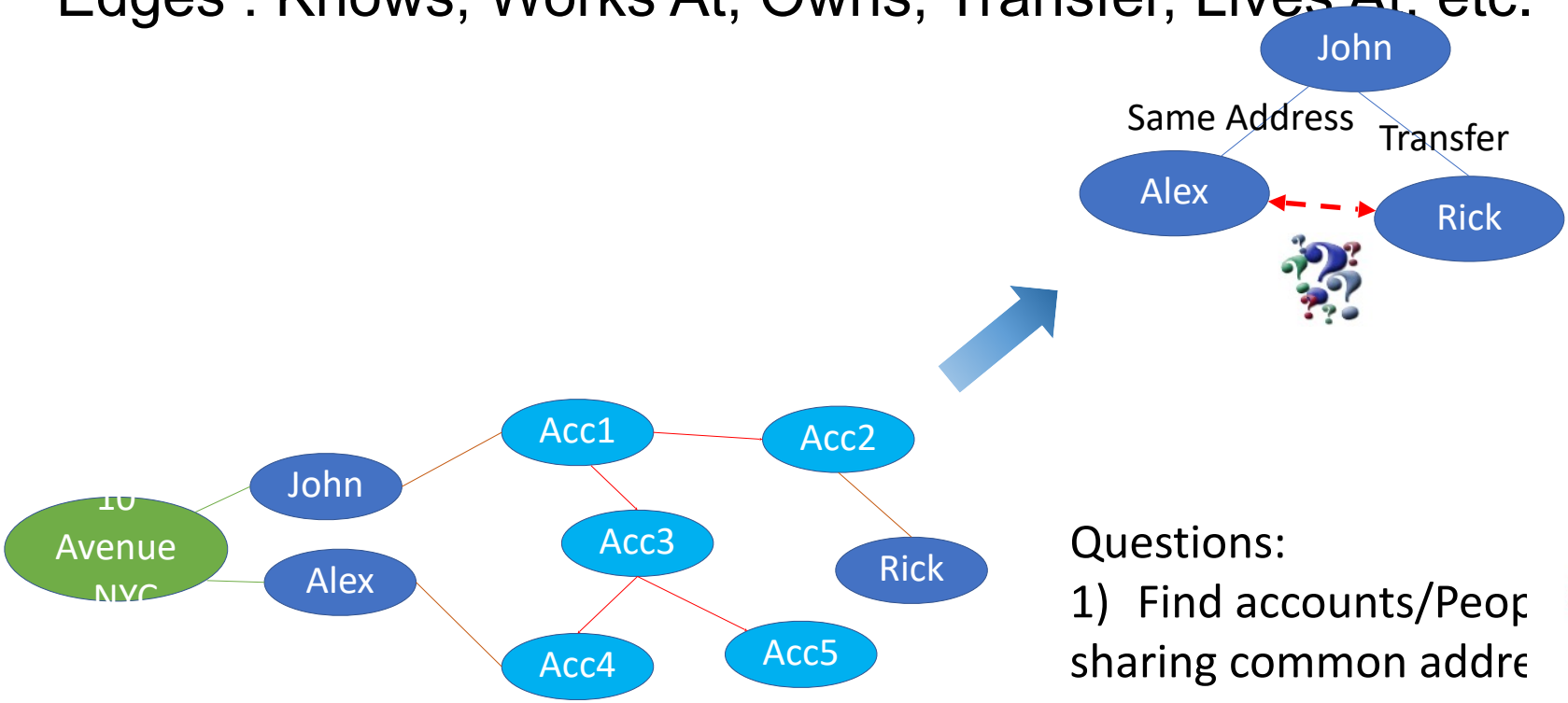
Stakeholders in fraud case



Real world graph representation

Vertices are People, Companies, Accounts, Address, Email

Edges : Knows, Works At, Owns, Transfer, Lives At, etc.



Questions:

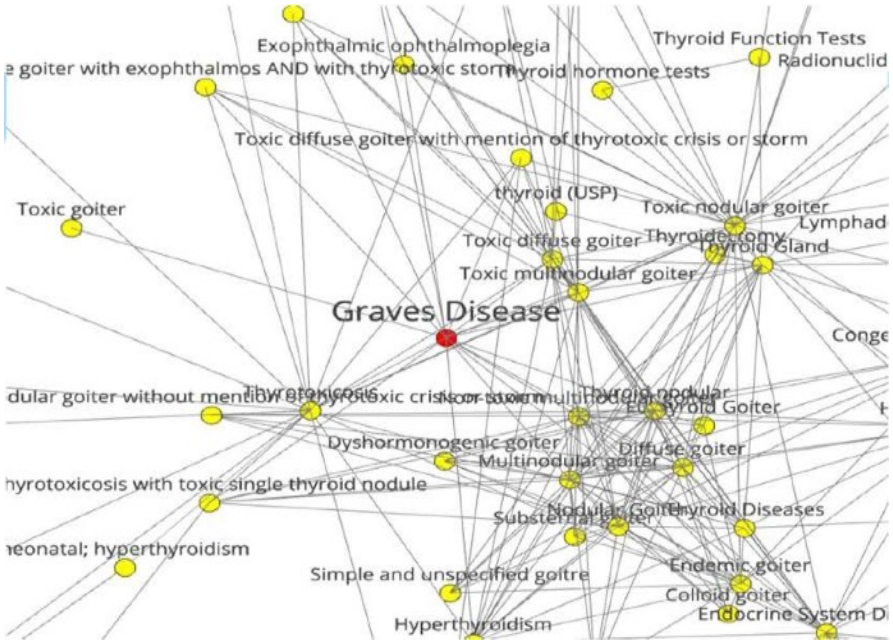
1) Find accounts/Peop
sharing common addre
phone, email, etc.

2) Infer indirect relationships

3) Estimate risk of fraud ring

Graphen Biomedical Knowledge Graph

Large-scale datasets
Uncertainty modeling
Complex relationships….



- 3.5 million biomedical concepts with 7.5 million relationships among them.

Growing List of Graph Analytics

- Shortest Path
- Connected
- Centrality
- Cycles
- Paths
- Cliques
- Egonet
- K-core
- Community
- Link Predictions

- MST - Minimum Spanning Tree
- Page Rank
- Sim Rank
- Strongly Connected Components
- Select Vertices
- Chaining of analytics:
  - Map
  - Reduce
  - Aggregated
  - Store

- Images
  - Image Classification and feature detection(TBD)

Deployments/Interactions

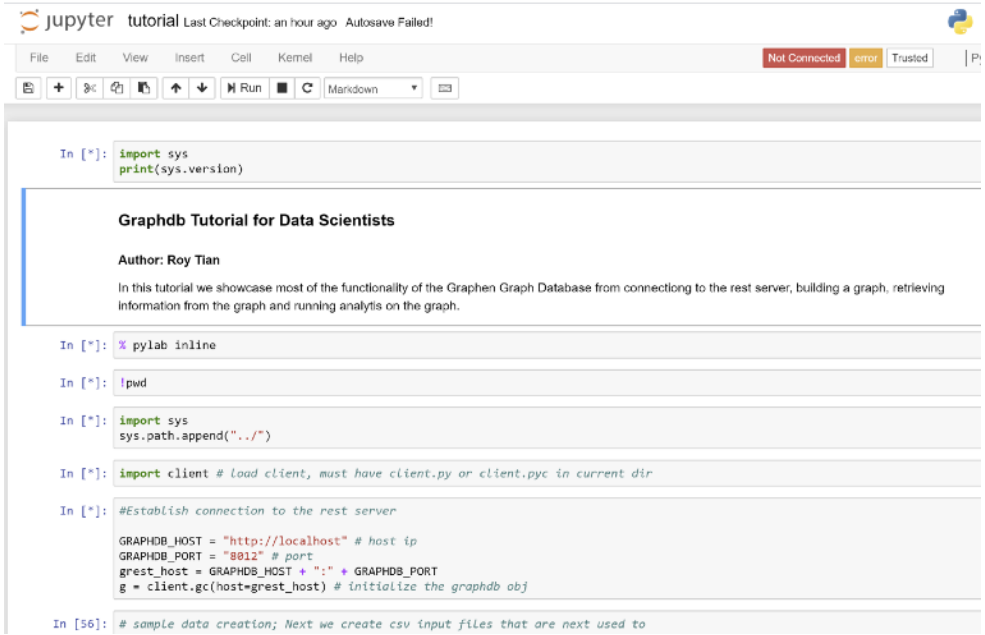
- Either using the local deployments or the cloud version:
  - Local deployments are through docker images.
  - Cloud version can directly use Ardi machine on AWS.
- Many ways to use:
  - Web UI with Graphen Viz through RestAPI
  - Rest client, with an wrapper, programs in Python but communicates through Rest

Local version

- tar zxf graphdb-client-master.tar.gz
- cd graphdb-client
- Run docker-run.sh to start the GraphDB server, it creates ~/graphdb-rest/csv and ~/graphdb-rest/data and mapped to inside docker, you may change the host directories through the docker run command.
- graphdb-client/tutorial has this Jupyter Notebook tutorial, run jupyter-notebook from the tutorial directory.
- The input csv files to be loaded must be placed in ~/graphdb-rest/csv/ directory
- The ~/graphdb-rest/data/ will contain the graph files
- tests/ contains some unit tests that can also be viewed for examples of analytics
- projects/ has some examples of more complicated dataset and analytics, feel free to explore

```
cyc@ubuntu1604:~$ tar zxf graphdb-client-master.tar.gz
cyc@ubuntu1604:~$ cd graphdb-client-master/
cyc@ubuntu1604:~/graphdb-client-master$ ls
client.pyc  csv  data  docker-run.sh  projects  README.md  tests  tutorial
cyc@ubuntu1604:~/graphdb-client-master$ cat docker-run.sh
docker run --network=dockernet -p 8012:80 -p 9090:9090 --rm -e "GRAPHDB_DATA=/data/systemG/graphdb-rest/database" -e "GRAPHDB_PLU
GINS=/opt/systemG/graphdb-rest/app/plugins" -e "GRAPHDB_FILE=/opt/systemG/graphdb-rest/app/csv" -e "ES_HOST=es5" -e "ES_PORT=9200
" -e "GRAPHDB_HOST=grest" -e "GRAPHDB_PORT=9090" -e "ES_ENABLED=0" -e "ASYNC_HOST=gasync" -e "ASYNC_PORT=8018" -e "KAFKA_BROKERS=
kafka1:9092" -e "KAFKA_TOPIC_NAME=systemg_test" --name grest -v ~/graphdb-rest/data:/data/systemG/graphdb-rest/database -v ~/grap
hdb-rest/csv:/opt/systemG/graphdb-rest/app/csv --log-opt mode=non-blocking --log-opt max-buffer-size=10m -d -e "GRAPHDB_LOG_LEVEL
=2" systemg/graphdb-rest:1.0-xenialcyc@ubuntu1604:~/graphdb-client-master$ ./docker-run.sh
5961e748eb8b26fa7cabbe302771e30ee9ea0f9bbcce154cb4b6d1f20ae0be2a
cyc@ubuntu1604:~/graphdb-client-master$ cd tutorial/
cyc@ubuntu1604:~/graphdb-client-master/tutorial$ jupyter-notebook
```

Open tutorial.ipynb Notebook



Make sure to append path where client.py is

*client.py is Python2-based, Python3 version will be available early February.

In this example, graphDB host and Jupyter Notebook is ran from the same host.

Ingest vertex CSV file into graphDB

Vertex csv file content, located in ~/graphdb-rest/csv

```
vertex_file_path = "vertex_2_files.csv"
has_header = 0
column_delimiter = ','
default_vertex_label = "2"
vertex_content_type = [
    {"position": [2, "STRING"]},
    {"age": [3, "DOUBLE"]},
]
column_number_map = {"vertex_id": 1,
                     "properties": vertex_content_type}

g.load_table_vertex(file_path=vertex_file_path,
                    has_header=has_header,
                    column_delimiter=column_delimiter,
                    default_vertex_label=default_vertex_label,
                    column_number_map=column_number_map,
                    column_header_map={},
                    content_type=vertex_content_type,
                    data_row_start=0,
                    data_row_end=0,
                    batch_size=10000)
```

Out[12]: '{"status": "success", "message": "5 vertices are added"}'

```
big_danny,old_man,90
daniel,engineer,35
gabi,boss,40
jun,engineer,35
tim,tester,45
```

Ingest edge CSV file into graphDB

In [12]:
```
# Ingest csv file into graphdb
edge_file_path = "edge_files.csv" # name of csv file
has_header = 0 # whether has header or not
column_delimiter = ',' # sep of csv
default_source_label = "1" # select the column for src_vertex label
default_target_label = "1" # select the column for tgt_vertex label
default_edge_label = "owes" # set default edge label

edge_content_type = [
    {"amt_of_money": [6, "DOUBLE"]}, # set properties, list of dicts;
                                     # dict is of format {"prop_name": [column, dtype]}
]

edge_column_number_map = {
    "source_id": 1,    # select column for src_vertex
    "source_label": 2, # select column for src_vertex_label
    "target_id": 3,    # select column for tgt_vertex
    "target_label": 4, # select column for tgt_vertex_label
    "edge_label": 5,   # select column for edge label
    "properties": edge_content_type
}

g.load_table_edge(file_path=edge_file_path,
                  has_header=has_header,
                  column_delimiter=column_delimiter,
                  default_source_label=default_source_label,
                  default_target_label=default_target_label,
                  default_edge_label=default_edge_label,
                  column_header_map={}, # csv header mapping
                  column_number_map=edge_column_number_map,
                  data_row_start=0, # skip rows
                  data_row_end=0, # end by rows
                  batch_size=3000)
```

Edge csv file content, located in ~/graphdb-rest/csv

```
roy,1,gabi,2,pays,1000
roy,1,andy,1,pays,50
andy,1,gabi,2,pays,25
daniel,2,bill,1,pays,100
daniel,2,jun,2,pays,200
big_danny,2,gabi,2,owes,50
roy,1,big_danny,2,owes,1
big_danny,2,tim,2,owes,1000
bill,1,tim,2,owes,10
gabi,2,daniel,2,owes,50
gabi,2,bill,1,owes,25
gabi,2,tim,2,owes,50
```

Read the graph schema and content as json

In [13]:
```
g.get_schema(graph_name="test_graph") # check the schema
```

Out[13]:

```
'{"status": "success", "schema": {"vertex_props": [{"prop_name": "position", "prop_type": "STRING"}, {"prop_name": "age", "prop
_type": "DOUBLE"}], "edge_labels": ["pays", "owes"], "vertex_labels": ["2", "1"], "edge_props": [{"prop_name": "amt_of_money",
"prop_type": "DOUBLE"}]}}'
```

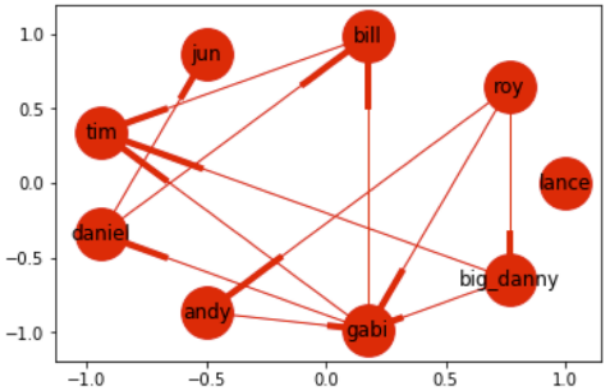

In [14]:
```
g.print_graph(graph_name="test_graph") # print the graph as json string
```

Out[14]:

```
'{"status": "success", "graph_type": 0, "statistics": {"num_edges": 12, "num_vertices": 9}, "data": {"edges": [{"source_label":
"2", "target_label": "2", "target_id": "gabi", "label": "owes", "eid": "13835058055282163712", "source_id": "big_danny", "prope
rties": [{"amt_of_money": 50.0}]}, {"source_label": "2", "target_label": "2", "target_id": "tim", "label": "owes", "eid": "1383
5058055282163714", "source_id": "big_danny", "properties": [{"amt_of_money": 1000.0}]}, {"source_label": "2", "target_label":
"1", "target_id": "bill", "label": "pays", "eid": "13835339530258874371", "source_id": "daniel", "properties": [{"amt_of_mone
y": 100.0}]}, {"source_label": "2", "target_label": "2", "target_id": "jun", "label": "pays", "eid": "13835339530258874372", "s
ource_id": "daniel", "properties": [{"amt_of_money": 200.0}]}, {"source_label": "2", "target_label": "2", "target_id": "danie
l", "label": "owes", "eid": "13835058055282163716", "source_id": "gabi", "properties": [{"amt_of_money": 50.0}]}, {"source_labe
l": "2", "target_label": "1", "target_id": "bill", "label": "owes", "eid": "13835058055282163717", "source_id": "gabi", "proper
ties": [{"amt_of_money": 25.0}]}, {"source_label": "2", "target_label": "2", "target_id": "tim", "label": "owes", "eid": "13835
058055282163718", "source_id": "gabi", "properties": [{"amt_of_money": 50.0}]}, {"source_label": "1", "target_label": "2", "tar
get_id": "gabi", "label": "pays", "eid": "13835339530258874370", "source_id": "andy", "properties": [{"amt_of_money": 25.0}]},
{"source_label": "1", "target_label": "2", "target_id": "tim", "label": "owes", "eid": "13835058055282163715", "source_id": "bi
```

Visualize

In [87]:
```
# Here we have a simple code on how to visualize the graph
# we need to make this into a function
import networkx as nx
import matplotlib.pyplot as plt

G = nx.DiGraph()
labels={}

verts = graph_to_dict["data"]["vertices"]
for v in verts:
    #print v
    G.add_node(v["id"])
    for p in v["properties"]:
        pname=p.keys()[0]
        pval=p.values()[0]
        #print pval,":::", pname
        print(pname)
        print(pval)
        print(v["id"])
        G.nodes[ v["id"] ][pname] = pval
    labels[v["id"]] = v["label"]

edges = graph_to_dict["data"]["edges"]
for e in edges:
    G.add_edge(e["source_id"],e["target_id"])

pos=nx.circular_layout(G) # positions for all nodes
nx.draw_networkx_nodes(G, pos, cmap=plt.get_cmap('jet'),
                       node_size = 1000)
nx.draw_networkx_labels(G, pos)
nx.draw_networkx_edges(G, pos, edge_color='r', arrows=True)
#nx.draw_networkx_edges(G, pos, edgelist=black_edges, arrows=False)
plt.show()
```



Query edges and neighbors

In [25]:
```
g.get_edge(source_id="roy", source_label="1", target_id="gabi", target_label="2")
```

Out[25]:
'{"status": "success", "statistics": {"num_edges": 1}, "data": {"edges": [{"source_label": "1", "target_label": "2", "target_id": "gabi", "label": "pays", "eid": "13835339530258874368", "source_id": "roy", "properties": [{"amt_of_money": 1000.0}]}]}}'

In [26]:
```
g.get_edge(source_id="gabi", source_label="2", target_id="roy", target_label="1")
```

Out[26]:
'{"status": "success", "statistics": {"num_edges": 0}, "data": {"edges": []}}'

In [27]:
```
# find the neighbors
g.get_neighbor_out(vertex_id="roy", vertex_label="1")
```

Out[27]:
'{"status": "success", "statistics": {"num_vertices": 3}, "data": {"vertices": [{"properties": [{"position": "boss"}, {"age": 40.0}], "id": "gabi", "label": "2"}, {"properties": [{"position": "data_scientist"}, {"age": 27.0}], "id": "andy", "label": "1"}, {"properties": [{"position": "old_man"}, {"age": 90.0}], "id": "big_danny", "label": "2"}]}}'

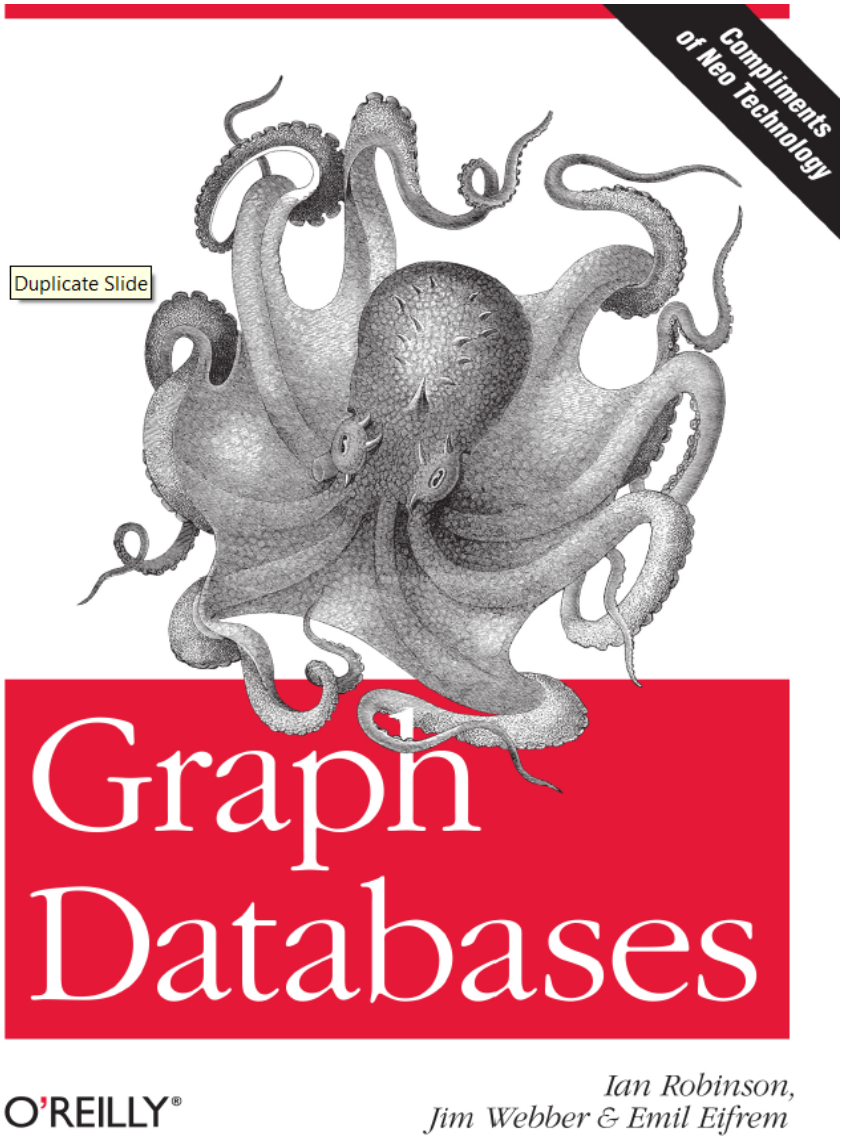
Example of Running analytics (Breadth-First Search)

In [33]:
```
json_str = '''
{
    "vertices": [{"id": "roy", "label": "1"}],
    "depth": 1,
    "with_vertex_props": "true"
}
'''

request_body = dict()
request_body["param"] = json_str

g.run_analytics(analytic_name="bfs", request_body=request_body)
```

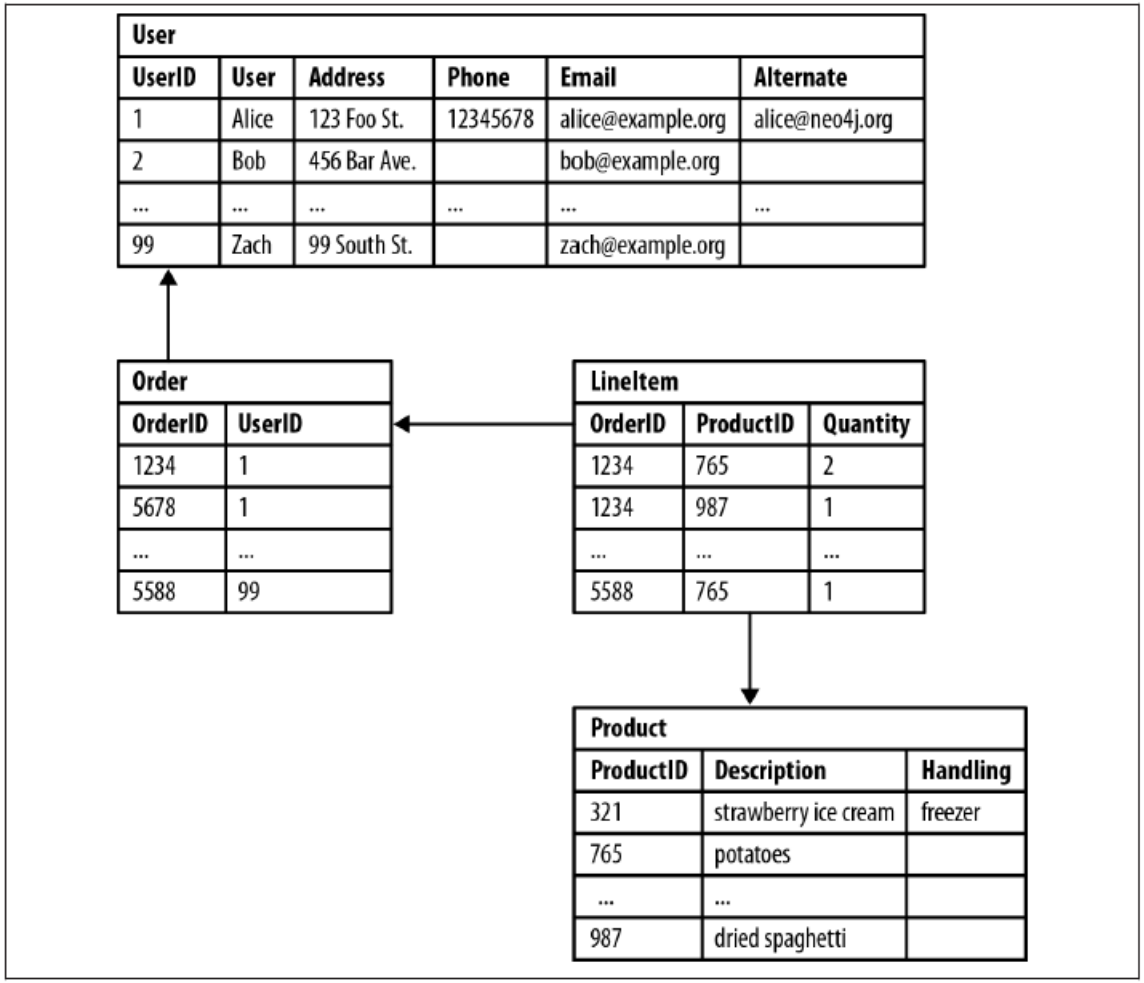
Out[33]:

```
'{"status": "success", "analytics": {}, "statistics": {"num_edges": 3, "num_vertices": 4}, "data": {"edges": [{"source_label":
"1", "target_label": "2", "target_id": "gabi", "label": "pays", "eid": 13835339530258874368, "source_id": "roy", "properties":
[]}, {"source_label": "1", "target_label": "1", "target_id": "andy", "label": "pays", "eid": 13835339530258874369, "source_id":
"roy", "properties": []}, {"source_label": "1", "target_label": "2", "target_id": "big_danny", "label": "owes", "eid": 13835058
055282163713, "source_id": "roy", "properties": []}], "vertices": [{"properties": [{"position": "old_man"}, {"age": 90.0}], "i
d": "big_danny", "label": "2"}, {"properties": [{"position": "data_scientist"}, {"age": 27.0}], "id": "andy", "label": "1"},
{"properties": [{"position": "boss"}, {"age": 40.0}], "id": "gabi", "label": "2"}, {"properties": [{"position": "engineer"},
{"age": 26.0}], "id": "roy", "label": "1"}]}}'
```

Reference



A usual example



User

UserID	User	Address	Phone	Email	Alternate
1	Alice	123 Foo St.	12345678	alice@example.org	alice@neo4j.org
2	Bob	456 Bar Ave.		bob@example.org	
...	...	...	...	...	...
99	Zach	99 South St.		zach@example.org	

Order

OrderID	UserID
1234	1
5678	1
...	...
5588	99

LineItem

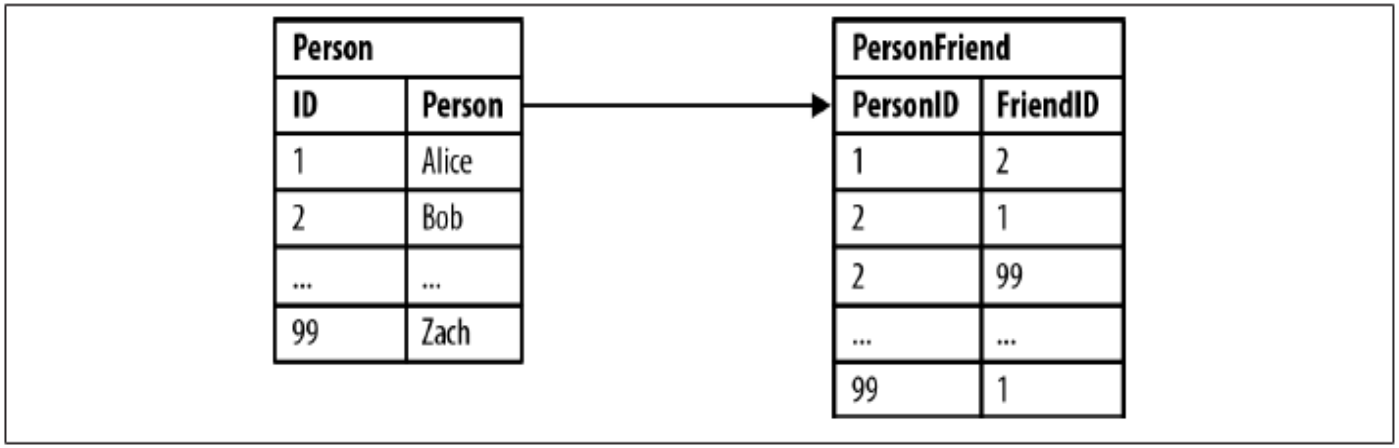
OrderID	ProductID	Quantity
1234	765	2
1234	987	1
...	...	...
5588	765	1

Product

ProductID	Description	Handling
321	strawberry ice cream	freezer
765	potatoes	
...	...	
987	dried spaghetti	

Figure 2-1. Semantic relationships are hidden in a relational database

Query Example – I



Person	
ID	Person
1	Alice
2	Bob
...	...
99	Zach

PersonFriend	
PersonID	FriendID
1	2
2	1
2	99
...	...
99	1

Figure 2-2. Modeling friends and friends-of-friends in a relational database

Asking "who are Bob's friends?" is easy, as shown in Example 2-1.

Example 2-1. Bob's friends

```
SELECT p1.Person
FROM Person p1 JOIN PersonFriend
  ON PersonFriend.FriendID = p1.ID
JOIN Person p2
  ON PersonFriend.PersonID = p2.ID
WHERE p2.Person = 'Bob'
```

Query Examples – II & III

Example 2-2. Who is friends with Bob?

```
SELECT p1.Person
FROM Person p1 JOIN PersonFriend
  ON PersonFriend.PersonID = p1.ID
JOIN Person p2
  ON PersonFriend.FriendID = p2.ID
WHERE p2.Person = 'Bob'
```

Example 2-3. Alice's friends-of-friends

```
SELECT p1.Person AS PERSON, p2.Person AS FRIEND_OF_FRIEND
FROM PersonFriend pf1 JOIN Person p1
  ON pf1.PersonID = p1.ID
JOIN PersonFriend pf2
  ON pf2.PersonID = pf1.FriendID
JOIN Person p2
  ON pf2.FriendID = p2.ID
WHERE p1.Person = 'Alice' AND pf2.FriendID <> p1.ID
```

Computational intensive

Graph Database Example

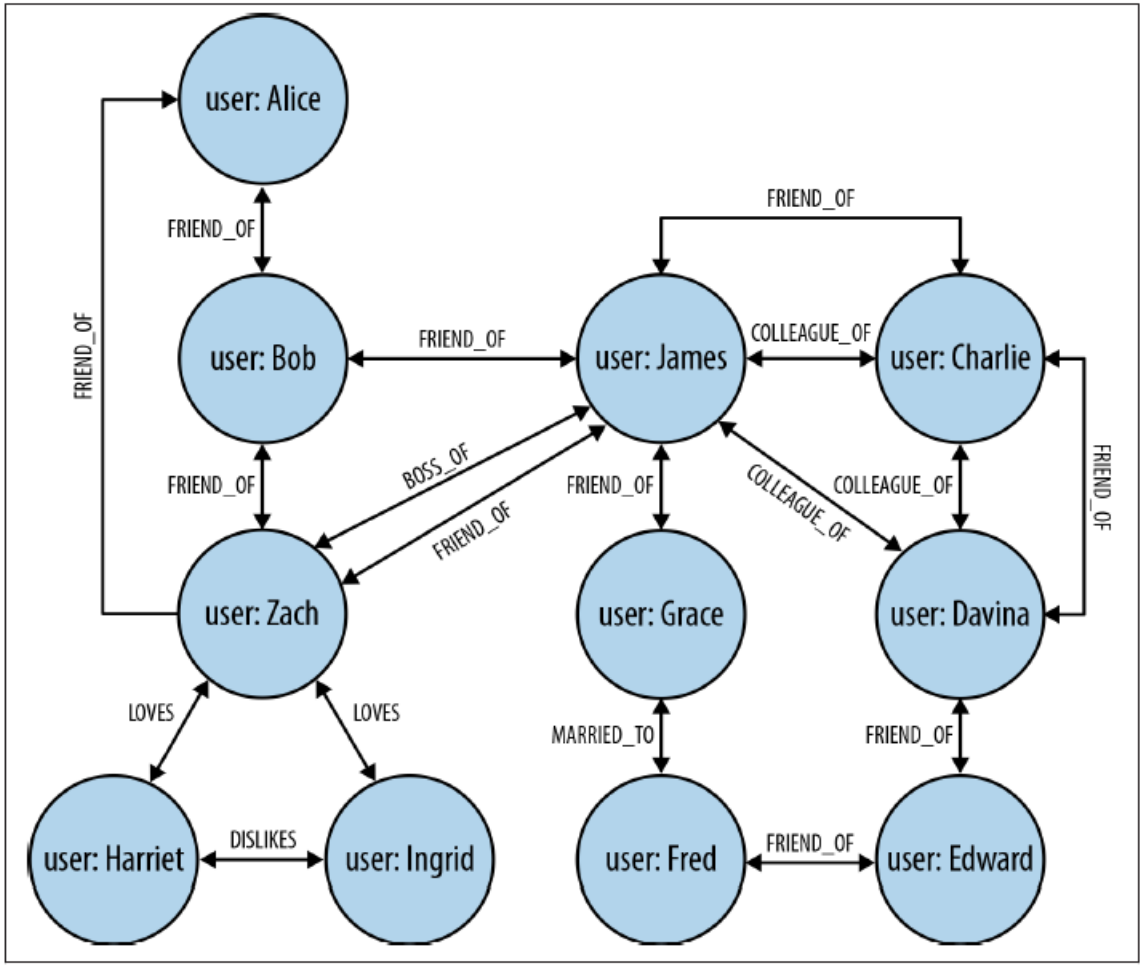


Figure 2-5. Easily modeling friends, colleagues, workers, and (unrequited) lovers in a graph

Execution Time in the example of finding extended friends (by Neo4j)

Partner and Vukotic's experiment seeks to find friends-of-friends in a social network, to a maximum depth of five. Given any two persons chosen at random, is there a path that connects them that is at most five relationships long? For a social network containing 1,000,000 people, each with approximately 50 friends, the results strongly suggest that graph databases are the best choice for connected data, as we see in Table 2-1.

Table 2-1. Finding extended friends in a relational database versus efficient finding in Neo4j

Depth	RDBMS execution time (s)	Neo4j execution time (s)	Records returned
2	0.016	0.01	~2500
3	30.267	0.168	~110,000
4	1543.505	1.359	~600,000
5	Unfinished	2.132	~800,000

Modeling Order History as a Graph

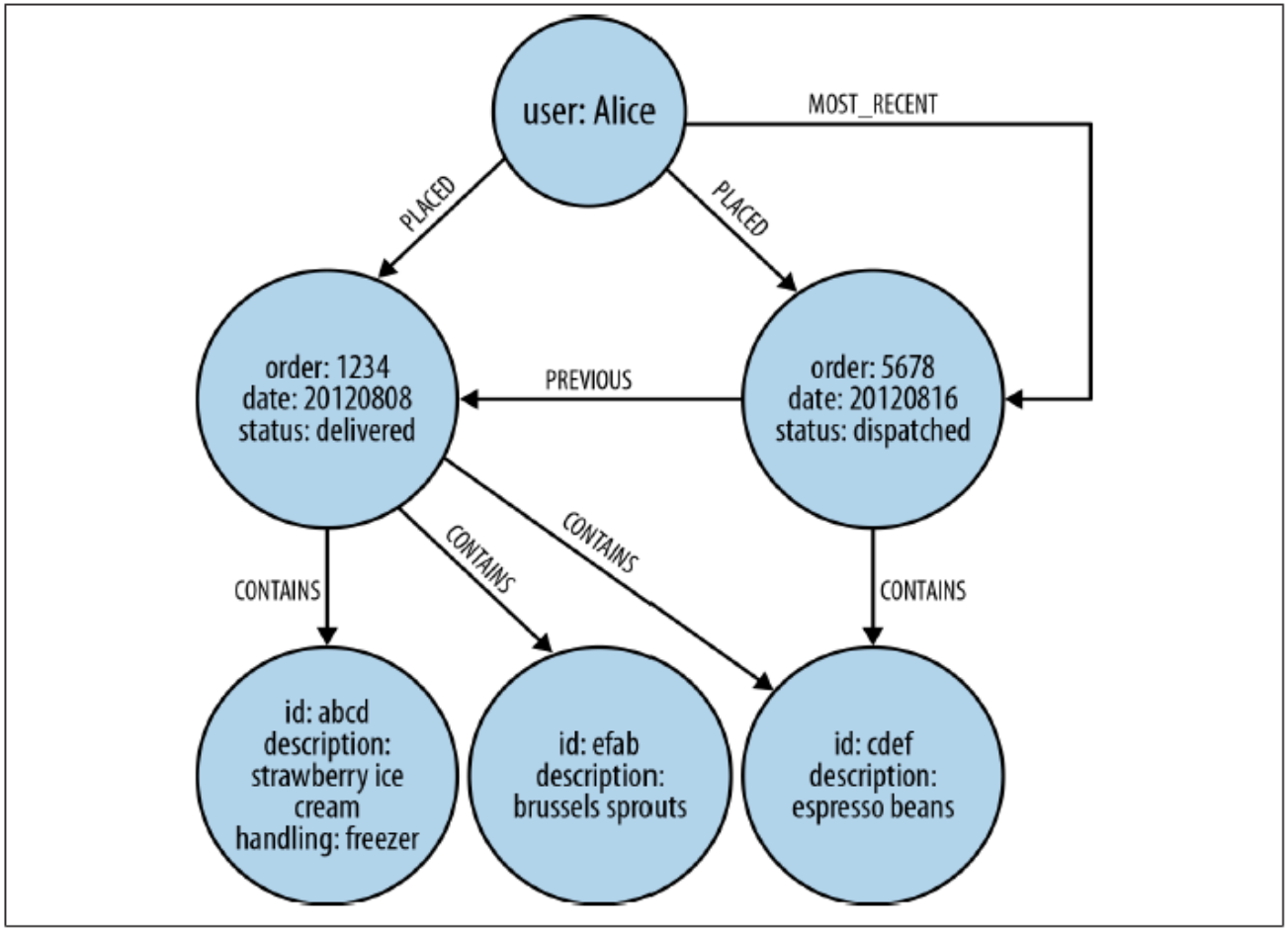


Figure 2-6. Modeling a user's order history in a graph

A query language on Property Graph – Cypher

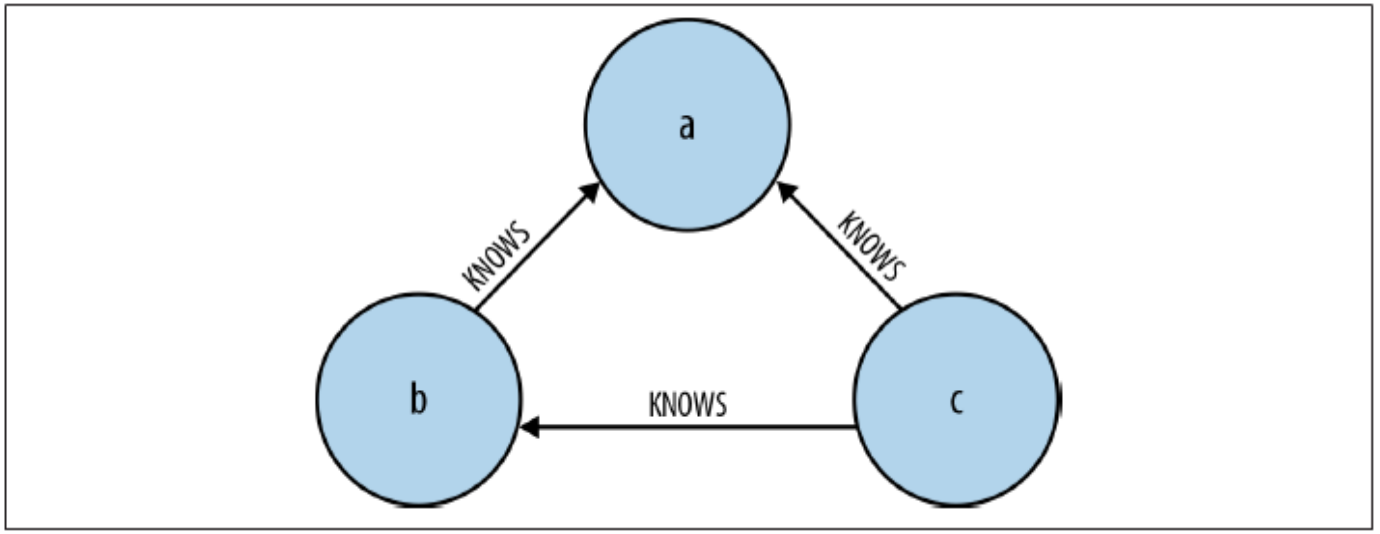


Figure 3-1. A simple graph pattern, expressed using a diagram

This pattern describes three mutual friends. Here's the equivalent ASCII art representation in Cypher:

```
(a)-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:KNOWS]->(c)
```

Cypher Example

Like most query languages, Cypher is composed of clauses. The simplest queries consist of a `START` clause followed by a `MATCH` and a `RETURN` clause (we'll describe the other clauses you can use in a Cypher query later in this chapter). Here's an example of a Cypher query that uses these three clauses to find the mutual friends of user named *Michael*:

```
START a=node:user(name='Michael')
MATCH (a)-[:KNOWS]->(b)-[:KNOWS]->(c), (a)-[:KNOWS]->(c)
RETURN b, c
```

Other Cypher Clauses

`WHERE`
: Provides criteria for filtering pattern matching results.

`CREATE` *and* `CREATE UNIQUE`
: Create nodes and relationships.

`DELETE`
: Removes nodes, relationships, and properties.

`SET`
: Sets property values.

`FOREACH`
: Performs an updating action for each element in a list.

`UNION`
: Merges results from two or more queries (introduced in Neo4j 2.0).

`WITH`
: Chains subsequent query parts and forward results from one to the next. Similar to piping commands in Unix.

Property Graph Example – Shakespeare

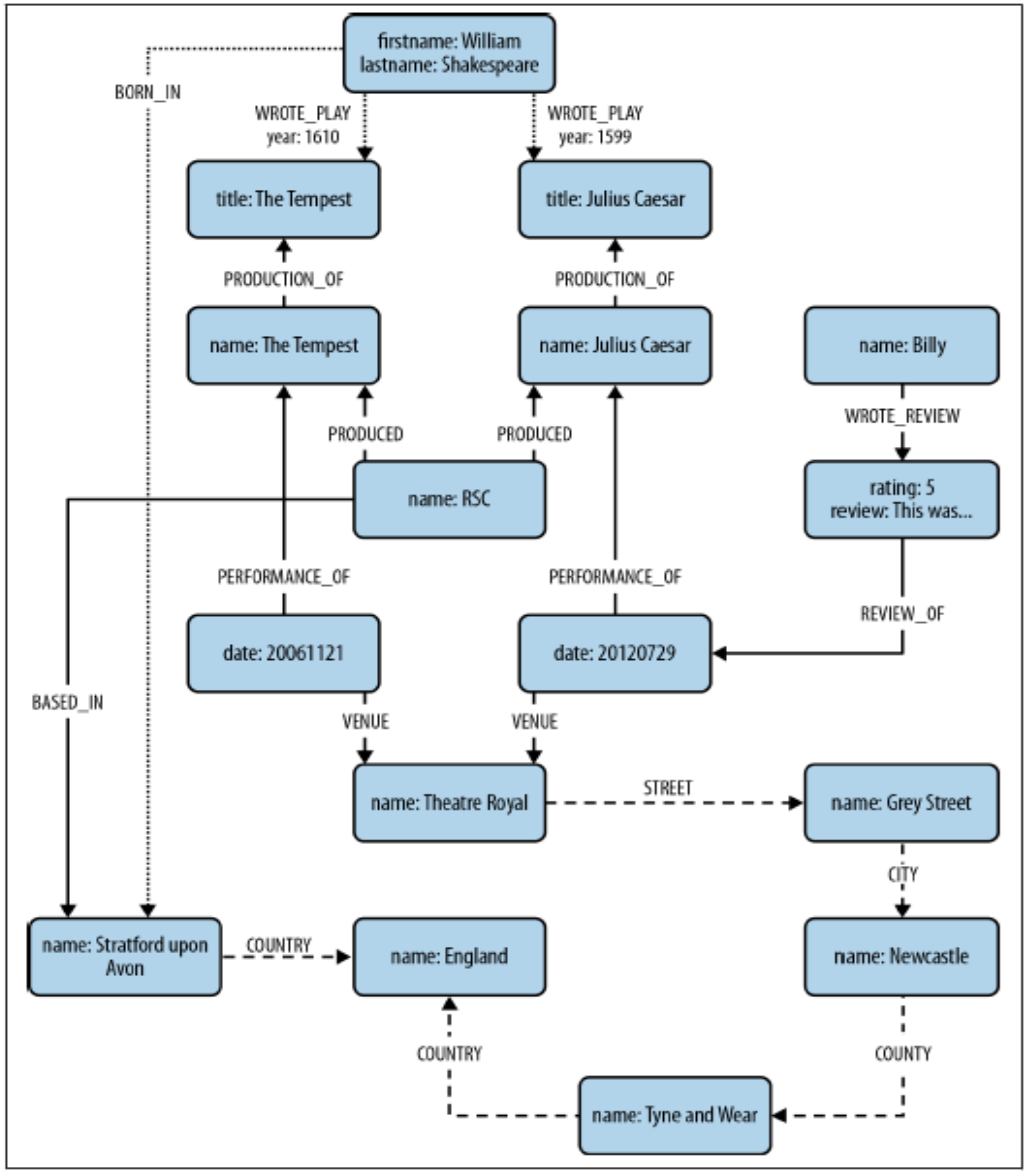


Figure 3-6. Three domains in one graph

Creating the Shakespeare Graph

```
CREATE (shakespeare { firstname: 'William', lastname: 'Shakespeare' }),
       (juliusCaesar { title: 'Julius Caesar' }),
       (shakespeare)-[:WROTE_PLAY { year: 1599 }]->(juliusCaesar),
       (theTempest { title: 'The Tempest' }),
       (shakespeare)-[:WROTE_PLAY { year: 1610}]->(theTempest),
       (rsc { name: 'RSC' }),
       (production1 { name: 'Julius Caesar' }),
       (rsc)-[:PRODUCED]->(production1),
       (production1)-[:PRODUCTION_OF]->(juliusCaesar),
       (performance1 { date: 20120729 }),
       (performance1)-[:PERFORMANCE_OF]->(production1),
       (production2 { name: 'The Tempest' }),
       (rsc)-[:PRODUCED]->(production2),
       (production2)-[:PRODUCTION_OF]->(theTempest),
       (performance2 { date: 20061121 }),
       (performance2)-[:PERFORMANCE_OF]->(production2),
       (performance3 { date: 20120730 }),
       (performance3)-[:PERFORMANCE_OF]->(production1),
       (billy { name: 'Billy' }),
       (review { rating: 5, review: 'This was awesome!' }),
       (billy)-[:WROTE_REVIEW]->(review),
       (review)-[:RATED]->(performance1),
       (theatreRoyal { name: 'Theatre Royal' }),
       (performance1)-[:VENUE]->(theatreRoyal),
       (performance2)-[:VENUE]->(theatreRoyal),
       (performance3)-[:VENUE]->(theatreRoyal),
       (greyStreet { name: 'Grey Street' }),
       (theatreRoyal)-[:STREET]->(greyStreet),
       (newcastle { name: 'Newcastle' }),
       (greyStreet)-[:CITY]->(newcastle),
       (tyneAndWear { name: 'Tyne and Wear' }),
       (newcastle)-[:COUNTY]->(tyneAndWear),
       (england { name: 'England' }),
       (tyneAndWear)-[:COUNTRY]->(england),
       (stratford { name: 'Stratford upon Avon' }),
       (stratford)-[:COUNTRY]->(england),
       (rsc)-[:BASED_IN]->(stratford),
       (shakespeare)-[:BORN_IN]->stratford
```

Query on the Shakespeare Graph

```
START theater=node:venue(name='Theatre Royal'),
      newcastle=node:city(name='Newcastle'),
      bard=node:author(lastname='Shakespeare')
MATCH (newcastle)<-[:STREET|CITY*1..2]-(theater)
      <-[:VENUE]-()-[:PERFORMANCE_OF]->()-[:PRODUCTION_OF]->
      (play)<-[w:WROTE_PLAY]-(bard)
WHERE w.year > 1608
RETURN DISTINCT play.title AS play
```

Adding this `WHERE` clause means that for each successful match, the Cypher execution engine checks that the `WROTE_PLAY` relationship between the Shakespeare node and the matched play has a `year` property with a value greater than 1608. Matches with a `WROTE_PLAY` relationship whose `year` value is greater than 1608 will pass the test; these plays will then be included in the results. Matches that fail the test will not be included in the results. By adding this clause, we ensure that only plays from Shakespeare's late period are returned:

```
+---------------+
| play          |
+---------------+
| "The Tempest" |
+---------------+
1 row
```

Another Query on the Shakespeare Graph

```
START theater=node:venue(name='Theatre Royal'),
      newcastle=node:city(name='Newcastle'),
      bard=node:author(lastname='Shakespeare')
MATCH (newcastle)<-[:STREET|CITY*1..2]-(theater)
      <-[:VENUE]-()-[p:PERFORMANCE_OF]->()-[:PRODUCTION_OF]->
      (play)<-[:WROTE_PLAY]-(bard)
RETURN   play.title AS play, count(p) AS performance_count
ORDER BY performance_count DESC
```

The RETURN clause here counts the number of PERFORMANCE_OF relationships using the identifier p (which is bound to the PERFORMANCE_OF relationships in the MATCH clause) and aliases the result as performance_count. It then orders the results based on performance_count, with the most frequently performed play listed first:

```
+-------------------------------------+
| play           | performance_count  |
+-------------------------------------+
| "Julius Caesar" | 2                 |
| "The Tempest"   | 1                 |
+-------------------------------------+
2 rows
```

Building Application Example – Collaborative Filtering

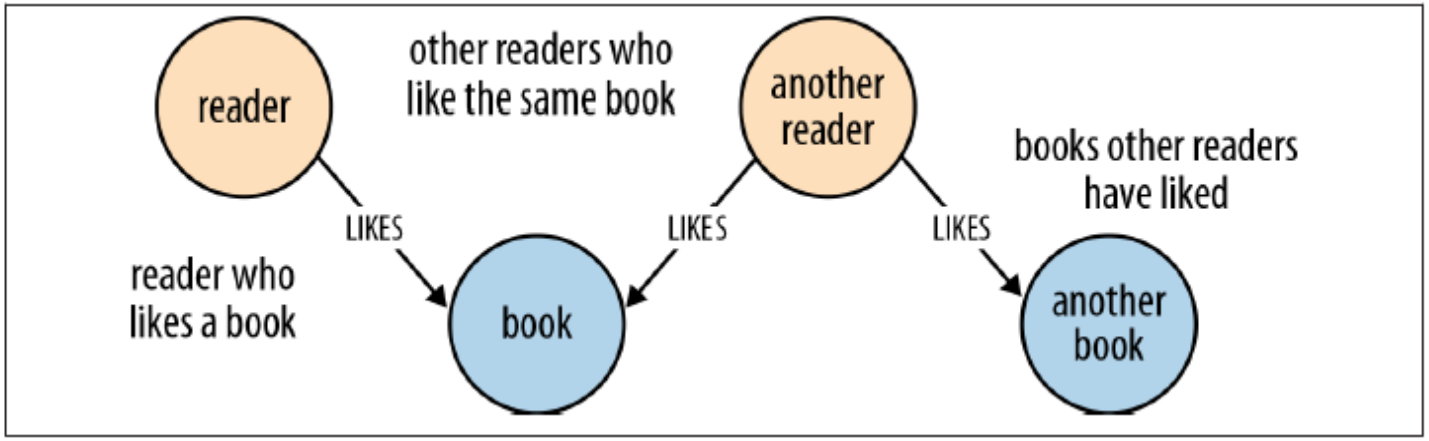


Figure 4-1. Data model for the book reviews user story

Because this data model directly encodes the question presented by the user story, it lends itself to being queried in a way that similarly reflects the structure of the question we want to ask of the data:

```
START reader=node:users(name={readerName})
      book=node:books(isbn={bookISBN})
MATCH reader-[:LIKES]->book<-[:LIKES]-other_readers-[:LIKES]->books
RETURN books.title
```

Chaining on the Query

```
START bard=node:author(lastname='Shakespeare')
MATCH (bard)-[w:WROTE_PLAY]->(play)
WITH play
ORDER BY w.year DESC
RETURN collect(play.title) AS plays
```

Executing this query against our sample graph produces the following result:

```
+---------------------------------+
| plays                           |
+---------------------------------+
| ["The Tempest","Julius Caesar"] |
+---------------------------------+
1 row
```

Example – Email Interaction Graph

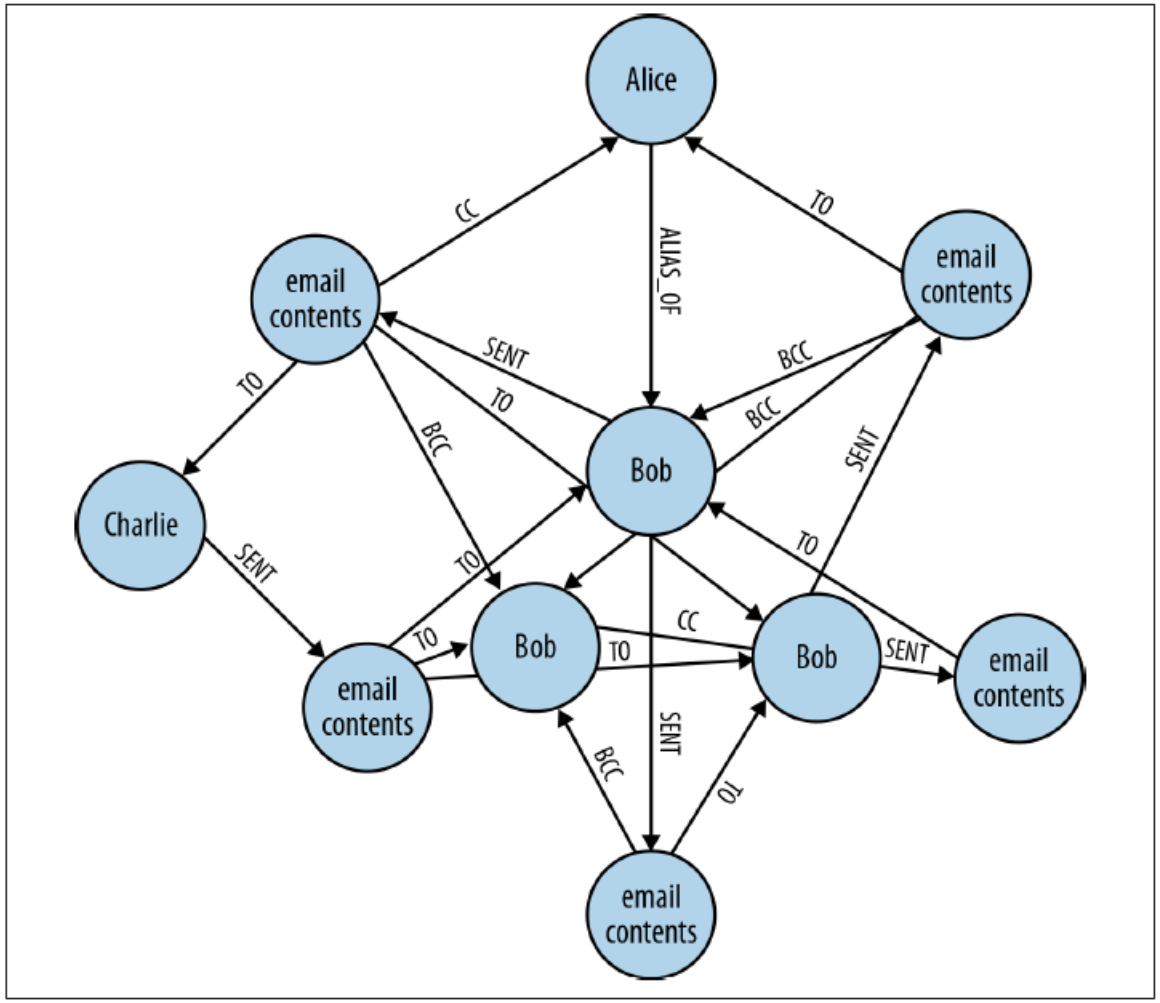


Figure 3-10. A graph of email interactions

```
START bob=node:user(username='Bob')
MATCH (bob)-[:SENT]->(email)-[:CC]->(alias),
      (alias)-[:ALIAS_OF]->(bob)
RETURN email
```

What's this query for?

How to make graph database fast?

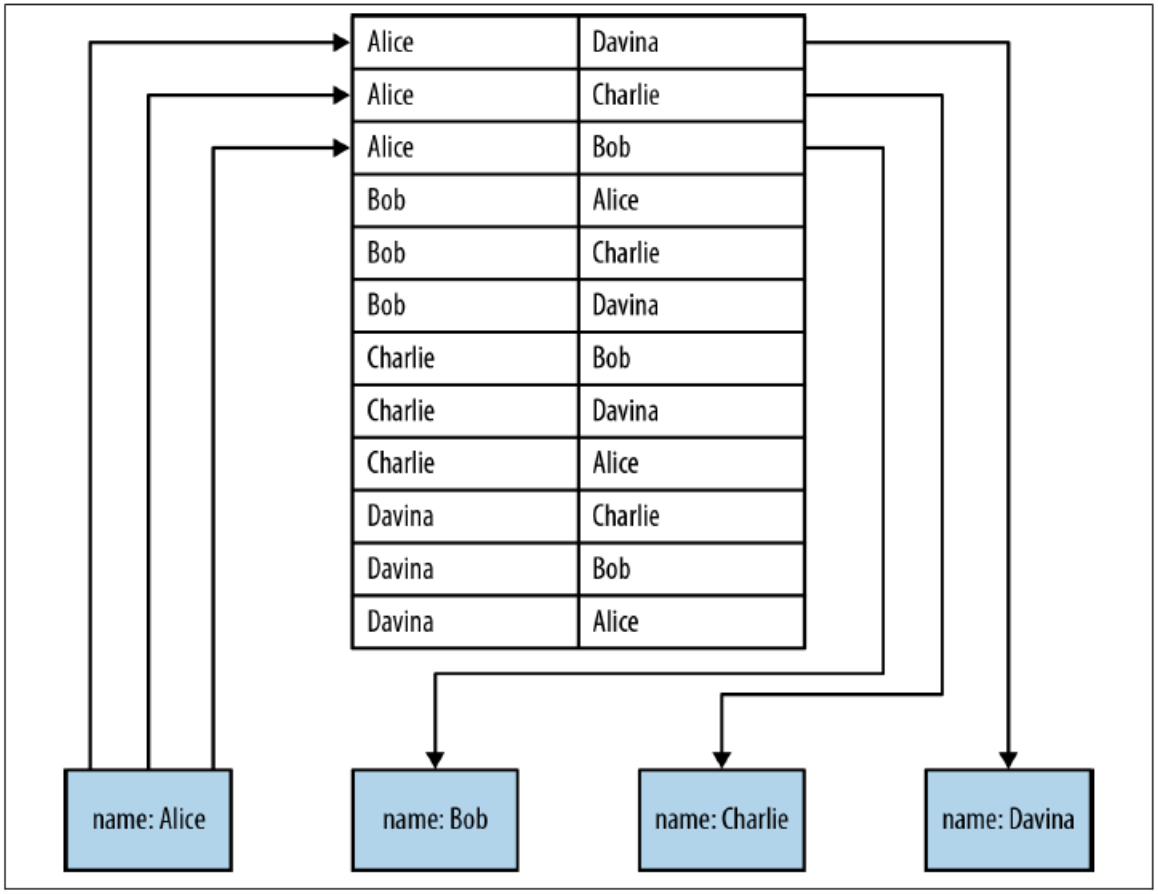
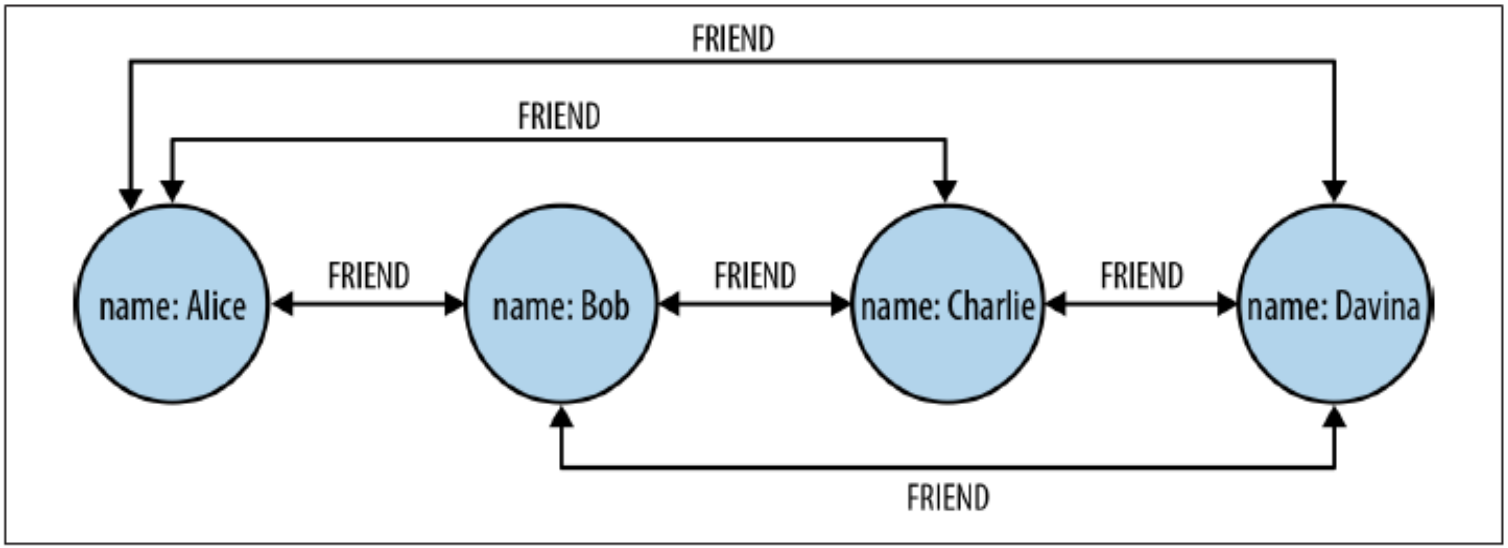


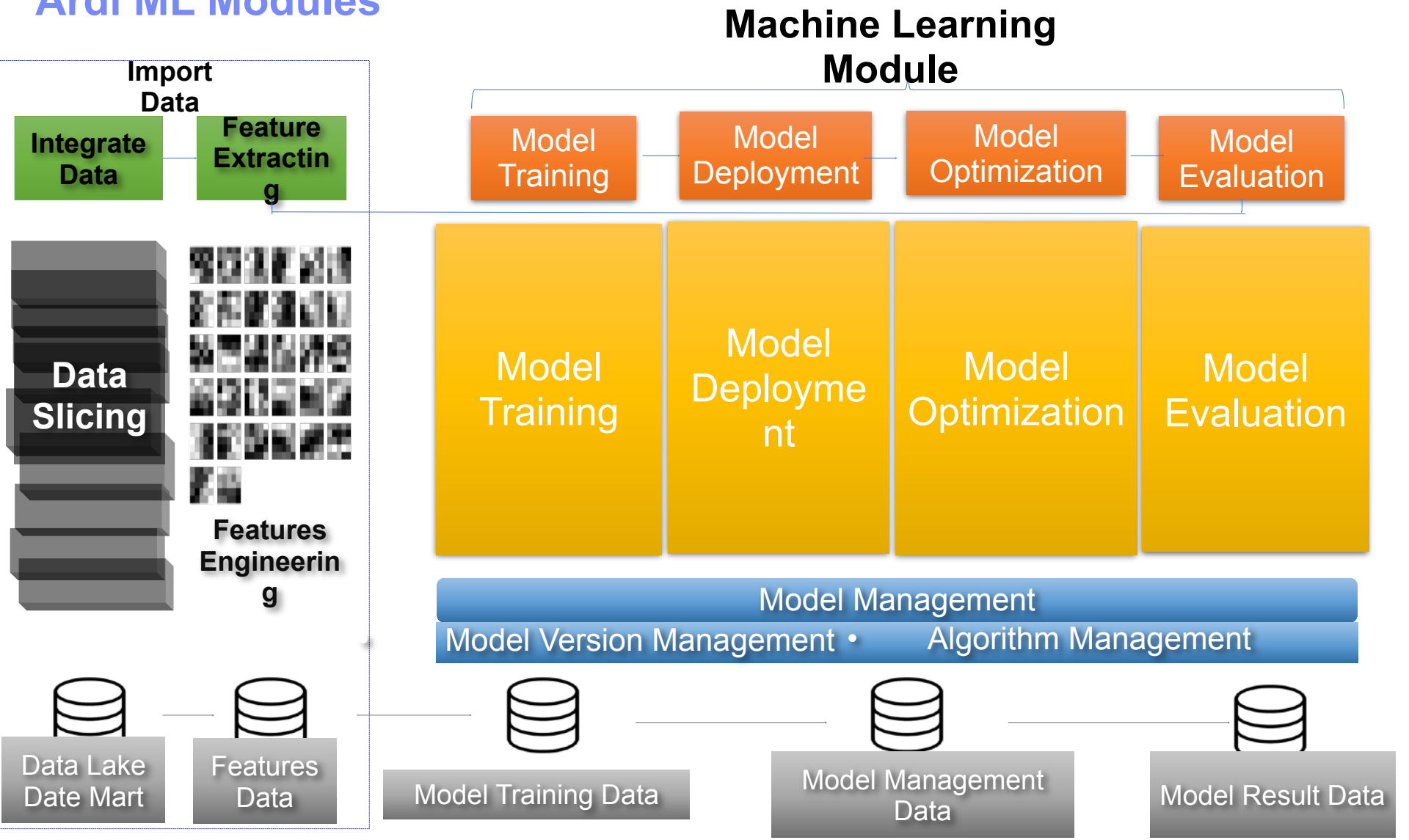
Figure 6-1. Nonnative graph processing engines use indexing to traverse between nodes

Use Relationships, not indexes, for fast traversal



Ardi Autonomous Model Optimizing System (AMOS)

Ardi ML Modules



Machine Learning Module

Model Training– Provides convenient functions such a features importing and preprocessing to model developer，user can choose machine learning model and algorithm and tune parameters flexibly

Model Deployment– Support user to set the frequency of model execution，model deployment and execution。

Model Optimization – Support users to optimize model flexibly

Model Evaluation– Support general evaluation criteria to regression and classification like accuracy and recall。

Model Management– Integrated supports of importing various features data, choosing model type, saving model, setting access right and deployment. Support importation of models trained on outer platforms. Support automatically generate version of models.

Algorithms Support

Classification

- Support Vector Machine
- XGBoost
- LightGBM
- Random Forest
- Decision Tree

Regression

- Ordinary Linear Regression
- Ridge Regression
- Logistic Regression

- **Clustering**
  - K-means
  - Birch
- **Deep Learning (via TensorFlow)**
  - Insert/delete layers
  - Recurrent Neural Network (RNN)
  - Deep neural network (DNN)
  - Convolutional neural network (CNN)

Modeling

- **Model**
  - Random seed
  - Choose parameters automatically
  - Supporting method of parameters choosing ： grid search, Bayesian optimization, random search, gradient boosted tree optimization, sequential optimization
  - Model training
    - Support manually tune parameters
  - Show the report of model training timeline and parameter tuning
- **Validation Method**
  - Hold out
  - Cross validation
  - Time series split
- **Model Evaluation Criteria**
  - Accuracy
  - Recall
  - precision
  - log_loss
  - jaccard similarity
  - confusion matrix
- **Prediction**
  - Predict the data without label
  - Save as CSV
  - Generate a final report

Ardi Autonomous Learning System

ENVIRONMENT ADDITION

DATA DESCRIPTION

MODEL GENERATION

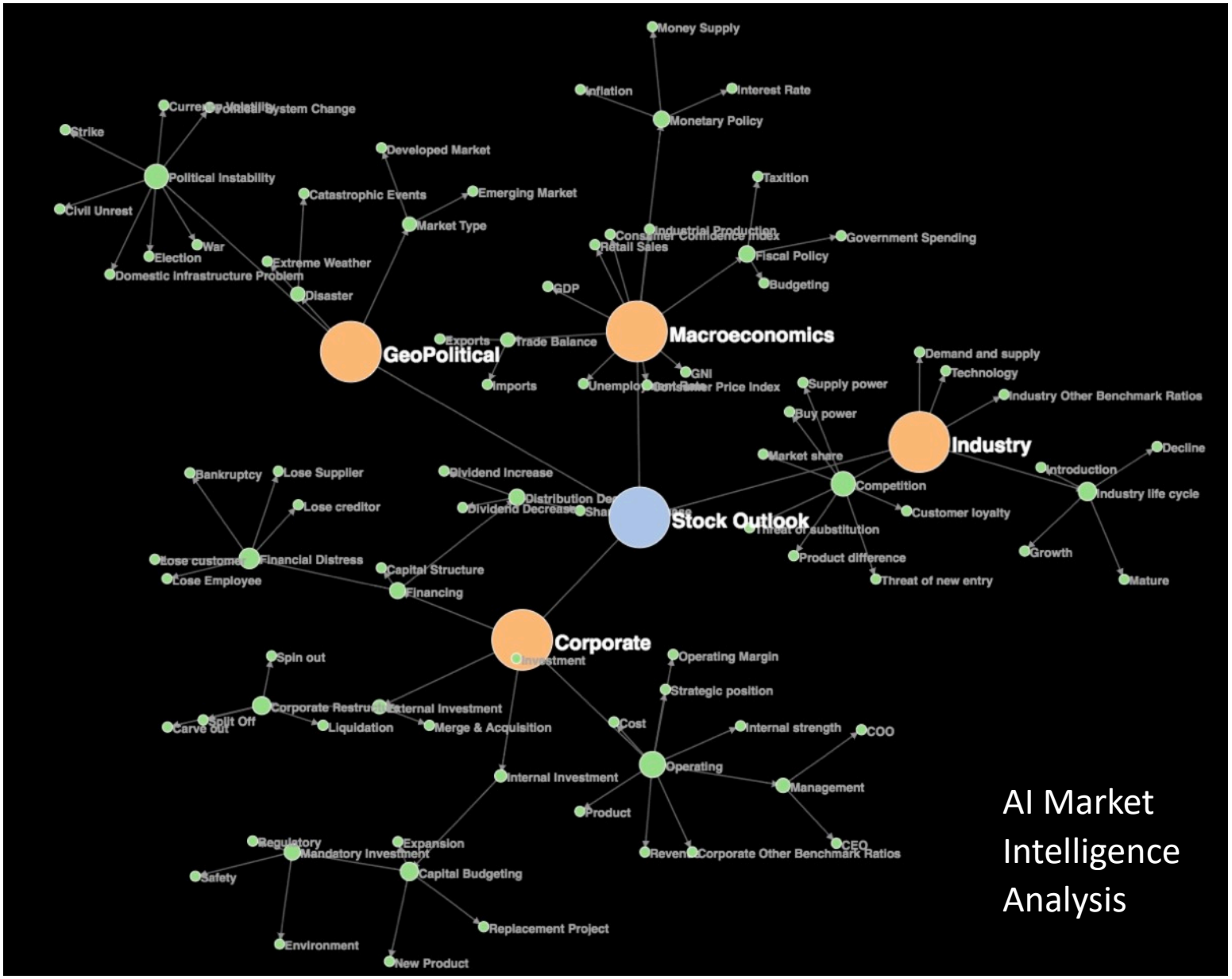
STRUCTURE OPTIMIZATION

ANALYTICS

MODEL UPDATE

MANAGEMENT

Bayesian Network Exploration



AI Market Intelligence Analysis

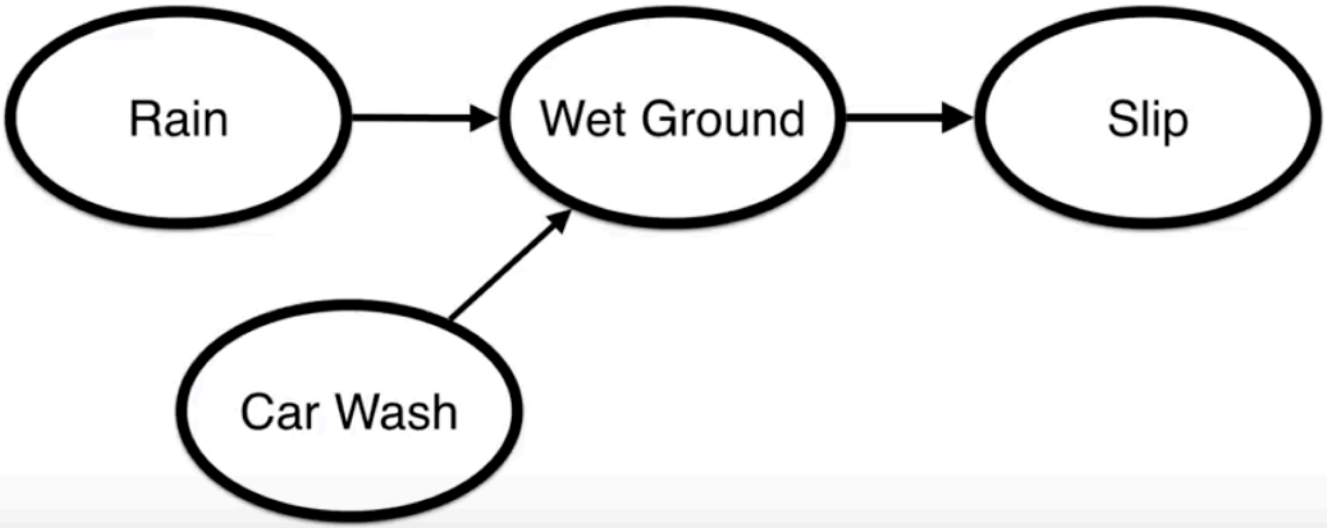
Chain together probabilities

Conditional probabilities represents parent-child relationship

$$P(child \mid parent_1, parent_2, \ldots)$$

$$P(Wet\ Ground \mid Car\ Wash, Rain),\ P(Slip \mid Wet\ Ground)$$

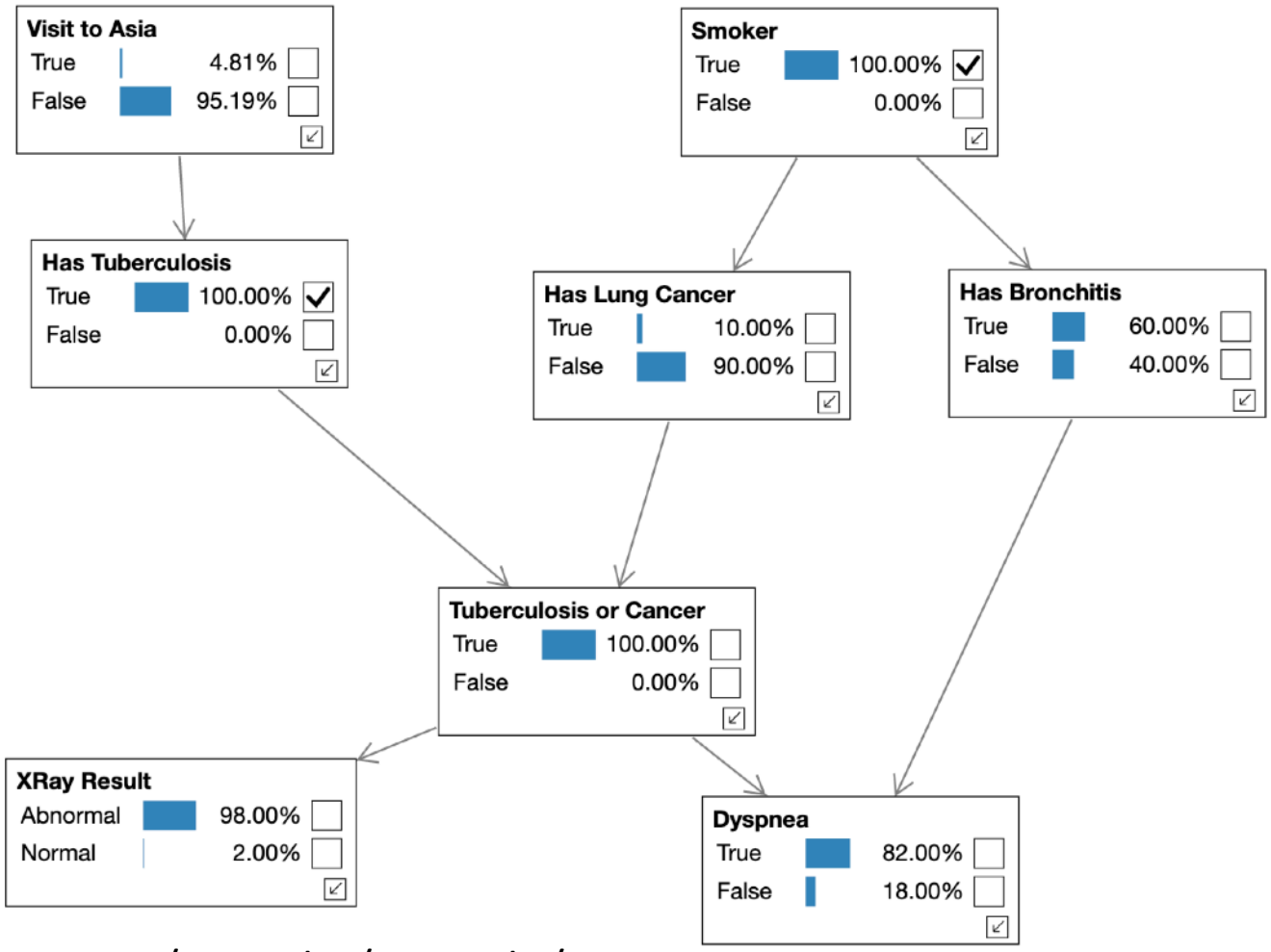
P(R, W, S, C) = P(R) P(C) P(W | C, R) P(S | W)



https://www.youtube.com/watch?v=TuGDMj43ehw&feature=youtu.be

Example — diagnosis

e.g. person is smoker and has tuberculosis (click on checkmark to choose true or false) increases chance of seeing anomalies in xray result and higher chance of having dyspnea compared to the baseline



https://www.bayesserver.com/examples/networks/asia

Creating a Bayesian Network

- We want to generate the network, but we may have only sets of observations for each condition or nodes.
- But it's a NP-Hard problem since we have to go through all different permutations of nodes and directed edges. The time complexity grows the more nodes you have. ($O(2^{n(n-1)})$)
- We could develop certain heuristics to reduce this search space
- We use K2 scoring as a metric to see how well the sample data fits well with a generated network. We iterate through several permutations of generated networks and pick the one with the best score.

```
2020-01-31 03:41:26,693::Current iteration got K2 score: -2853.3557535380787
2020-01-31 03:41:26,949::Finished running with best network K2 score: -2357.270191418432
```

Show best score and plot best network

```
In [6]: print('Best K2 score: {}'.format(best_score))
        Best K2 score: -2357.270191418432

In [7]: best_network_obj.plot_model()
```